VAX 8800 Family

Digital Technical Journal

Digital Equipment Corporation

$15

Number 4

February 1987

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

Changes of address should be sent to Digital Equipment Corporation, attention: Media Response Manager, 200 Baker Ave., CFO1-1/M94, Concord, MA 01742.

Comments on the content of any paper are welcomed. Write to the editor at Mail Stop HL02-3/K11 at the published-by address. Comments can also be sent on the ENET to RDVAX::BEANE or on the ARPANET to BEANE%RDVAX.DEC@DECWRL.

ISBN 1-55558-001-7

Documentation Number EY-6711E-DP

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

Cover Design

This issue features the VAX 8800 family. Our cover depicts the growth of a chambered nautilus as a metaphor for the growth of the VAX family. As those chambers spiral from the center, so the power of the VAX family grows from the MicroVAX systems, through the VAX 8200 and 8300 CPUs, to the new VAX 8800 multiprocessor. The image was created using the Lightspeed system.

The cover was designed by Deborah Falck, Eddie Lee and Tsuneo Taniuchi of the Graphic Design Department.

Contents

Editor's Introduction

Richard W. Beane
Editor

This issue features papers about the design of the VAX 8800 family of CPUs, written by members of the design team. The technology used in Digital's latest high-end machine, the VAX 8800 multiprocessor, also forms the basis for the other three family members: the 8700, 8550, and 8500 CPUs.

Bob Burley's overview relates the processes used in the 8800 design and the functions of the memory interconnect (NMI), the VAXBI I/O bus, and the four logic boxes forming the five-stage pipeline. The early discovery of design flaws and the use of automated tools helped to achieve an aggressive completion schedule.

The micromachine implements the microarchitecture and contains four of the five pipeline stages. Sudhin Mishra describes how microinstructions are handled, emphasizing the use of microbranches and microtraps to ensure coherency.

The VAX 8800 clock system, discussed by Bill Samaras, was designed using an automated timing verifier. He describes the trade-off between using the verifier and maximizing the accuracy of timing signals by minimizing their skew.

The C Box and the M Box are two parts of the pipeline. John Fu, Jim Keller, and Ken Haduch describe the C Box's no-write allocate cache and the delayed-write algorithm that ensures correct write-through. The C Box must also handle pipeline stall conditions and maintain data coherency between processors. The M Box handles read and write requests for the memory arrays. Paul Natusch, Dave Senerchia, and Gene Yu explain how the designs of the NMI and the cache affected their design, and why they used TTL in the memory controller.

The VAX 8800 family does not have a separate floating point accelerator. As John Zurawski, Kathy Pratt, and Tracey Jones point out, however, a custom ECL unit achieves high performance through the normal datapaths. Thus less hardware is needed, and operands are fetched faster.

I/O devices are linked to the CPU by the VAXBI bus. In his paper, Jim Janetos discusses the NBI adapter, which contains logic to handle CPU references and DMA requests. Then Paul Wade describes how the VAXBI design team had to abandon the traditional approach and use a variety of techniques to specify the bus. Some chip problems were resolved only after a thorough analysis of the physical configuration.

Jerry Brand and Mike Kement discuss the importance of using ground correctly as a signal conductor to achieve high performance. They describe the sources of ground-related noise in the CPU, and what they did to isolate and control those sources.

Many VMS features support multiprocessing. Stu Farnham, Mike Harvey, and Kathy Morse first describe the hardware that supports multiprocessing, then the interlocked instructions, exception handlers, and traps that implement VMS multiprocessing. To show how multiprocessing decreases execution time, Gabriel Bischoff and Steve Greenberg converted the SPICE circuit simulator into CAYENNE, a parallel program. They created master and slave processes that ran CAYENNE 1.7 times faster than SPICE.

The final two papers relate some of the automated tools and techniques used on the 8800 project. Dennis Bak first describes building the CAD suite from existing tools, newly developed ones, and modifications. The methodology was truly innovative, serving as a framework for future projects. Then Andy Matthews discusses the on-line system that transformed CAD data into specifications used by Manufacturing. This system minimized the product start-up time by eliminating paperwork.

Dick Beane

Biographies

Dennis T. Bak Dennis Bak is a principal software engineer in the Advanced VAX Development Group. As a project leader, he is currently developing new CAD tools to improve designer productivity on future design projects. In other positions, Dennis performed configuration testing for PDP-11 and VAX systems. Prior to joining Digital in 1980, he worked as a research engineer at Ford Motor Company, doing advanced development on electronic engine-control systems. Dennis earned a B.S. degree in electrical engineering from the University of Michigan in 1974.

Gabriel P. Bischoff In 1985, Gabriel Bischoff joined Digital after receiving a Diploma of Engineer and a Diploma of Advanced Studies in device physics from the Ecole Centrale de Lyon (1980) and a Ph.D. degree in E.E. from Cornell University (1985). As a senior software engineer in the Semiconductor Engineering Group, he is investigating the application of parallel computing architectures for VLSI CAD tools, particularly circuit simulators. Gabriel developed a parallel version of the circuit simulator SPICE for shared-memory multiprocessors. A member of IEEE, he has published papers on device modeling and circuit simulation.

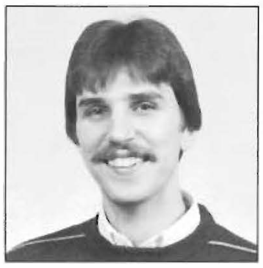
Gerald J. Brand Jerry Brand is a principal engineer currently developing high-density, high-availability power systems. Prior to working on the power and packaging team for the VAX 8800 family, he designed two MPS power modules that are widely used in Digital's products. Before joining Digital in 1980, Jerry worked for over 14 years in disciplines ranging from oceanography to gas-turbine instrumentation. He holds a B.S.E.E. degree from the University of Illinois and participated in the M.S.E.E. program at the University of New Hampshire. Jerry teaches circuit analysis and electronics in the continuing education program at the University of Lowell.

Robert M. Burley As a senior product management manager, Bob Burley was the engineering product manager for the four systems in the VAX 8800 family. As a program manager in the LSI Acquisition and Test Group, he was responsible for relations with external vendors and acquiring technologies for the advanced gate arrays used in new CPU designs. Prior to joining Digital in 1980, Bob was a product and business development manager at Colt Industries, Inc., and a product and manufacturing manager at Scott Paper Company. He earned his B.S. degree in mathematics and economics from Hobart College in 1965.

Stuart J. Farnham As a principal software engineer in the VMS Development Group, Stu Farnham is currently working on future directions in multiprocessing. Earlier, he provided VMS support at the corporate level for Software Services. Stu was a developer and instructor for the VAX/VMS Systems Seminar. He joined Digital in 1982 after working as a software engineer at Pitney Bowes, Inc.

John Fu Currently earning his M.S. degree in computer science at the University of Illinois, John Fu was a principal engineer on the VAX 8800 project. He worked on the design of the C Box and configurations for the VAX 8800 family. Formerly, he worked on large-systems designs at International Computers Limited and on microprocessor control systems for Siemens Limited. John was also a project manager at Systems and Software, Inc. He received a B.Sc. (Hons) in computer science (1977) from the University of Manchester in England. John is a member of the British Computer Society and the IEE in England.

Steven S. Greenberg As a team leader in the CAD Department of the Semiconductor Engineering Group, Steve Greenberg codeveloped the CAYENNE program. An early provider of circuit and process simulators at Digital, he did research in timing verification and circuit simulators. As a Digital industrial fellow at the University of California at Berkeley, Steve performed research on iterated timing analysis. Before joining Digital in 1976, he was a member of the technical staff at RCA and a CAD engineer at Texas Instruments. Steve received a B.S.E.E. degree (1966) from M.I.T. and an M.S.E.E. degree (1979) from Northeastern University. He is a member of IEEE and Tau Beta Pi.



Kenneth J. Haduch In 1974, Ken Haduch joined Digital after earning his Associate in Electronic and Computer Technology degree from the Electronic Institutes, Pittsburgh. He worked as a technician in Manufacturing on the PDP-11/70 and VAX-11/780 CPUs and in Engineering on the DR750 and FP750 designs. Ken helped to develop the C Box as a hardware designer on the VAX 8800 project. He is currently a hardware engineer in the Advanced VAX Development Group, working on the hardware design for a new VAX processor. Ken is also pursuing a B.S. degree from Northeastern University.

Michael S. Harvey Mike Harvey joined Digital in 1978 after receiving his B.S. degree in computer science from the University of Vermont. He worked on developing the RSX-11M and RSX-11M-PLUS operating systems and then led the team that developed the VAX-11 RSX layered product for the VMS system. Since joining the VMS Development Group, Mike has participated in new processor support for the VAX 8300 and 8800 systems, specializing in multiprocessing. As a principal software engineer, he is currently working on future directions for VMS multiprocessing and support for high-end VAX CPUs.

James P. Janetos Jim Janetos is currently studying computer architecture as a graduate student at Purdue University. He joined Digital in 1980 after receiving his B.S.E.E. degree (Summa Cum Laude) from the University of Michigan, where he was elected to Tau Beta Pi. As a design engineer, Jim worked on memory upgrades for the PDP-11/24 and 11/44 systems, on memory system designs, and on dynamic RAM evaluations. On the VAX 8800 project, he initially worked on the diagnostic software for the I/O adapter, the NBI. Later, he designed the NBIB module, one of the two modules in the NBI.

Tracey L. Jones Earning her B.S. degree in computer engineering from Boston University, Tracey Jones joined Digital after graduation in 1982. As a firmware engineer in the Advanced VAX Engineering Group, she wrote a major portion of the microcode that performs floating point operations in the VAX 8800 family of processors. After promotion to senior engineer, Tracey enrolled in Digital's Graduate Engineering Education Program and is now pursuing an M.S. degree in electrical engineering at Brown University.

James B. Keller Jim Keller is the project leader for the instruction-fetch and execution units, the I and E Boxes, and the console for a new VAX processor. On the VAX 8800 project, he worked on the design of the C Box. Prior to joining Digital in 1982, Jim worked on fiber optics and the designs of several microprocessor boards at Harris Corporation. He earned a B.S. degree in electrical engineering in 1980 from Pennsylvania State University, where he was elected to Eta Kappa Nu. Jim has applied for three patents on the technology in the VAX 8800 design.

Michael W. Kement Mike Kement is a senior design engineer in the Power System Technology Group, currently working on EMI and EMC. He was the design engineer for the power system on the VAX 8800 project. Mike has worked on the power systems of many products since joining Digital in 1974, including the LA36 and LA180 terminals, the PDP-11/44, VAX-11/780 and 11/750 systems, and the VAX 8600 CPU.

Andrew J. Matthews As a senior software manager in the Advanced VAX Systems CAD Group, Andy Matthews is currently automating the CAD to CAM transition. He has managed the development of surface-mount CAD processes and a pilot program of advanced CAD to CAM data methods. Andy designed the prototype and first release of VLS, the VAX layout software Digital uses for module design. He worked for Adage, Inc., as the manager of applications programming before coming to Digital in 1977. Andy holds a B.S. degree in C.S. and M.E. (1968) from Boston University. He has presented two papers at the Design Automation Conference.

Sudhindra N. Mishra Sudhin Mishra is a project leader in the Advanced VAX Development Group, currently developing a design verification CAD tool. As a principal engineer on the VAX 8800 project, he designed and implemented most of the I Box and originated the system-level simulation of the CPU. Before joining Digital in 1982, he was a senior research engineer at Prime Computers, Inc. Sudhin has worked on projects ranging from radar and heat-seeking missiles to computers. He earned a B.Sc. degree in engineering from Ranchi University and an S.M. in E.E. and C.S. from M.I.T. Sudhin has applied for a patent on the technology in the VAX 8800 design.

Kathleen D. Morse As a consulting software engineer, Kathy Morse is responsible for all low-end CPUs and peripherals. She is also one of the designers for future directions in VMS multiprocessing. Kathy provided VMS support for the VAX-11/782 and MicroVAX I and II systems, and the MA780 memory. She joined Digital after receiving her B.S.C.S. degree (1976) from Worcester Polytechnic Institute, where she also earned her M.S.C.S. degree (1985). Kathy is a member of IEEE, the Professional Council, ACM, Tau Beta Pi, and Upsilon Phi Epsilon. She has published in the Computer Measurement Group's Conference Proceedings, the *Digital Technical Journal,* and *DATAMATION.*

Paul J. Natusch As a principal hardware engineer, Paul Natusch is currently managing the hardware development for a new VAX processor in the Advanced VAX Development Group. On the VAX 8800 project, he was a member of the memory system team and later took over as its leader. Earlier, he worked on an upgrade to the VAX-11/750 memory controller, which expanded it from 2MB to 8MB. Paul joined Digital in 1980 from Storage Technology Corporation, where he was a diagnostic engineer. He received his B.S.E.E. degree from Cornell University in 1979 and an M.B.A. degree from Northeastern University in 1985.

Kathleen L. Pratt Educated at Rensselaer Polytechnic Institute, Kathy Pratt came to Digital after receiving her B.S. degree in computer and systems engineering in 1980. She worked on hardware designs for networks in the Local Area Networks Group, then on the design of the floating point hardware for the VAX 8800 central processor in the Advanced VAX Development Group. Kathy is currently a senior engineer working on the floating point design for a new VAX processor.

William A. Samaras Bill Samaras is a principal engineer working to design a new VAX processor. He joined Digital in 1982 to design the clock system on the VAX 8800 project. Formerly, at Accutest Corporation, Bill designed VLSI testers and timing systems. He holds an Associates degree (1973) from Northern Essex Community College, and B.S. degrees in engineering technology (1975) and electrical engineering (1976), both from Southeastern Massachusetts University. Bill teaches digital electronics for continuing education at the University of Lowell. He has applied jointly for a patent on the technology in the 8800 clock system.

David C. Senerchia Dave Senerchia is currently a senior engineer in the Electronic Storage Development Group. He is a member of the design team working on the main memory for a new mid-range VAX system. On the VAX 8800 team, Dave designed the initial array module for main memory and participated in the architecture and design of the memory system, the M Box. He joined Digital in 1982 after earning a B.S. degree in electrical engineering from Washington University.

Paul C. Wade As a principal engineer, Paul Wade is working on advanced development for future VAX CPUs. He was responsible for the electrical design, verification, and testing for the VAXBI bus. Paul also designed parts of the VAX 8200 system. Before joining Digital in 1980, he worked as a project engineer at Microwave Semiconductor Corporation, RCA, and Lockheed Electronics. Paul earned a B.S.E.E. degree (1973) from Newark College of Engineering. He holds a patent on gallium arsenide technology and has written nine papers on that topic. One paper won the Beatrice Winner Award at the 1980 ISSCC.

Cheryl A. Wiecek Cheryl Wiecek is the engineering manager of the Systems Architecture Group and is responsible for the VAX architecture and a number of Digital's interconnect architectures. She worked on VAX instruction-set characterization and performance simulation for the VAX 8800 CPU. Cheryl also worked on PDP-11 performance simulation after coming to Digital in 1978. She was a programmer/analyst at the Connecticut Education Association and taught mathematics in Connecticut. Cheryl holds a B.A. degree in mathematics (1974) and an M.S. degree in computer science (1979) from the University of Connecticut. She has published five papers on computer performance in ACM and IEEE journals.

Eugene L. Yu Gene Yu is a senior design engineer in the Workstation Engineering Group at Palo Alto. On the VAX 8800 project, he designed the memory system interface to the memory interconnect, the NMI. Before joining Digital in 1982, Gene worked at Prime Computer as a hardware designer on their 400 and 9900 systems, and at Data General Corporation on Nova products. He earned a B.S. degree in electrical engineering from the University of Massachusetts. Gene has applied for a patent as coinventor of the NMI and memory design for the VAX 8800 CPU.

John H.P. Zurawski John Zurawski is a consulting engineer working as the project leader for computer arithmetic in the Advanced VAX Development Group. He led the team that designed the floating point strategy and hardware for the VAX 8800 family. John joined Digital in 1982 from the University of Manchester, where he was a post-doctoral research associate. He holds a B.Sc. degree in physics (1976), and M.Sc. (1977) and Ph.D. (1980) degrees in computer science, all from the University of Manchester. A member of IEEE, John has published four papers on computer technology.

Foreword

Donald J. McInnis
Group Manager,
Advanced VAX Engineering

Since the announcement of the VAX-11/780 system in November 1977, Digital Equipment Corporation has steadily expanded the VAX family with new VAX products: the VAX-11/750, VAX-11/730, MicroVAX I, VAX-11/725, VAX-11/785, VAX 8600, MicroVAX II, VAX 8650, VAX 8200, and VAX 8300 systems. The market acceptance of the VAX family has been excellent across almost all computing applications. This remarkable and steady increase in the use of VAX systems creates a continuous demand by the VAX customer base for enhanced products across all segments of the computing industry. In the fall of 1982, the development team for the 8800 project (known internally as "Nautilus") was assigned the responsibility of designing new systems to enhance the mid-to-high end of the VAX family.

This issue of the *Digital Technical Journal* represents a sampling of the types of design engineering that went into the VAX 8800 family. It takes an amazingly large number of different engineering disciplines to design and manufacture a product of this complexity. As time moves on, each successive development project seems to require a bigger investment in a larger number of disciplines to produce a product attractive to the marketplace. It is unfortunate that neither time nor space permits us to give proper visibility to all the design, manufacturing, and customer-service engineering efforts that led to the shipment of the VAX 8800 family.

The VAX 8800 family consists of four new processors: the VAX 8800, VAX 8700, VAX 8550, and VAX 8500 CPUs. The VAX 8800 family and the VAX 8200 system introduced a major new I/O bus, the VAXBI. We also introduced a completely new set of I/O adapters for the VAXBI bus, which will be the new foundation I/O channel for many future mid- to high-end VAX systems. The VAXBI bus will replace the UNIBUS on this class of system. The VAXBI offers a six-fold increase in performance and substantially better reliability and maintainability features in comparison to the UNIBUS.

The 8800 represents a significant advance into new areas of high-performance computing for the VAX family. A customer can replace a VAX-11/780 CPU with a VAX 8800 CPU in the same footprint and effect an order of magnitude increase in the amount of work done. The VAX 8500 CPU is really a replacement product for the VAX-11/785 CPU kernel. However, the 8500 has the same price, twice the performance, and one-third the footprint.

To produce a product that has a good price/performance ratio in the marketplace, you have to push hard on some dimensions of technology. A number of new pieces of technology were introduced on the VAX 8800 project, such as the 22-layer backplane and a 480-pin, zero insertion force connector. In the VLSI technology area, one 8800 includes a total of 186 emitter-coupled logic (ECL) gate arrays and a total of 28 custom-designed ECL parts.

The cycle time of a VAX CPU is a large determinant in its performance. The challenge of meeting a 45-nanosecond cycle time (versus 200 nanoseconds for the 11/780) required significant advancements in technology implementation and in CAD tools for analysis.

Enhancements were made to the base operating system software for the VAX 8800 processor. These software enhancements represent a basic technological change that is available to our customers. The VMS operating system was improved significantly to provide much better throughput for customers using the VAX 8800 dual processor as a general-purpose system. The ULTRIX-32 operating system was enhanced to support tightly coupled multiprocessing. Software library structures were also developed for customers who might want to improve the throughput of a single job by decomposing it to run in parallel on the tightly coupled dual processors of an 8800.

To meet the performance goals, the overall design of the VAX 8800 system is necessarily quite complex and was potentially difficult to implement quickly and correctly. We understood this from the beginning of the project, based on our understanding of the experiences of previous projects (e.g., the VAX-11/750, VAX 8600, and J11 VLSI CPU chip projects). To manage that complexity in a timely manner, we selected some key strategies and stuck with them through the completion of the project. They proved to be very successful since the hardware prototypes were relatively error free, and the manufacturing start-up was very smooth and rapid. Some of these strategies are as follows:

- The project followed a structured design methodology that ensured the completion of comprehensive specifications before any detailed design was done.
- We made a large investment in our CAD team and in CAD tools to automate the design process.
- The basic design was managed by a chief architect.
- The system was simulated extensively before we built any hardware. (We finished the project with 14 VAX-11/780 and 11/785 systems in our cluster. During our peak simulation effort, however, over 30 dedicated VAX systems were used for a period of several months.)
- Since many different engineering and manufacturing locations were involved, we made extensive use of Digital's worldwide network for electronic mail and data exchange.

A more important factor than any of the above examples, however, was the people who worked on the project. We attempted to build an excellent team that worked well together. The attribute of teamwork and the willingness of people to have a broad engineering focus proved to be invaluable, especially in the simulation and prototyping phases. The core management team started with very experienced people, most of whom had VAX-11/780 or VAX-11/750 development experience: Sas Durvasula, VAX 8500 project manager; John Hittell, manufacturing manager; Steve Jenkins, engineering manager; Nancy Kronenberg, VMS engineering; Bob Kusik, CAD manager; Steve Omand, customer service engineering; and Bob Stewart, chief architect. Many contributors at the next level also had similar backgrounds, and all remained in place for the duration of the project. This continuity was a major factor in completing a very successful project and a very successful family of products.

Robert M. Burley

An Overview of the Four Systems in the VAX 8800 Family

The VAX 8800 multiprocessor and the VAX 8700, 8550, and 8500 systems all derive from the same fundamental design. Their sustained applications throughput ranges from 3.0 to 12 times that of the VAX-11/780 system. In the design process, automated tools helped to correct design bugs early. ECL technology and a two-phase clock system achieve a 45-nanosecond cycle time. Microinstructions are processed simultaneously through four logic boxes that implement a five-stage pipeline. A high-speed memory interconnect, the NMI bus, links CPUs to memory and the I/O subsystem, which connects to VAXBI buses. Many reliability features, including extensive diagnostics, are implemented.

Design work on the VAX 8800 system began in September 1982 and concentrated on developing a balanced, high-performance system based upon the use of ECL components and multiprocessing. Although performance was the primary product goal, many technology, packaging, and implementation decisions reflected the equally pressing business requirements for reliability and ease of manufacturing.

The flexibility of the design ultimately spawned four CPU systems: the VAX 8800, VAX 8700, VAX 8550, and VAX 8500 models. These systems share many common functional and design attributes yet maintain noticeable implementation differences in the areas of performance, multiprocessing, expansion capability (memory and I/O), and packaging. As a result of these implementation variations, the sustained applications throughput (SAT) rates for these systems range from approximately 3.0 to 12 times the rate for a VAX-11/780 system. Sustained applications throughput is more indicative of usable performance for a given system than the more frequently reported peak numbers that can be derived from ideal or biased conditions. Table 1 compares the physical and performance attributes of these four VAX processor systems.

Design Environment

Traditional design environments have placed the greatest emphasis on discovering and eliminating design errors in the physical hardware. The complexity of the VAX 8800 design coupled with the new technologies involved would have created costly delays in the development schedule had traditional approaches been used. Early in the project, goals were defined to identify logic design problems and to solve all timing problems through the use of extensive design verification tools.

A hierarchical design and simulation environment allowed the engineers to move freely throughout the design at any level from gates, layouts, and behavioral models through complete system simulation and timing verification. Considerable computing resources were required to allow that freedom. This environment, with its carefully managed libraries and databases, allowed this work to be done before any hardware was actually assembled.[1] As a result, the design matured within our VAXcluster systems, evolving to hardware prototypes only after it was essentially complete and stable. In addition to the expected savings in prototype costs and a reduction in overall development time, the pervasive use of software tools significantly shifted the traditional debug effort to an earlier point in the design process. Cumulative bug-detection plots were used extensively to provide insight into the stability of the design.

The effect of this shift was to provide stable, early prototypes for extensive system characterization and testing, leading to earlier design

Table 1 CPU and Memory Attributes of the VAX 8800 Family

	VAX 8500	VAX 8550	VAX 8700	VAX 8800
CPU Attributes				
SAT (compared to VAX-11/780)	3.5	6.0	6.0	10.0 to 12.0
Cycle Time	45 ns	45 ns	45 ns	45 ns
Number of Processors	1	1	1	2
Upgrade Potential	To 8550	None	To 8800	None
Writable Control Store (Words)	15K	15K	15K	15K in each CPU
User Control Store (Words)	1K	1K	1K	1K in each CPU
Microword Size	143 Bits	143 Bits	143 Bits	143 Bits
CACHE Size	64KB	64KB	64KB	64KB (in each CPU)
Internal Datapath	32 Bits	32 Bits	32 Bits	32 Bits
Instruction Buffer Type	16 Byte Look Ahead	16 Byte Look Ahead	16 Byte Look Ahead	16 Byte Look Ahead in each CPU
Maximum Total I/O Data Rate	16MB/s	16MB/s	Over 30MB/s	Over 30MB/s
Maximum I/O Channels	2	2	4	4
Memory Attributes				
Maximum Physical Memory Size	80MB	80MB	128MB	128MB
Cycle Times:				
Hexword Read (256 bits)	495 ns min. 1260 ns max.	495 ns min. 1260 ns max.	495 ns min. 1260 ns max.	495 ns min. 1260 ns max.
Octaword Write (128 bits)	270 ns min. 540 ns max.	270 ns min. 540 ns max.	270 ns min. 540 ns max.	270 ns min. 540 ns max.
Longword Write (32 bits)	135 ns min. 495 ns max.	135 ns min. 495 ns max.	135 ns min. 495 ns max.	135 ns min. 495 ns max.

acceptance. This strictly controlled design environment allowed us to complete physical debug along with the required system evaluation and testing in only eight months.

In a software-intensive design environment, the production of actual hardware is deferred somewhat in favor of design stability, resulting in a slightly longer soft-design period. The delay in hardware availability, however, is more than balanced by the stability of the hardware prototypes, which can then be accelerated through the evaluation and qualification-testing phases. The design schedule recovers during these later phases, and substantial cost savings are realized because fewer engineering changes are made and stable manufacturing can begin quickly.

CPU Design Overview

The VAX 8800 family of designs were structured around the functional elements, or "boxes," of the system. The CPU, memory, I/O, and bus subsystems were all matched to provide the necessary system balance. One simple model is to treat performance as a function of two variables:

the instruction execution rate, and the amount of "work" each instruction can perform. The design of the VAX 8800 family focused on what we call the "short tick" approach to achieve the necessary, sustained performance.

In this approach, the instruction and data streams are kept simple and are executed quickly. Any design trade-offs were resolved in favor of speed and simplicity, thus reducing design complexity. The use of high-speed custom and semicustom VLSI components combined with several new internal bus architectures resulted in a family of processors with a 45-nanosecond (ns) cycle time. All models employ a five-stage instruction execution pipeline, integral floating point acceleration (F, D, G, H formats), and the VAXBI bus as the primary I/O subsystem. The extensive use of microcode controls with minimal hardware assist augments current performance while providing flexibility for future enhancements. The block diagram in Figure 1 (using the VAX 8700 and VAX 8800 systems) illustrates the key functional elements common to the VAX 8800 family design.

Technology

The raw speed, off-chip drive capabilities, and availability of bipolar emitter-coupled logic (ECL) logic components provided the most straightforward means of achieving the desired performance of the VAX 8800 family. Most logic is implemented in 1200-gate ECL arrays. Custom logic chips designed by Digital provide further performance gains for floating point operations and general-purpose registers. The cache is implemented in 10-ns and 15-ns ECL RAMs. Nine-layer, controlled-impedance CPU logic modules and a 22-layer, controlled-impedance CPU backplane were developed to meet the signal-integrity and signal-propagation requirements crucial to an ECL design. Other multilayer backplanes were designed for the private memory array bus and I/O subsystems.

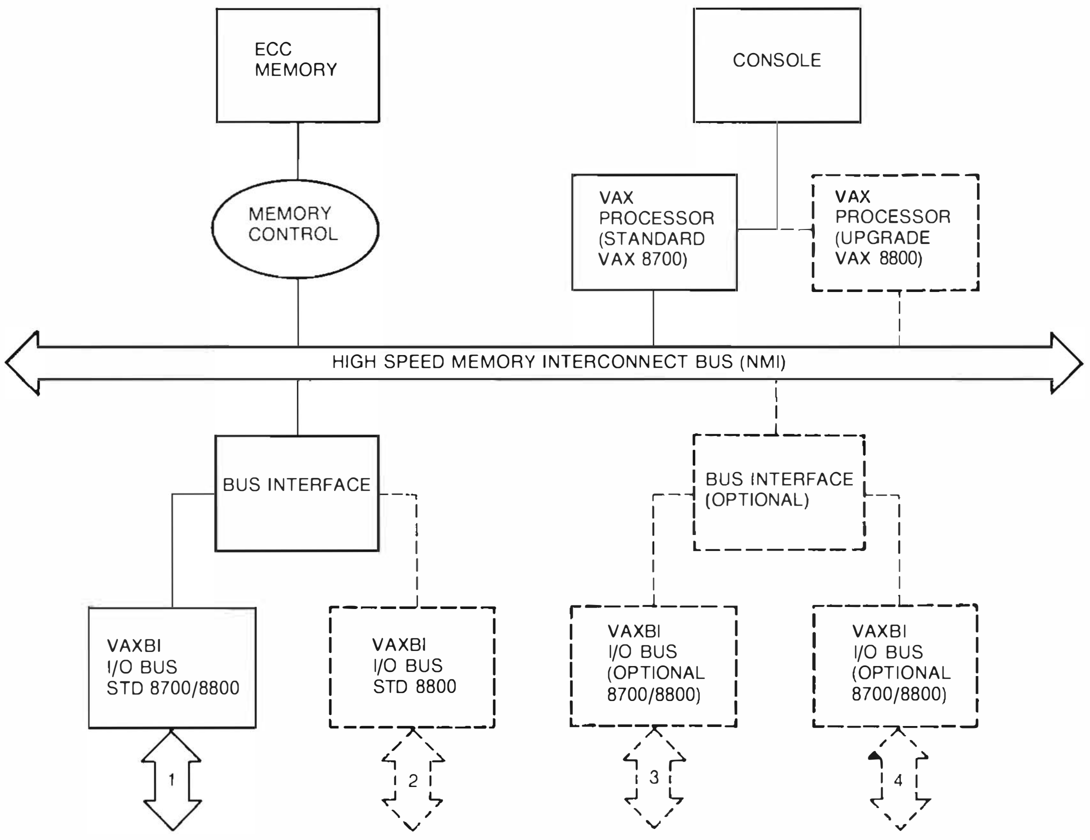

ECC MEMORY
CONSOLE
MEMORY CONTROL
VAX PROCESSOR (STANDARD VAX 8700)
VAX PROCESSOR (UPGRADE VAX 8800)
HIGH SPEED MEMORY INTERCONNECT BUS (NMI)
BUS INTERFACE
BUS INTERFACE (OPTIONAL)
VAXBI I/O BUS STD 8700/8800
VAXBI I/O BUS STD 8800
VAXBI I/O BUS (OPTIONAL 8700/8800)
VAXBI I/O BUS (OPTIONAL 8700/8800)
1
2
3
4

Figure 1 VAX 8700/8800 Block Diagram

An innovative scheme of bus bars and ribbon straps routes the appropriate power to each of the backplanes, minimizing cable management problems for system power. The eight CPU logic modules, all memory arrays, and all I/O controllers attach to their respective backplanes by means of zero insertion force (ZIF) connectors, which improve our ability to manufacture and service the system. Figure 2 shows the two different module types (CPU and VAXBI) used in the VAX 8800 family.

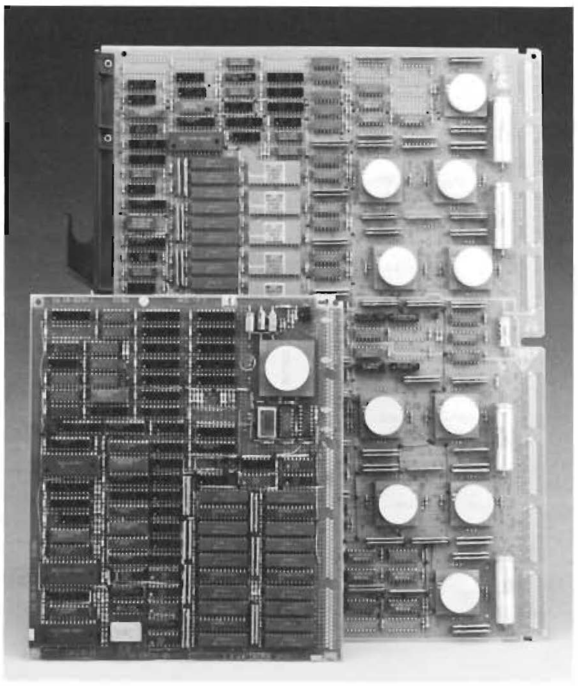
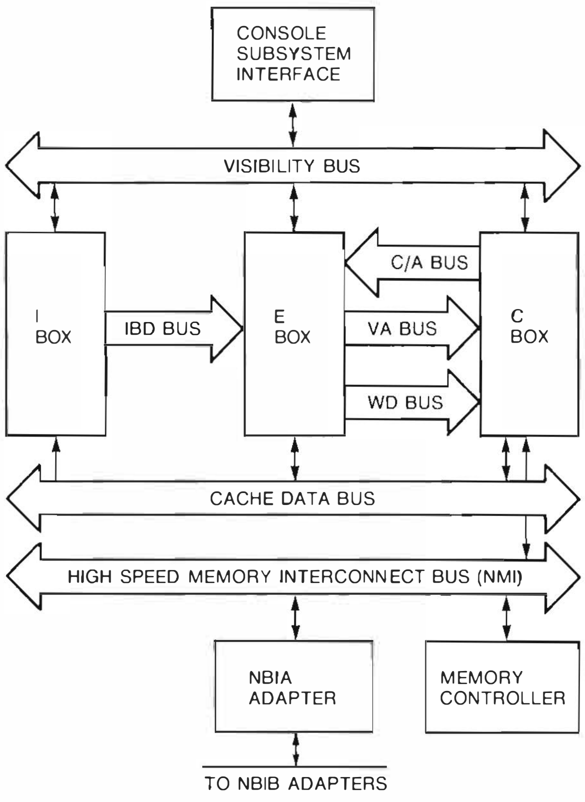


Figure 2 Typical CPU and I/O Modules

An extensive environmental monitoring subsystem, called the EMM, has been implemented throughout the system. The EMM constantly monitors current fluctuations, air flows, and temperature variations, providing warnings at the system console. The EMM can automatically power down the system in the event that safe operating limits are violated.

CPU Subsystems

The designs of the CPUs in the VAX 8800 family are partitioned along the logical functions performed within each processor. There are four logical boxes: the instruction unit (I Box), the cache (C Box), the execution unit (E Box), and the memory subsystem (M Box). Each processor contains these functional units and their related buses. Five buses are implemented within each CPU: the cache/ALU bypass bus, the cache data bus, the instruction-buffer data bus, the virtual-address bus, and the write data bus. Figure 3 is a block diagram of the processor configuration.



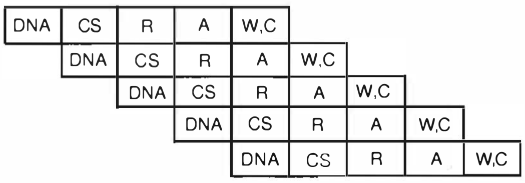
C/A BUS - CACHE/ALU BYPASS BUS
IBD BUS - INSTRUCTION BUFFER DATA BUS
VA BUS - VIRTUAL ADDRESS BUS
WD BUS - WRITE DATA BUS

Figure 3 Processor Block Diagram

A short overview of each functional box follows. Other papers in this issue of the *Digital Technical Journal* and the *VAX Hardware Handbook* contain substantially more detail.[2]

Pipelining the VAX 8800 Family

Pipelining, which functionally involves the E Box, the C Box, and the M Box, is primarily controlled by the I Box. Pipelining is a proven method to improve performance. The incorporation of pipelining, in conjunction with faster microcode instruction execution rates, or cycle times, increases aggregate throughput more than can be achieved by improvements of the cycle time alone. The concept of pipelining is based upon partitioning instruction execution to allow simultaneous operations upon multiple microinstructions. The VAX 8800 family employs a five-stage pipeline. In this design a new microinstruction executes every 45 ns, with five microinstructions executing simultaneously. A simplified schematic of the VAX 8800 family pipeline is represented in Figure 4.



DNA - DECODE/NEXT ADDRESS
CS - CONTROL STORE LOOK-UP (MICROCODE INSTRUCTION)
R - REGISTER READ
A - ALU OPERATION
W,C - REGISTER WRITE, CACHE OPERATION

Figure 4 The Pipeline in the VAX 8800 Family

The I Box

The I Box contains the microcode store and control center and performs five primary functions.

- Buffering the prefetched VAX instruction-stream data received from the cache
- Decoding and controlling the execution of microinstructions
- Monitoring and servicing microtraps, interrupts, and exceptions
- Supplying instruction-stream embedded data
- Interfacing between the console interface module and the processor

For each processor, a writable control store of 16K words by 143 bits is loaded directly from the intelligent console subsystem upon system start. A segment of control store with 1K words by 143 bits, the user-writable control store, is provided for the system user to optimize applications. The logical function of the I Box includes the following:

- The instruction buffer
- The instruction decoder
- The microsequencer
- The condition code and microbranch logic
- The interrupt and processor-register logic
- The file-address generator

Figure 5 depicts the implementation of the I Box.

The C Box

The C Box for each processor is built around a 64-kilobyte (KB) write-through data cache memory that is physically indexed and direct mapped. Functionally, the C Box provides very high-speed physical memory, high-speed address translations, and a communication path for the processor to the NMI bus. The comparatively large cache size was specifically selected to allow large applications to remain fully resident in the cache, substantially reducing memory traffic and processor wait states. The complete C Box implementation includes a 1KB translation buffer, a 64KB cache data store, and an NMI interface. The translation buffer consists of a 1K-entry cache of virtual-to-physical address translations. This translation buffer contains a tag store and a data store organized into 512 process-translation slots and 512 system region-translation slots. Using a portion of the virtual address to compare the tag-store and data-store addresses, the translation buffer concatenates the page frame number with the low-order virtual-address bits to form the physical address for the data store cache.

Data read from the cache data store (a cache "hit") requires no memory request. If the required data is not in the cache data store (a cache "miss"), logic embedded in the NMI interface uses the cache-miss address to spawn a command/address transaction that is sent to the memory subsystem. Upon return, the requested data from memory is passed to the requesting CPU and then placed in the cache data store for subsequent use. This design allows the translation

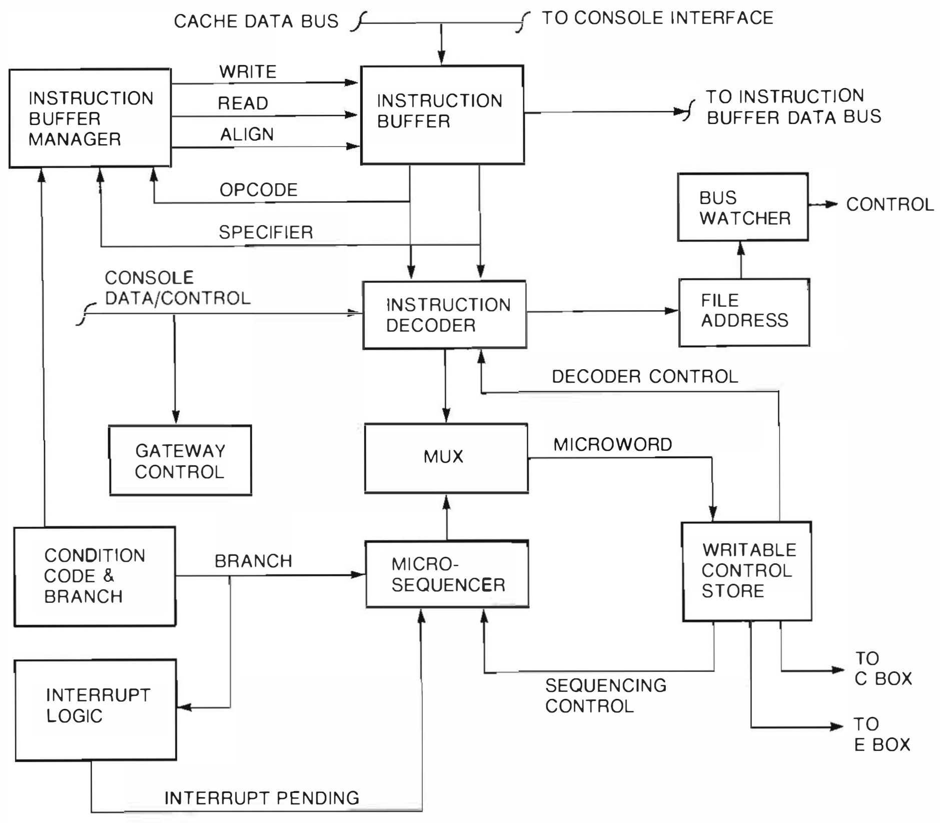


Figure 5 I Box Block Diagram

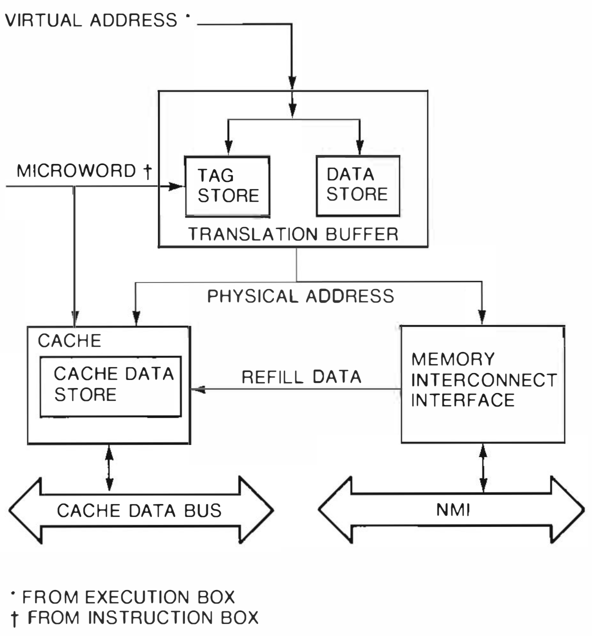


Figure 6 C Box Block Diagram

buffer and the cache data store to be free to process other processor requests until the requested data arrives from memory.

A block diagram of the C Box is shown in Figure 6.

The E Box

The E Box receives data from the I Box and the C Box, processes that data, and returns it to the C Box. The E Box performs five primary functions required by the processor.

- Handles all arithmetic, logical and bit-shift operations
- Maintains the program counter and general registers
- Maintains the processor registers
- Controls data transfers between the C Box, the I Box, and the clock-module registers
- Provides condition-code information to the I Box microsequencer

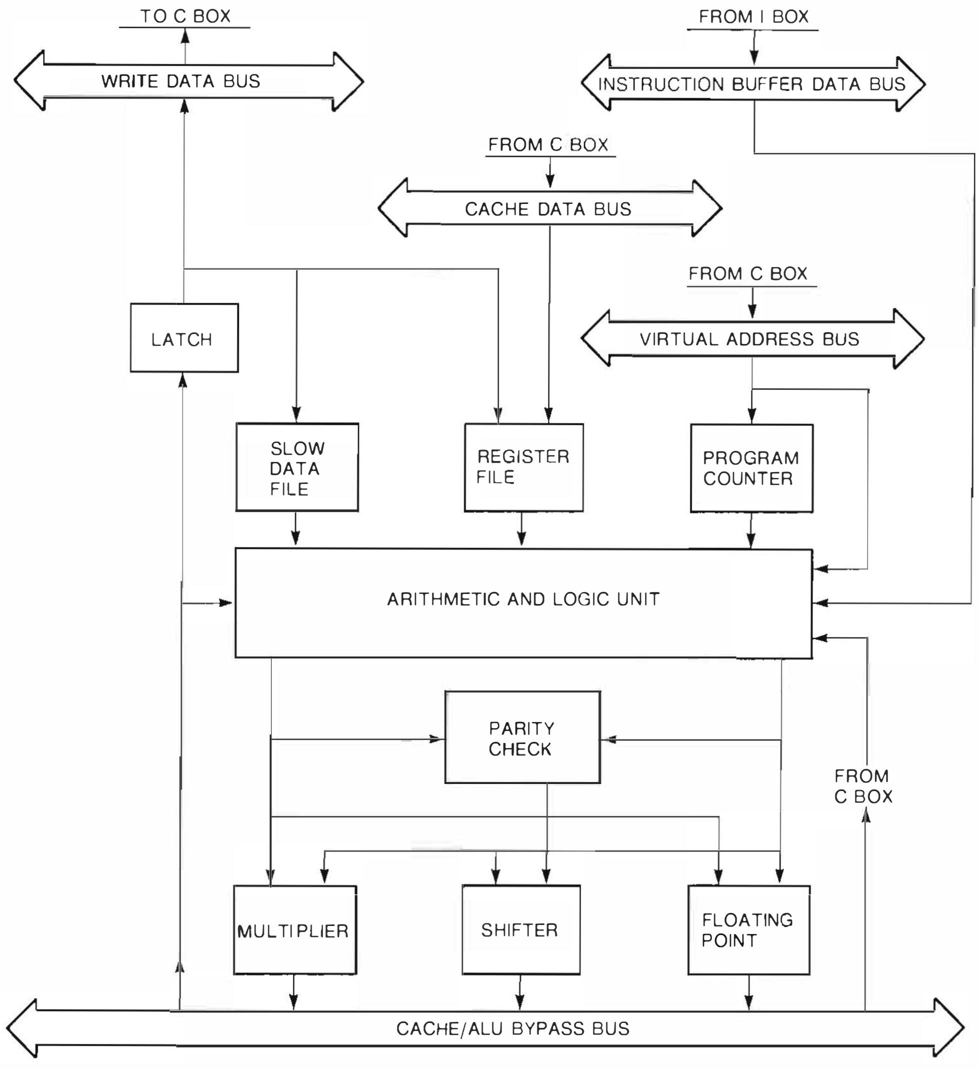


Figure 7 E Box Block Diagram

The major elements of the E Box, located physically on the data-slice modules and the shifter module, consist of a register file, a data file, the program-counter logic, the main ALU, and a shifter. The logic of the E Box includes integral floating point operations that are optimized and a 64-bit multiplier (implemented in custom-designed VLSI chips) that augments the speed of both integer and floating point multiplication. Figure 7 is a block diagram of the E Box.

The M Box

The M Box, the memory subsystem, consists of memory control logic, memory arrays, and a dedicated memory array bus that provides a usable data rate of over 50MB per second to the memory subsystem. The control logic optimizes multiple memory read and write operations, implements three-way interleaving, and buffers memory transactions for optimum data movement. The dedicated memory array bus, coupled with the memory control logic, effectively off-loads the NMI bus, providing balanced bus access and loads. The interleaving algorithms are based upon array boundaries, making the memory control logic technology independent. The result is that as increasingly dense memory arrays become available, few if any controller modifications will be required.

The error checking and control (ECC) is built around 7 check bits for every 32 bits of data. This protocol provides automatic single-bit correction and double-bit detection.

In the VAX 8800 multiprocessor, all memory is fully sharable. Current systems in the VAX 8800 family are offered with 16MB per memory array, giving the VAX 8700 and VAX 8800 systems a maximum memory capacity of 128MB, and the VAX 8500 and VAX 8550 systems a maximum of 80MB. Figure 8 is a block diagram of the M Box.

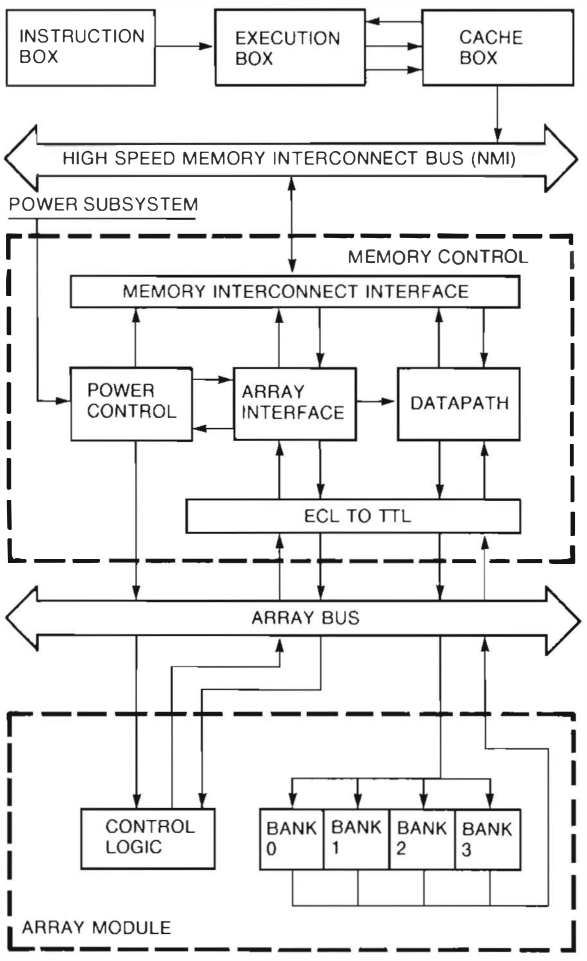


Figure 8 M Box Block Diagram

The Clock Subsystem

The clock subsystem generates, controls, and distributes timing signals to all the components of the processor system. The clock subsystem contains the console interface, an oscillator, a phase generator, clock-control logic circuits, and the logic circuits for clock signal distribution.

The VAX 8800 family implements a two-phase, nonoverlapped clock subsystem operating at a cycle time of 45 ns. A stable, high-frequency oscillator (120 MHz nominal with variable output), coupled with a phase generator, provides the signal. The implementation of a two-phase design with matched signal-length distribution throughout the CPU is most efficient for the pipelined, latch-based design of the VAX 8800 family. This design avoids the inefficiencies associated with the compressed signal-assertion times resulting from approaches that specify minimum delays for given logic elements.

A-clock and B-clock signals are distributed to alternate latches in a given logic stream. All data transfers occur between latches clocked by different phases to assure a race-free design. The essence of fast-processor design is managing and controlling skew. In this regard, signal propagation and distribution presented significant challenges in the areas of controlled etch lengths, controlled impedance, routing, and placement. To assure a stable, reliable design, all design activity was predicated on worst-case design rules rather than using the typical-case limits.

The NMI Bus

Integral to the design of this family of processors was the development of a high-speed memory interconnect bus called the NMI bus. This bus, analogous to the synchronous backplane interconnect (SBI bus) in the VAX-11/780 CPU, links the subsystems for CPU logic, central memory, and I/O. The NMI bus is a 32-bit synchronous bus, physically implemented within the 22-layer backplane. This bus provides the control and datapath functions as well as the distribution of clock signals for the VAX 8800 family.

One fundamental problem in the design of high-performance systems revolves around balancing the bus access needed at any given instant with the raw bandwidth available. To provide the correct balance, the NMI bus was implemented as a pended (vs. interlocked) bus, resulting in very high bus-access availability.

Since memory is the critical resource in sustained operations, the NMI bus uses a modified round-robin arbitration that gives the memory a higher priority when there is contention for the bus. This arbitration priority eliminates any lock-step conditions and also provides for recovery of states and data in the event of preemption. This high bus-access capability, coupled with usable data rates of up to 60MB per second, provides the necessary balance to support CPU, memory, and I/O transactions. The inclusion of write buffers within each CPU, coupled with the large cache size, effectively reduces the number of transactions presented to the bus. Measurements on a VAX 8800 system in our Engineering VAXcluster environment have indicated that the NMI bus is rarely busy more than 50 percent of the time; the CPUs use approximately 25 percent of the available access time and bandwidth. Other applications may see somewhat different ratios.

VAXBI Bus

The VAX 8800 family uses the VAX bus interconnect, called the VAXBI bus, for the I/O subsystem in order to provide adequate balance for the CPU performance. The VAXBI bus, a 32-bit clocked bus with distributed arbitration, is capable of usable data rates in the VAX 8800 family up to 8MB per second, depending upon word size and application. Custom logic on each interface module provides all bus protocols, as well as integral data-integrity features, including master transmit and command acknowledge.

The VAX 8800 and VAX 8700 systems can be configured with up to four VAXBI channels, whereas the VAX 8550 and VAX 8500 systems accept up to two. Therefore, fully configured VAX 8800 and VAX 8700 systems can support aggregate I/O bandwidths up to 30MB per second. Similarly, fully configured VAX 8550 and VAX 8500 systems can support aggregate bandwidths up to 16MB per second. Each VAXBI bus can support up to 16 nodes, or logical addresses, which connect to any combination of networks, intelligent and nonintelligent devices, DMA devices, and VAXcluster systems, as well as providing for connection to existing UNIBUS-based devices.

All of Digital's network protocols interface directly to the VAXBI on the VAX 8800 family. Thus, VAXcluster, Ethernet, DECnet and DSA (Digital Storage Architecture) devices are all ported directly to this high-performance I/O subsystem.

Reliability

Reliability was one of the primary goals of the VAX 8800 design. Numerous features were implemented that more than doubled the basic computing kernel availability compared to the VAX-11/780 system. Some of the key functions include

- Environmental and power monitors that query the system and maintain safe system operating levels
- Automatic verification of hardware, firmware, and software revision compatibility
- Electrically keyed modules and module slots that prevent improper installation and damage to the modules or the system
- Automatic electrostatic discharge (ESD) protection of modules during installation and removal
- ECC on main memory
- Parity checking on internal RAMs
- Bus protocol checking for the memory interconnect
- Timing and voltage margining
- Remote diagnostics capability
- Dual-to-single processor reconfiguration (VAX 8800 system only)

Diagnostic Development

Similar to the hardware development, the design methodology for the diagnostics depended very heavily on simulation. Almost all the diagnostic tests were debugged on behavioral and structural models of the design before the initial prototype was powered up. There were three major benefits of this methodology.

1. Microdiagnostic and macrodiagnostic tests were useful for design verification testing.
2. Test vectors for automatic test equipment (module test) were extracted from the simulation database.
3. A comprehensive diagnostic package was available shortly after the prototype was powered up.

The diagnostic for the VAX 8800 family consists of tests specific to this processor and generic to the VAX architecture. The processor is tested primarily with microdiagnostics. These tests execute from the processor's writable control store and are governed by the console.

VAX generic diagnostics are included to test the UNIBUS and VAXBI adapters and options. All the diagnostic code fits on the console's Winchester disk. When the system is powered up, a subset of the microdiagnostic tests are executed.

Balanced Systems

The VAX 8800 design effort delivered four different systems, the 8800, the 8700, the 8550, and the 8500, all reflecting the overriding concept of balanced system design. While the CPUs themselves demonstrate excellent internal balance between their logical and functional subsystems, they are also balanced members of the extended system that can span much larger physical distances. Monolithic or isolated computing resources are no longer capable of accessing, manipulating, and distributing the volumes of information needed for complex or extended solutions. In this light, the VAX 8800 family should be viewed in the context of a balanced network. The movement of data is governed by speed and distance. An inverse relationship exists as shown in Figure 9. The VAX 8800 family fits on the top bound of the bandwidth range throughout the distance function.

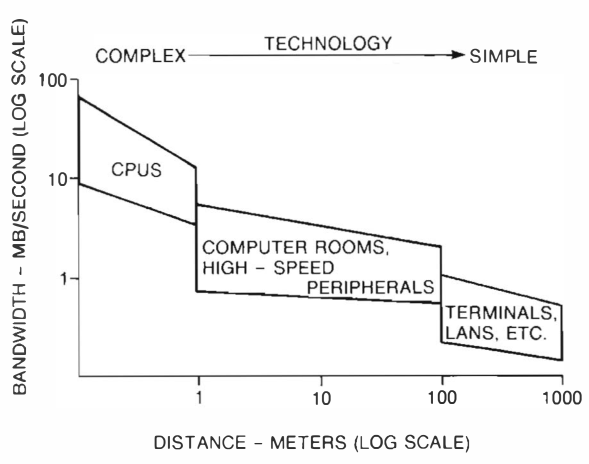


Figure 9 Bandwidth versus Distance

Summary

The VAX 8800 family of products merges fast instruction-execution rates, large physical memories, large high-speed data caches, VAXBI I/O channels, pipelining, and balanced internal-bus architectures to provide high system-applications throughput. Spanning an applications throughput range that is from 3 to 12 times that of the VAX-11/780 system, the VAX 8500, VAX 8550, VAX 8700, and VAX 8800 systems are matched to the network and applications strategies offered by Digital Equipment Corporation.

References

1. D. Bak, "The Impact of VAX 8800 Design Methodology on CAD Development," *Digital Technical Journal* (February 1987, this issue): 129–135.

2. *VAX Hardware Handbook* (Maynard: Digital Equipment Corporation, Order No. EB-21710-20, 1982).

Sudhindra N. Mishra

The VAX 8800 Microarchitecture

The VAX 8800 processor has a simple but efficient microarchitecture. Its pipelined micromachine has a one-cycle next-address loop and four-cycle latencies for both microbranches and microtraps. Instruction prefetch and decode are done in parallel with microcode execution. The instruction buffer is a bit-sliced, four-longword circular queue. The decoder is primarily a RAM-based table. For special events, hardwired logic is used for decoding. A bit-sliced microsequencer provides up to 32-way conditional microbranching, using a collection of about 80 branch conditions. A hardware microstack provides up to 15 levels of nested subroutine calls and returns. Microtrap conditions are prioritized over 16 levels, and microtraps are chained, not nested.

The term "microarchitecture" means the specification or description of the interrelationships between the parts of the micromachine that implements the instruction set processor. In terms of this definition, the microarchitecture of the VAX 8800 processor will be described by elucidating the organization of its micromachine and the interaction between its components.

Figure 1 shows a simple three-stage state-machine model of an abstract micromachine appropriate for implementing the control unit of a typical von Neumann processor. Figure 2 shows a block diagram depicting the essential elements of such a micromachine. This state-machine is capable of executing microcode routines to implement an instruction set processor. In such a system, every macroinstruction is decoded by the hardware to produce the starting addresses of a small set of microprograms, which execute sequentially to produce the desired effect. Barring some exceptions, a microprogram or microcode routine can execute rather independently in the sense that each microinstruction produces the address of the next microinstruction. The last microinstruction causes the selection of an external address, such as one produced by the decoder, to start the execution of another routine.

In Digital's vernacular, the I Box is the logical partition containing the instruction-processing hardware. Figure 3 shows a block diagram of the VAX 8800 I Box with the basic elements of its micromachine.

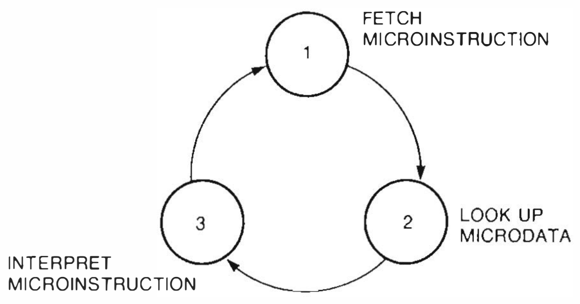


Figure 1 State-machine Model of an Abstract Micromachine

From the early IBM and CDC computers to the modern CRAY machines, computer designers have used a technique called "pipelining" to obtain higher performance. Pipelining overlaps the execution of instructions in time; that is, several instructions can be executing at the same time. This technique provides a higher throughput when the pipeline is fully loaded, but there is a cost involved. If the pipeline is broken, extra processing is required to refill it. Moreover, if any active instructions have partially executed, information about their states may have to be saved to continue processing after an abrupt interruption.

The degree of pipelining varies from one machine to another depending upon the design choices and trade-offs made by the system architects. A metaphor often used to indicate the degree of pipelining is the length of the pipeline

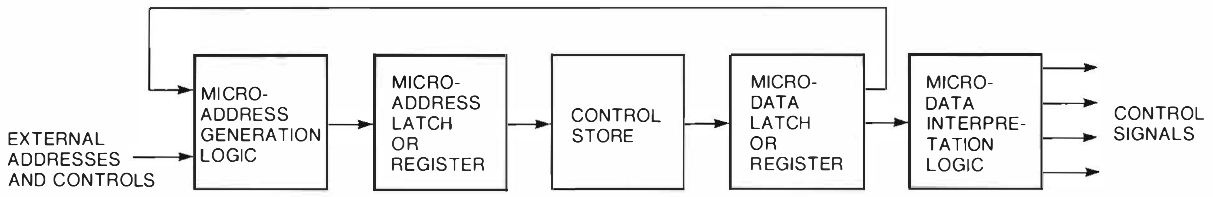

EXTERNAL ADDRESSES AND CONTROLS
MICRO-ADDRESS GENERATION LOGIC
MICRO-ADDRESS LATCH OR REGISTER
CONTROL STORE
MICRO-DATA LATCH OR REGISTER
MICRO-DATA INTERPRE-TATION LOGIC
CONTROL SIGNALS

Figure 2 Block Diagram of an Abstract Micromachine

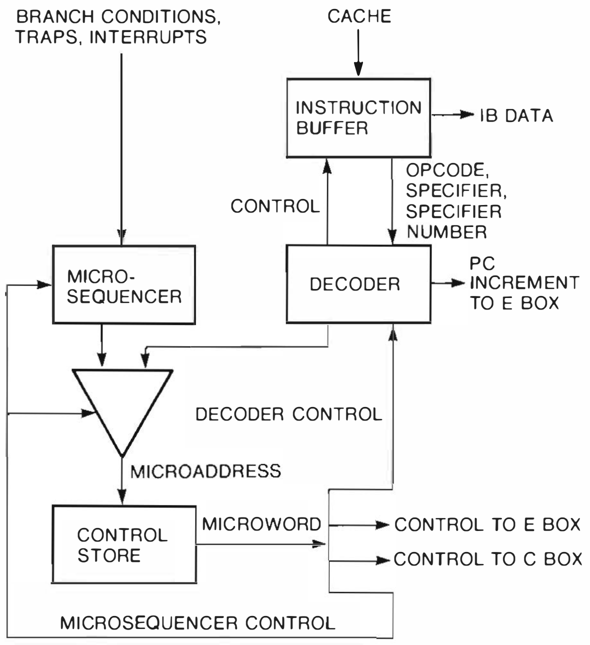


Figure 3 VAX 8800 I Box

stated as the number of stages, for example, a three-stage pipeline or a four-stage pipeline. The number of stages conveys the extent of time overlap for typical operations in a computer. In a machine with a pipelined microarchitecture, these operations are executions of microinstructions. A higher degree of pipelining makes short cycle times possible, thus leading to a higher throughput when the pipeline is fully loaded. But longer pipelines entail increased overhead in terms of their ability to resume operations after a break in the pipeline caused by any abnormal event. Therefore, an architect's goal is to design the system so that the pipeline remains loaded most of the time and recovery from a broken pipeline is not too inefficient. The VAX 8800 CPU is a prime example of a processor with a pipelined microarchitecture.

System Considerations

The design philosophy of the VAX 8800 processor was to optimize the hardware so that it would execute the microcode efficiently. A large control store (144 bits by 16,000 entries) holds the entire microcode. Using fairly generalized datapaths, the microcode executes the logic of the instructions. However, special hardware is used to speed up performance in critical areas. The processor logic is primarily designed with latches, which are clocked with a globally distributed, two-phase, nonoverlapping clocking scheme. The two clock phases are called the A-clock and the B-clock. A typical example of logic design, based on the above approach, is shown in Figure 4.

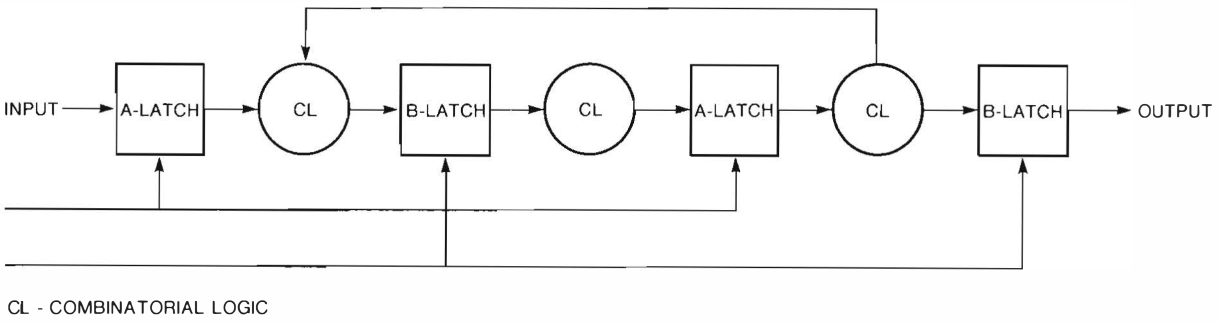


Figure 4 A Typical Section of the VAX 8800

It is apparent from Figure 4 that the data flow in such a logic system occurs through the perpetual data transfers between the latches connected to the A-clock and those connected to the B-clock. Each data transfer may be considered atomic in the sense of hardware operation. A microoperation may be envisioned as a logical operation that is atomic in terms of the execution of a microinstruction, such as a register read, a register write or an ALU function. Hence a microoperation constitutes one or more data transfers, and the microinstruction execution simply constitutes a time sequence of microoperations, as shown in Figure 5.

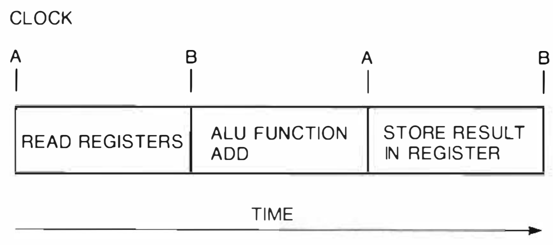


Figure 5 Example of a Microinstruction

In high-performance machines, like those in the VAX family, there is usually a mismatch between CPU cycle times and memory-access times. For example, consider an ADD instruction. If the operands are in registers, the ADD can be done rather quickly. But if one of the operands has to be read out of memory, the ADD cannot be performed until the desired data arrives from memory. Most VAX processors have a fast cache memory, tightly bound to the processor's arithmetic units, to alleviate the memory-latency problem. In the case of a cache miss on a required datum, however, the only alternative for a von Neumann processor is to wait. A processor in such a state is said to be "stalled." Under such conditions, the state of the processor must be "frozen" until the cause of the stall no longer persists and the stall is broken. The two-phase clocking scheme provides a convenient way to implement stalls, in which one of the clock phases (the A-clock in the 8800) may be blocked. Stalls are controlled by the cache through a special hardware signal distributed globally to block the A-clock. Thus, the processor logic contains two flavors of A-latches:

- Stalled A-latches, which are affected by a stall
- Unstalled A-latches, which are not affected by a stall

The micromachine is implemented only with stalled A-latches. Hence the effect of stalls on the execution of the micromachine is largely transparent.

A mechanism is also required to deal with hardware exceptions when the results of the execution of a microinstruction have to be undone. In a pipelined microarchitecture, several microinstructions may have partially executed when an exception condition is detected. In that case it is necessary to undo the effects of all those microinstructions. The most common technique used to deal with such situations is called a microtrap. Since microtraps relate closely to the micromachine execution, every processor has its own scheme to implement them. In every case, however, microtraps must permit the "roll back" of some number of microinstructions because the detection of a trap condition usually occurs quite late with respect to microinstruction execution.

In the VAX 8800 processor, microtraps are implemented so that the offending microinstruction is allowed to complete, but subsequent microinstructions in the pipeline are blocked. Since the offending microinstruction may have caused some undesirable results, the trap-handler microcode must fix the problem. Depending on the particular situation, either the microinstruction execution flow is resumed from the blocked state or a new flow is originated.

System Buses and Datapath

Figure 6 is a block diagram of the VAX 8800 CPU datapath, showing all the major buses. The hardware organization of the CPU provides a two-cycle operation between the cache and the ALU, as shown. The processor has several functional units in addition to the main ALU. These additional units perform high-speed multiply and divide, shifting, and floating-point arithmetic operations.

There are several possibilities for selecting inputs to these functional units. For operations involving two inputs, both can be presented simultaneously onto the two legs of the main ALU as well as most other functional units. The results from these functional units are sent on the W bus for writing to either the multiport

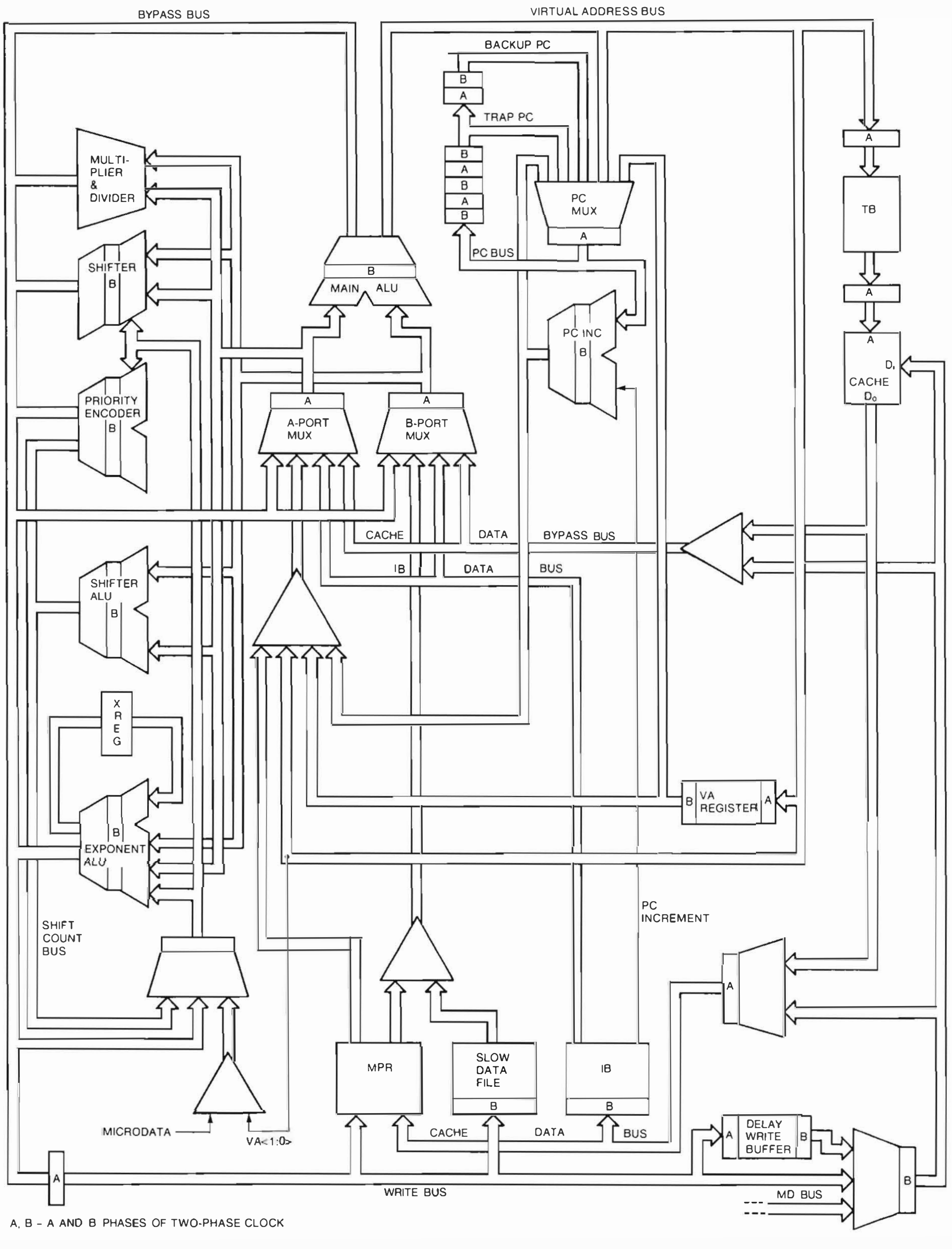


Figure 6   VAX 8800 Datapath

register file (MPR) or the cache. However, since the write actually occurs in the following cycle, the bypass bus provides a shortcut (saving a cycle) in case the write datum is read by the very next microinstruction.

The virtual address bus carries the virtual address of any data-stream (d-stream) references, whereas the program-counter bus has the current program counter (PC). The instruction-buffer data bus provides the instruction-stream (i-stream) data. The instructions and data from the cache are returned on the cache data bus. However, a cache data bypass bus provides a direct path to the functional units for the data returned by the cache, in case the processor is or will be stalled for that data.

Microinstruction Pipeline

The top part of Figure 7 shows the execution of microinstructions as a function of time in a nonpipelined microarchitecture; the bottom depicts that in a pipelined microarchitecture.

The basic data flow in a processor occurs in the following sequence:

1. Read the register operands into a functional unit, such as the ALU.
2. Perform some ALU function.
3. Write the results into the destination register.
4. If there is a cache, start a cache operation at approximately the same time as a register write since memory references are buffered through special-purpose memory data registers (MDRs or MDs) in most high-performance processors.

Figure 5 shows that the sequence above occurs in a natural order in time as a consequence of the microinstruction execution. With pipelined microarchitectures, a time reference is needed to correlate the microoperations performed by various microinstructions with respect to each other. The notion of canonical times is very convenient for this purpose. The clock ticks of the reference microinstruction may be labeled with a monotonically increasing set of T numbers starting at T_0 as shown in Figure 8. These T numbers are called the canonical times of a particular microinstruction. The microoperation labeled T_0 marks the start of a microinstruction execution cycle. Figure 8 shows the basic microoperations of a VAX 8800 microinstruction with their canonical times.

We shall use the simple model of a micromachine in Figure 1 to describe the VAX 8800 micro-

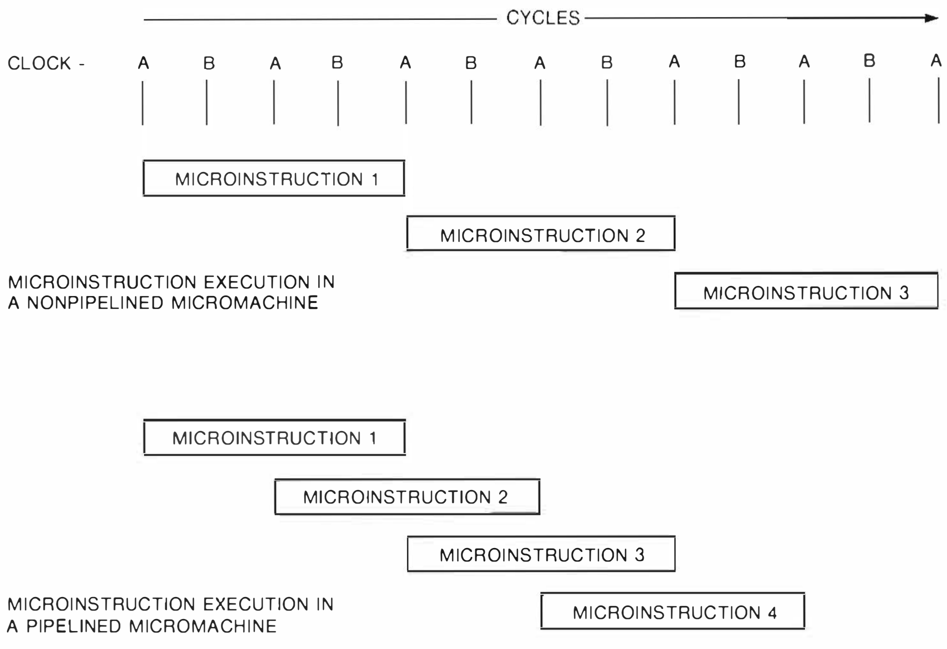

CYCLES
CLOCK -
A
B
MICROINSTRUCTION 1
MICROINSTRUCTION 2
MICROINSTRUCTION 3
MICROINSTRUCTION 4
MICROINSTRUCTION EXECUTION IN A NONPIPELINED MICROMACHINE
MICROINSTRUCTION EXECUTION IN A PIPELINED MICROMACHINE

Figure 7 Microinstruction Execution

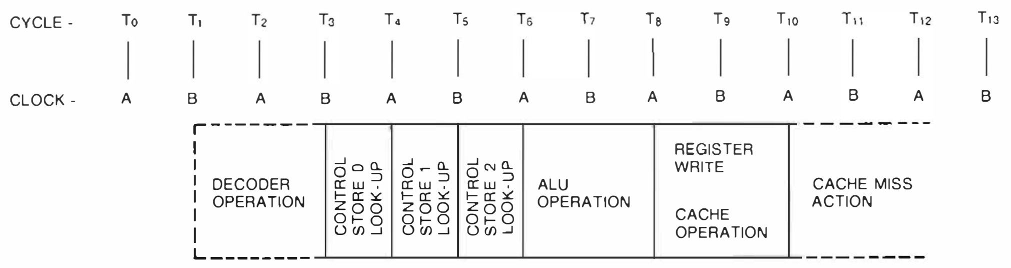


Figure 8 Canonical Times of a VAX 8800 Microinstruction

instruction format as a sequence of basic microoperations like those in Figure 8. The first stage in the microinstruction execution cycle is the microaddress fetch. The microinstruction execution cycle begins with a decoder operation. The decoder produces the starting microaddress for every new microinstruction sequence and presents it to the microsequencer. The decoder determines that address on the basis of the contents and current state of the instruction buffer (IB). Each microinstruction specifies to the microsequencer whether or not to accept the decoder's microaddress. If not, the microinstruction must either specify the address of the next microinstruction directly, as a part of the microword, or indicate an alternate source for the address within the microsequencer. Since the decoder's operation is concurrent with the microsequencer's, the decoder always has a starting microaddress for the microsequencer. It is convenient to think of this IB-decoder concurrency as a "hidden decoder cycle."

The next stage in the microinstruction execution sequence is the fetch of the microinstruction, performed by a look-up in the control store. In the VAX 8800 system, the microaddress is pipelined, not the microdata. Consequently, the microdata from a segmented control store appears at the appropriate time for the three basic operations to occur in the indicated order.

The microdata looked up causes a sequence in which the register read occurs between the times T_5 and T_6, the ALU function between T_6 and T_8, and the register write between T_8 and T_{10}. The cache operations also occur between the times T_8 and T_{10}. The section beyond T_{10} denotes cache activity with respect to the memory if there is a cache miss. (The cache/memory interface is controlled by an independent micromachine.) During every cycle, a microinstruction produces the address of the next microinstruction, which is then executed. Figure 9 depicts the generic microinstruction pipeline of the VAX 8800 processor.

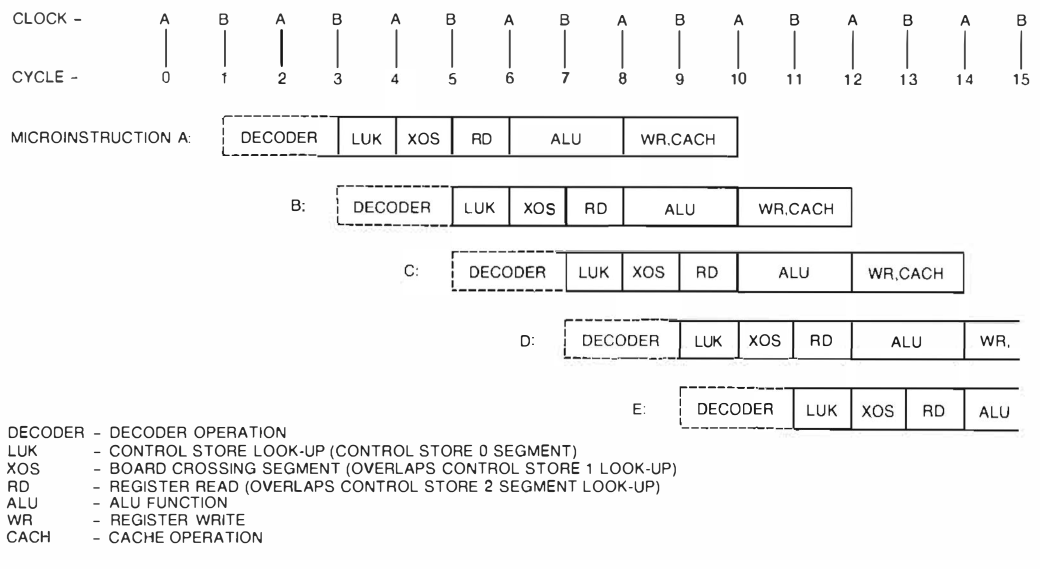

CLOCK -
A B A B A B A B A B A B A B A B
CYCLE -
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
MICROINSTRUCTION A: DECODER LUK XOS RD ALU WR,CACH
B: DECODER LUK XOS RD ALU WR,CACH
C: DECODER LUK XOS RD ALU WR,CACH
D: DECODER LUK XOS RD ALU WR,
E: DECODER LUK XOS RD ALU
DECODER - DECODER OPERATION
LUK - CONTROL STORE LOOK-UP (CONTROL STORE 0 SEGMENT)
XOS - BOARD CROSSING SEGMENT (OVERLAPS CONTROL STORE 1 LOOK-UP)
RD - REGISTER READ (OVERLAPS CONTROL STORE 2 SEGMENT LOOK-UP)
ALU - ALU FUNCTION
WR - REGISTER WRITE
CACH - CACHE OPERATION

Figure 9 Microinstruction Pipeline of the VAX 8800 CPU

Microbranch Latency

One consequence of pipelining is that any intervening microinstructions must be spaced between the instruction that produces a branch condition and the instruction that can branch on it due to latency in the development of the branch condition. Obviously, the execution of the intervening microinstructions must be independent of the branch. Usually, microcoders are able to code some useful operations during the inevitable wait. Otherwise, the intervening instructions must be NOPs (no operation). Figure 10 shows the microbranch latency in the VAX 8800 CPU.

Microtrap Latency

A hardware exception causes a microtrap. However, the trap conditions, like the branch conditions, may develop after some execution cycles have been completed. Once again there must be some intervening microinstructions between the trap-causing microinstruction and the trap-handling routine. Moreover, the state of the micromachine must be saved so that the current execution can be resumed in such a way that the intervening execution of the trap routine appears to be transparent. This state consists primarily of microbranch conditions that result from the execution of microinstructions in the pipeline since those could influence subsequent microaddresses and hence the execution sequence. Therefore, on interruption of the current sequence by the trap routine, the branch conditions from the earlier execution are essential to reproduce the same sequence.

To simplify the hardware design, all early traps are delayed to a fixed canonical time (T_{10}). Some trap conditions, however, develop later than the canonical time with the consequence that those traps cannot be returned from. In such cases the microcode must roll back the state to the beginning, which causes a reexecution of the entire macroinstruction.

Figure 11 shows a sequence in which a microinstruction at address T provokes a microtrap. At the earliest, the trap-handling routine can begin at microinstruction X. Meanwhile, microinstructions U, V, and W follow T, quite unaware of the impending trap. In fact, they are in partial execution when the trap condition is detected. These microinstructions are said to be in the trap shadow, and they must be blocked from writing any registers, thus making it appear as if they had never executed. When control is returned from the trap-handling routine, these trap shadow microinstructions are reexecuted, continuing the sequence that would have arisen had the trap not occurred.

Instruction Buffer and Decoder

The IB buffers the prefetched VAX i-stream delivered by the cache and in turn delivers the opcode and specifier to the decoder. The IB also delivers the i-stream data to the execution unit, the E Box. The decoder expects to receive the current opcode and the current specifier byte.

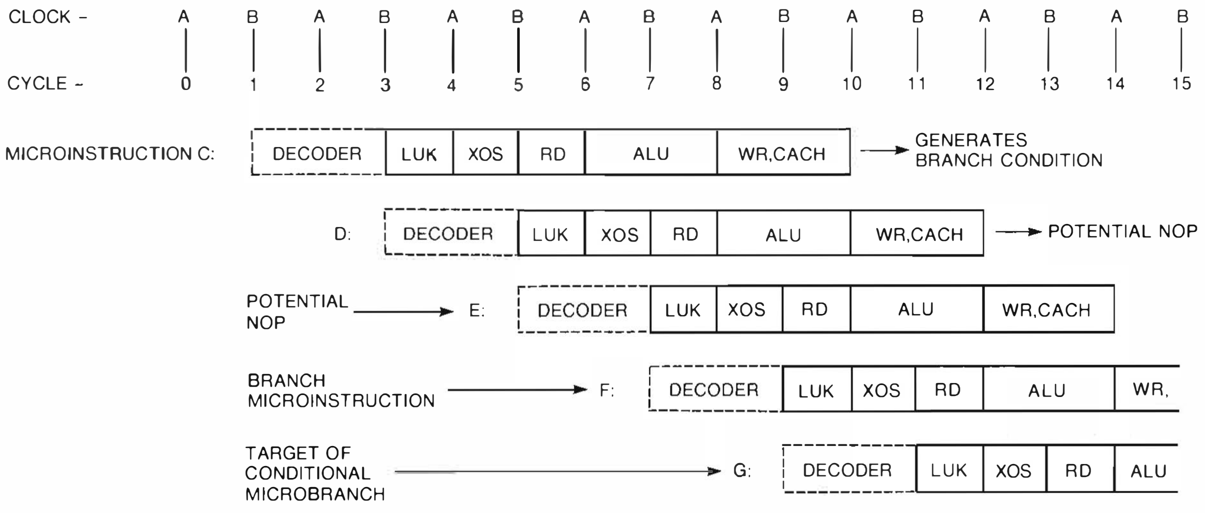


Figure 10 Microbranch Latency

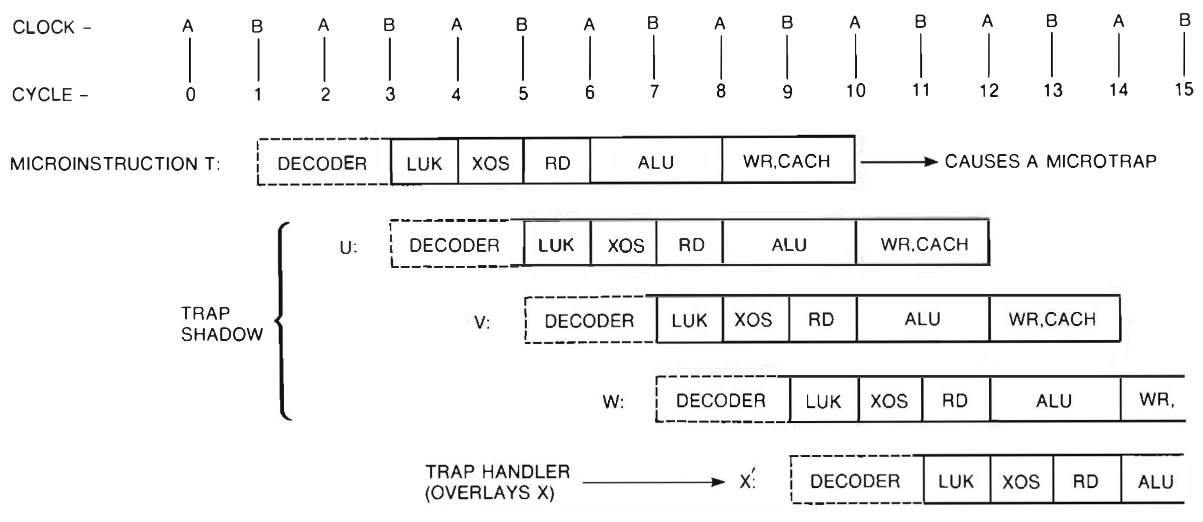


Figure 11 Microtrap Latency

Hence the IB saves the opcode for the duration of the instruction execution and shifts the buffered i-stream along to send each specifier in turn to the decoder. The goal of the VAX 8800 decoder is to produce a starting microaddress corresponding to the opcode and the specifiers. The sequence of microcode execution caused by the decoder is first to process all the specifiers, making all the operands available, and then to execute the operation specified by the opcode. If an instruction has no specifiers, the execution microcode is initiated directly. In any case the decoder always has a microaddress ahead of time for the microsequencer. This microaddress is the starting address of either a specifier routine or the execution routine, based on the contents and the state of the IB.

If at any time the IB does not contain enough i-stream data for a successful decode, the decoder will produce a special microaddress. The microinstruction at that address is simply a NOP that again requests the selection of the decoder's address. The micromachine thus waits in a loop for sufficient i-stream data to arrive in the IB so that the decoder can again dispatch a useful microaddress. This wait-loop state of the micromachine is commonly referred to as the IB stall, which is different from the stall described earlier. Note that clocks to stalled A-latches are not blocked for an IB stall. On the contrary, the micromachine runs normally as does the rest of the processor hardware. IB stalls may occur when the instruction prefetch pipeline is broken due to macroinstruction branches. This condition requires the current contents of the IB to be discarded and new i-stream data to be prefetched into the IB.

The VAX 8800 IB is a four-longword circular queue, which is usually long enough to hold an entire instruction. The data is consumed out of the IB from the position pointed to by the read pointer. However, new data could be written concurrently by the cache at the position pointed to by the write pointer. Whenever it has room, the IB is loaded by the cache if the cache has no other higher priority job to do. Occasionally, the IB becomes full (the write pointer catches up with the read pointer), and then it does not accept the datum from the cache. If a datum is not accepted by the IB, the cache keeps repeating the transfer until the datum is accepted. Occasionally, the IB becomes empty if the cache is busy doing other things and the decoder has consumed all the data from the IB (the read pointer and the write pointer point to the same location).

The IB in the VAX 8800 family is implemented with four identical gate arrays with 8-bit slices designed to use a rather clever bit-scattering/gathering scheme. The IB also contains logic to extract and format i-stream data, making it available to the E Box. A common silo holds the opcode history for the duration of a macroinstruction's execution, as well as for recovery from microtraps. The VAX 8800 decoder is a RAM-based look-up table for generating

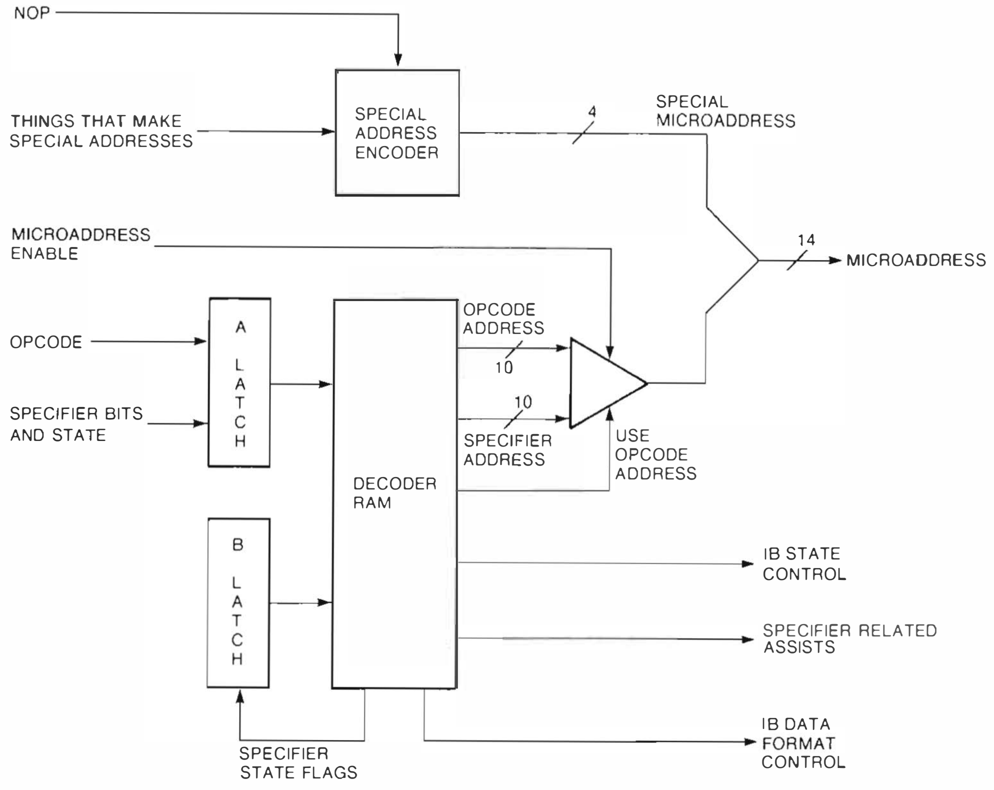


Figure 12 VAX 8800 Decoder

microaddresses. In the case of special events, however, hardware logic is provided for generating special microaddresses, as shown in Figure 12, thus bypassing the RAM look-up. The decoder also provides controls for the IB state-machine as well as some other hardware assists.

Microsequencer

The state-machine responsible for generating the next microaddress for a microinstruction sequence is commonly called the microsequencer. As shown in Figure 13, this state-machine is realized collectively by the control store, the next

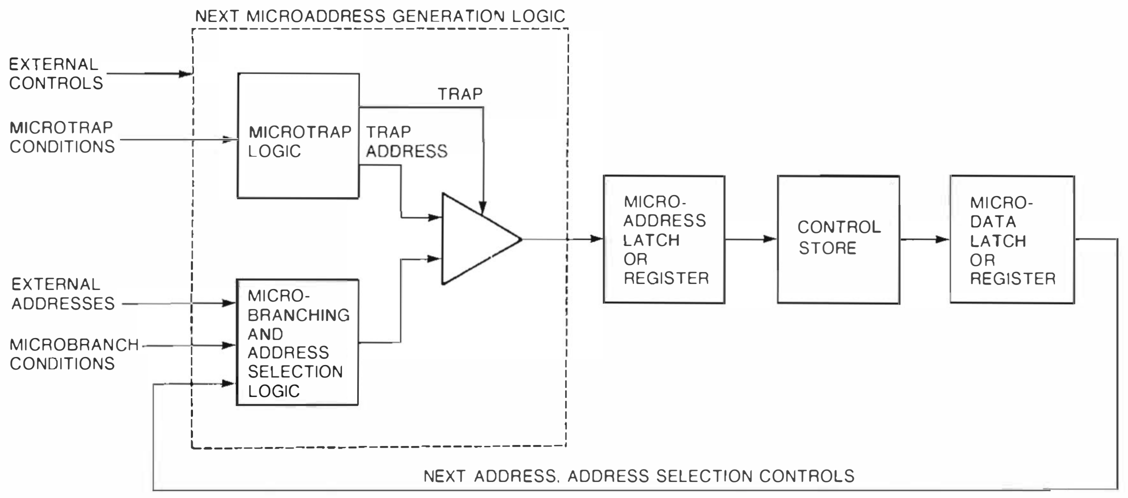

NEXT MICROADDRESS GENERATION LOGIC
EXTERNAL CONTROLS
MICROTRAP CONDITIONS
MICROTRAP LOGIC
TRAP
TRAP ADDRESS
EXTERNAL ADDRESSES
MICROBRANCH CONDITIONS
MICRO-BRANCHING AND ADDRESS SELECTION LOGIC
MICRO-ADDRESS LATCH OR REGISTER
CONTROL STORE
MICRO-DATA LATCH OR REGISTER
NEXT ADDRESS, ADDRESS SELECTION CONTROLS

Figure 13 An Abstract Microsequencer

microaddress generation logic, and the microaddress and microdata latches (or registers).

The goal of the VAX 8800 microsequencer is to produce the address of the next microinstruction during every cycle. Figure 14 depicts how the microsequencer achieves this goal.

Each microinstruction may modify its next-microaddress field through a microbranch command to produce the address of the target microinstruction. Microbranch conditions are delivered by other sections of the machine, such as the ALU. These conditions are grouped together in ways convenient for microprogramming so that multiway branches can be taken. Microsubroutines can be called and returned from by means of a hardware microPC stack.

Stalls cause the microsequencer state to be frozen on a cycle boundary (i.e., the clocks on microaddress and microdata latches are effectively blocked). Microtraps allow the microcode to deal with unusual events that would be too slow or inconvenient to check normally with microbranches, such as TB misses and address misalignments. The VAX 8800 processor does not permit traps to be nested. Instead, traps are "chained," meaning that trap routines and hardware trap priorities are carefully arranged so that a second trap is taken only when the first trap routine finishes. (Machine check traps cannot be controlled in this way.)

Sources of Microaddresses

There are five sources for microaddresses:

- The decoder
- The next-address field in the microword
- The microstack upon returning from a subroutine
- The microPC silo for a saved microtrap
- The micromatch register for an address from the console

An address from the console is selected in response to an explicit console request and takes precedence over everything else. Addresses from the silo are requeued in response to a trap-return command. Addresses from the microstack are selected in response to a subroutine-return command. A decoder-generated address is selected whenever the current sequence ends and a new specifier or execution routine should begin. Normally, this selection is caused by the assertion of a microword bit in the very last microinstruction of the current sequence. The next-address field is selected as the default for normal sequencing. This field is also used to provide an offset in case of subroutine returns.

Microbranching

In normal cases, part of the selected microaddress can be modified according to the branch conditions, that is, whenever the next-address field is selected. A combination of two microword fields, branch type and branch mask, selects the branch conditions, which are then ORed into part of the target microaddress. In the VAX 8800 system, the microbranch logic is implemented with five identical gate arrays, each of which generates a 3-bit slice of the microaddress. One microaddress bit is branch sensitive in each slice. This organization permits up to 32-way branching. Branchings of 2, 4, 8, and 16 ways are also made possible by a separate mask bit, called the branch mask, to every slice. This bit is used to turn off the sensitivity to branch conditions in a particular slice.

There are 16 basic recipes for conditional branching in each slice. This arrangement of slicing, masking, and branch-condition selection in every slice requires that all the microbranch conditions be organized into 5 groups of 16 conditions each. The branch conditions are classified as either static or dynamic. Static conditions, once captured, are available for branching in any later cycle as long as those conditions remain unchanged. Dynamic conditions are asserted for just one cycle and must be branched on in that cycle.

Some special trap-related branch conditions are saved at the time of the trap so that the trap routine may use them. For speed reasons, the basic hardware mechanism for multiway branching is that the selected condition is ORed rather than added to the branch-sensitive microaddress bit. The OR implies that the branch-sensitive bits of a microaddress must be "zeros" by convention. If branching is masked in any slice, however, only unmasked branch-sensitive bits need to be zeros. Thus the branch-masking scheme leads to a substantial increase in the number of conditional branch-target addresses, constrained by the requirement for zeros.

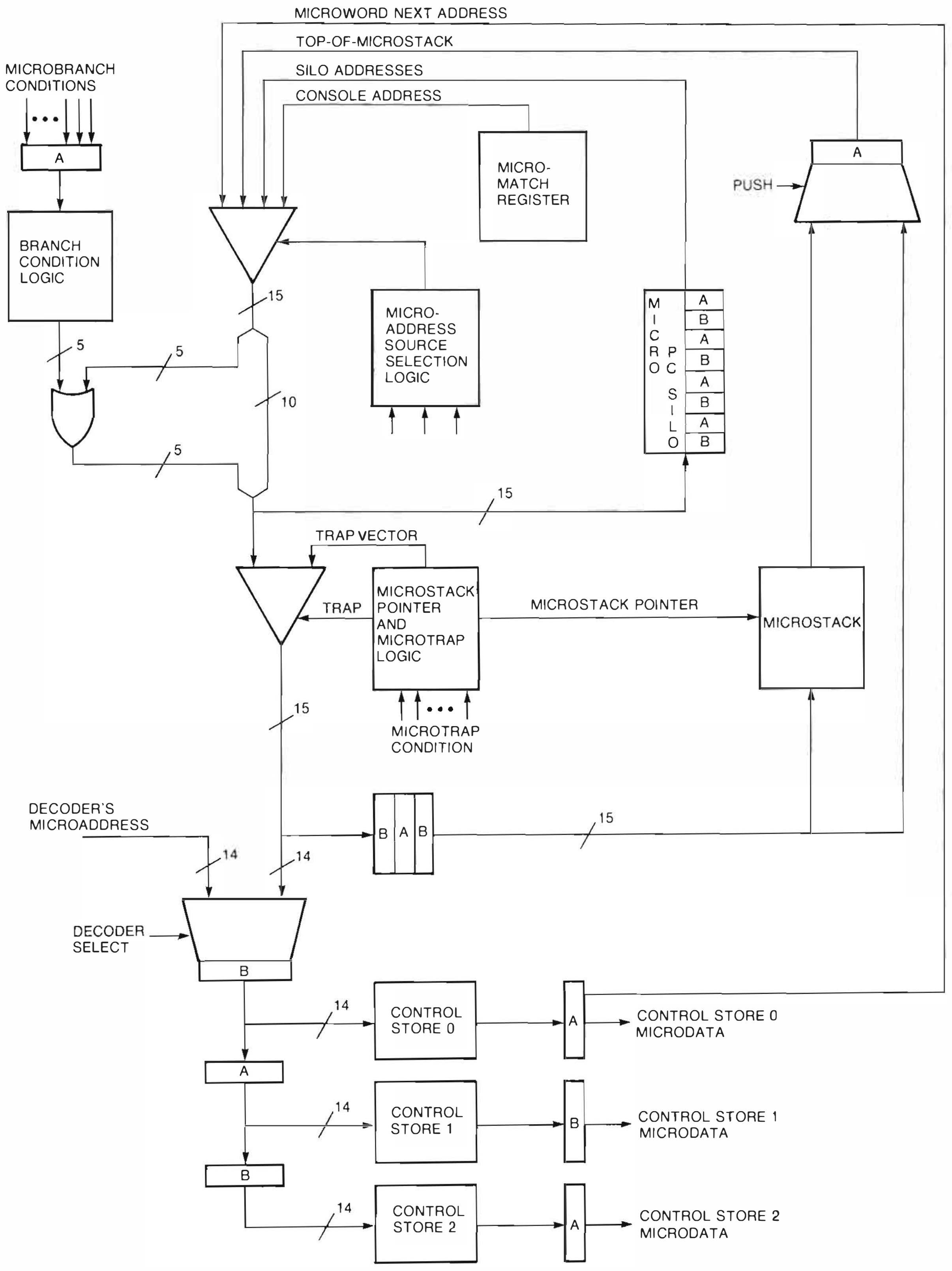


Figure 14 VAX 8800 Microsequencer

Table 1 Microbranch Conditions

Slice Number	Microbranch Conditions
1	State flags
2	WBUS low-order bits
3	WBUS high-order bits
4	SALU condition codes
5	PSL condition codes
6	XALU condition codes
7	Priority encoder condition codes
8	ALU condition codes
9	TB-status
10	Cache command
11	MD number
12	AC low
13	Digit valid
14	NMI ID
15	Interrupt pending
16	Interval timer carry
17	Halt pending
18	Console mode
19	Interrupt ID
20	Non_Retry flag

Table 1 shows an example of several microbranch conditions.

Microsubroutine Call and Return

As in the normal case just discussed, the default microaddress, the next-address field, is selected as the starting address of a microsubroutine. However, a subroutine-calling microinstruction pushes its own address onto the microstack. During the subroutine return, the microstack is selected as the source and then popped. Thus the address of the calling instruction is used as a base for the return. The returning instruction may OR an offset from the next-address field to that base, thus yielding the target return address. The fact that bits are ORed rather than added constrains the calling addresses to have zeros in the low-order bit positions.

The write path to the microstack (PUSH) is pipelined by a cycle for timing reasons. However, a bypass path saves what would be the top entry of the microstack in the read latch (POP) so that PUSHs and POPs occur in a fairly unrestricted manner. There are, however, some minor coding restrictions with respect to traps and decoder-made addresses.

Subroutine calls and returns are unaffected by stalls. In the VAX 8800 CPU, the microstack is 16 entries deep and is used exclusively for subroutine calls and returns (i.e., microtraps do not use the stack). Subroutine calls may be nested up to 15 entries deep, beyond which the microstack wraps around and overwrites previous call addresses. Since the next-address field is conditionally ORed into the calling address to make the return address, a conditional multiway return becomes feasible.

Microtrap and Return

A microtrap is caused when the hardware detects a condition that would not allow the current microinstruction to complete its execution successfully. The hardware forces the next microaddress to a fixed location that depends on the particular condition, thus overriding the address that would otherwise be selected. This special location is the starting address of the trap-handling microcode routine specific to that trap condition. Microtraps are used extensively by the memory management system to implement the virtual memory architecture. Microtraps are also caused by serious system faults (i.e., machine checks), such as control-store or bus parity errors. Table 2 lists the microtrap conditions and their priorities. The priorities are arranged so that if more than one microtrap occurs during a cycle, the one with the highest priority will be serviced and the others ignored.

Table 2 Microtrap Conditions and Priorities

Microtrap Condition	Priority
Microbreak	Highest
Machine check	↑
VA parity error	
TB tag parity error	
Reserved for ECO	
Reserved float operand	
Add rounding	
Multiply rounding	
Integer overflow	
TB miss	
Access violation	
Modify bit	
Page cross	
Unaligned page cross	
Unaligned trap	
Conditional VAX branch	Lowest

Figure 11 shows the microtrap latency and its consequences on pipelining. As described earlier, a trap-causing microinstruction, even if it writes the wrong results, is allowed to complete because it is too late to block it anyway. (The canonical time of register write is T_9, whereas the microtrap signal occurs at canonical time T_{10}). The only recourse is to let the trap-handling microcode correct any problems caused by the trapping microinstruction. The microtrap signal occurs in time to block all three microinstructions in the trap shadow. Therefore, the microtrap logic generates two global signals, the global microtrap (one-cycle long) and the block writes (three-cycles long), at time T_{10}. The purpose of the global-microtrap signal is to trigger any necessary trap-contingent actions in various parts of the processor. The purpose of the block-writes signal is to block register writes at canonical times T_{11}, T_{13}, and T_{15}, thus rendering ineffectual microinstructions U, V, and W in Figure 11. In other words the blocking of writes by hardware is in effect until the trap-handling microcode takes control of the micromachine.

A silo is generally used to save the state of the machine across a microtrap. In most cases the length of the silo is equal to the depth of pipelining. Since there are many more branch-condition bits than microaddress bits, it is more economical to save microaddresses in the trap silo than to save the conditions causing those addresses. Microaddresses U, V, and W must be saved in the silo since they may be branch targets of some previous microinstructions. For the same reason, however, the address X (overridden by X′, the starting address of the trap routine) must be saved as well. During the execution of the trap routine, the trap silos are "frozen" (blocked from loading), thus saving the state of the micromachine at the time of trap.

After the trap routine has completed, two conditions are possible:

1. The recovery from the trap is impossible, and hence the microinstruction sequence cannot be continued. Then the only recourse is to roll back and reexecute the macroinstruction. That is, the macroPC is backed up from its silo, the IB is flushed, and if necessary, any register changes are undone. In this case the last microinstruction of the trap routine performs a trap release, which unblocks the silos so they can resume loading the new states.
2. Microcode can remedy the cause of the trap so that the microinstruction sequence can be continued. In this case the last microinstruction of the trap routine performs a trap return, causing the hardware to recycle microaddresses U, V, W, and X through the microaddress pipe. This action results in the reexecution of aborted microinstructions from the trap shadow.

In the case of a trap return, the hardware selects the microPC silo as the microaddress for the next four cycles. As shown in Figure 14, however, the microPC silo does not contain the microaddresses made by the decoder. Therefore, it is necessary to resynchronize the microinstruction execution sequence with the decoder, while requeuing the trapped microaddresses from the silo. This is made possible by keeping a tag bit in the silo to identify the positions of the microaddresses made by the decoder in the sequence. If a microaddress from the silo is found to be tagged, the requeuing is terminated immediately and the microaddress generated by the decoder is selected. A complete recovery thus occurs since the state of the IB has by this time been backed up, and therefore the decoder-generated microaddress can be used for the continuation.

Chaining of Microtraps

By convention, microtraps are not allowed to nest; instead, they are chained. In other words the trap-handling microcode must ensure that it will not cause any microtraps itself. The sole exception is its last microinstruction, which may cause a second microtrap to follow immediately, even as the saved microaddresses from the silo are being requeued to resume the original flow. Note that this second microtrap does not take effect until four cycles later, whereas intervening microinstructions are blocked by the hardware as a result of this second microtrap. Consequently, the same microaddresses end up in the microPC silo once again during the execution of the second trap routine. The original sequence may finally resume after the last of such chained traps has been serviced.

Acknowledgments

The specification and design of the VAX 8800 I Box was a team effort. Dave Laurello contributed to the IB design, the i-stream data formatter, and the interrupt logic. Bei Pong Wang was responsible for the decoder, the PC increment logic, and the IB-state manager. Jack Ward looked after the physical construction of the sequencer and the control store. The entire development was carried out under the excellent leadership of Doug Clark. Many thanks also go to both Doug Clark and Bob Stewart for their suggestions and guidance during the course of this development.

William A. Samaras

The CPU Clock System in the VAX 8800 Family

The clock system in the VAX 8800 CPU sends timing signals to every state device every 45 nanoseconds. The lack of accuracy of these timing signals is called skew, which must be minimized. Two skews exist: global, between modules; and local, within a module (the lower of the two). The design complexity of the overall system dictated the use of an automated timing verifier. Although advantages accrue from designing for local skew, the verifier could not segregate between skew types. To gain the benefit of the verifier, a unique hardware trade-off was made to minimize total skew: local was made equal to global. The result was that 83 percent of the cycle time is used productively.

All synchronous computers must provide some means of generating and distributing accurate timing signals. The goal of the timing system in the VAX 8800 family is to provide low-skew (therefore, accurate) timing signals to all parts of the processor without any manufacturing adjustments. Furthermore, the design team wanted to automate the verification of the timing during the design phase. Therefore, design trade-offs in the clocking system were necessary to accomplish that automation. This paper discusses how the hardware designs of the clocking system were influenced to provide a good environment for the automatic timing verification.

Clocking System Requirements

The design of the clocking system required us to address many interrelated problems that had to culminate in a common solution. This design depended on certain fundamental specifications that were established for the VAX 8800 CPU by the system architects. The two primary requirements are described below.

Cycle Time

The cycle time of the VAX 8800 family of processors is 45 nanoseconds (ns), which means that a CPU can accomplish some amount of work during that period. Looking at it another way, these processors can do 22.5 million actions every second. Usually, a number of these 45-ns cycles are required by a processor to produce just one VAX instruction. The clocking system must keep the thousands of circuits in the processor "ticking" in perfect step together every 45 ns.

The 8800 was designed to contain two complete CPUs in the same cabinet. Since both CPUs share a common memory, it is beneficial to make the memory system and both CPUs synchronous with each other. The clock system must keep all three items running together, precisely locked in time.

Modules

All the circuitry for both processors and the memory controller is contained on 20 16-inch by 12-inch modules, or printed circuit boards. These modules occupy slots in a 21-inch-wide backplane. Each module contains up to 20 ECL gate arrays and miscellaneous ECL logic. The state devices, called latches, reside both in the gate arrays and the miscellaneous logic of each module.

The Clocking Problem

The basic difficulty for this (and any) clocking system is to get the timing signals to every state device in the machine at precisely the same time. Every synchronous machine faces this problem. However, in faster computers, like the VAX 8800 system, the tolerances placed on the timing signals are more severe. In a physical sense, it is simply not possible to send all the timing signals to every part of each module at the same instant. There is some precision, however, that should and can be achieved. We now discuss how important this tolerance is to the VAX 8800 systems, and what we did to minimize it.

The tolerance, or time difference, that we encounter in attempting to provide timing signals to every state device at the same time is called the clock skew. Clock skew is the uncertainty in the time of a particular event. As an analogy, consider an airline flight that is scheduled to arrive at an airport at precisely 5:02 P.M. Now, we know this flight will not arrive at 5:02 P.M. on the dot; it will probably arrive within a minute or two of that published arrival time. This uncertainty in the time of arrival is the skew of that time. If the uncertainty of arrival is 30 seconds, this skew would probably be a very acceptable value and we would say the flight is right on time: it arrived with low skew.

On the other hand, if the uncertainty of arrival is large, say 30 minutes, we would probably try another airline. Why? Not simply because we are impatient but for a more fundamental reason. When the uncertainty is large, we have *less* time to do other things that are valuable to us. Usually, we are committed to the entire time of the uncertainty. Put another way, this uncertainty, or skew, is wasted time. Enough of this analogy — how does this skew affect the operation of a digital computer?

As mentioned earlier, since the cycle time of each CPU is 45 ns, all state devices are "scheduled" to clock at the start of that period. Any uncertainty in this time from one latch to another is called clock skew. As in our airline example, clock skew is wasted time. There are many factors that increase the clock skew; let us consider one of the most important ones.

Since the backplane width is 21 inches, all the CPU hardware modules are separated by no more than that distance. Since all the wiring in the system is composed of controlled-impedance transmission lines, the logic signals can travel at close to the speed of light. At that speed a logic signal could circle the earth about 4.5 times in 1 second, or it takes about 4 nanoseconds to travel the 21 inches across the processor backplane. Now we can begin to understand the skew problem. The minimum uncertainty of any signal traveling through the entire processor would be at least 4 ns, which is almost 10 percent of the 45-ns cycle. And that is only one source of skew.

Since skew can be wasted time, our goal was to make it as small as possible. In the 8800 system, there are three major contributors to clock skew: variations in the semiconductor components, variations in the wiring lengths (described above), and different manufacturing tolerances of the modules. One common way to remove skew from a system is to make some type of adjustment during the assembly of the hardware. Theoretically, at least, all the skew could be removed through this method of adjustment. To keep the cost of manufacturing low, however, another of our goals was to require no adjustments of any kind. That goal placed an extra burden on the clock system to deliver accurate signals without excessive skew. By carefully designing the circuits of the clocking system and controlling the skew sources mentioned above, we held the overall clock skew in the VAX 8800 family to 7.5 ns. Thus, on average, 83 percent of our 45-ns cycle is utilized. The remainder of the paper explains some of the tradeoffs we made to achieve this figure.

Clock Hardware Overview

Figure 1 depicts the hardware in the clock system of the VAX 8800 family.

The oscillator section is the time base of the whole machine. The implementation is a custom phase-locked-loop design that allows the clock period to be varied for test purposes during the manufacturing process. Using a phase-locked loop makes it possible to have a very accurate timing source at many specific clock periods.

The output of the oscillator section connects to a phase generator that provides two clock phases with the proper timing relationship between them. The outputs (called the A-Clock and the B-Clock) of the phase generator are the actual clock signals distributed to all state devices in the machine. The phase generator is implemented digitally by high-speed, 100K ECL shift registers. This technology creates very accurate timing without requiring any manufacturing adjustments.

Since there is only one phase generator and thousands of state devices requiring the clocks, or timing signals, a method is needed to get the output of the phase generator to every state device without adding very much skew. That is the purpose of the distribution stage of the clock system. The actual circuitry used for the distribution consists of 100K ECL differential devices and 10KH ECL devices. The distribution was

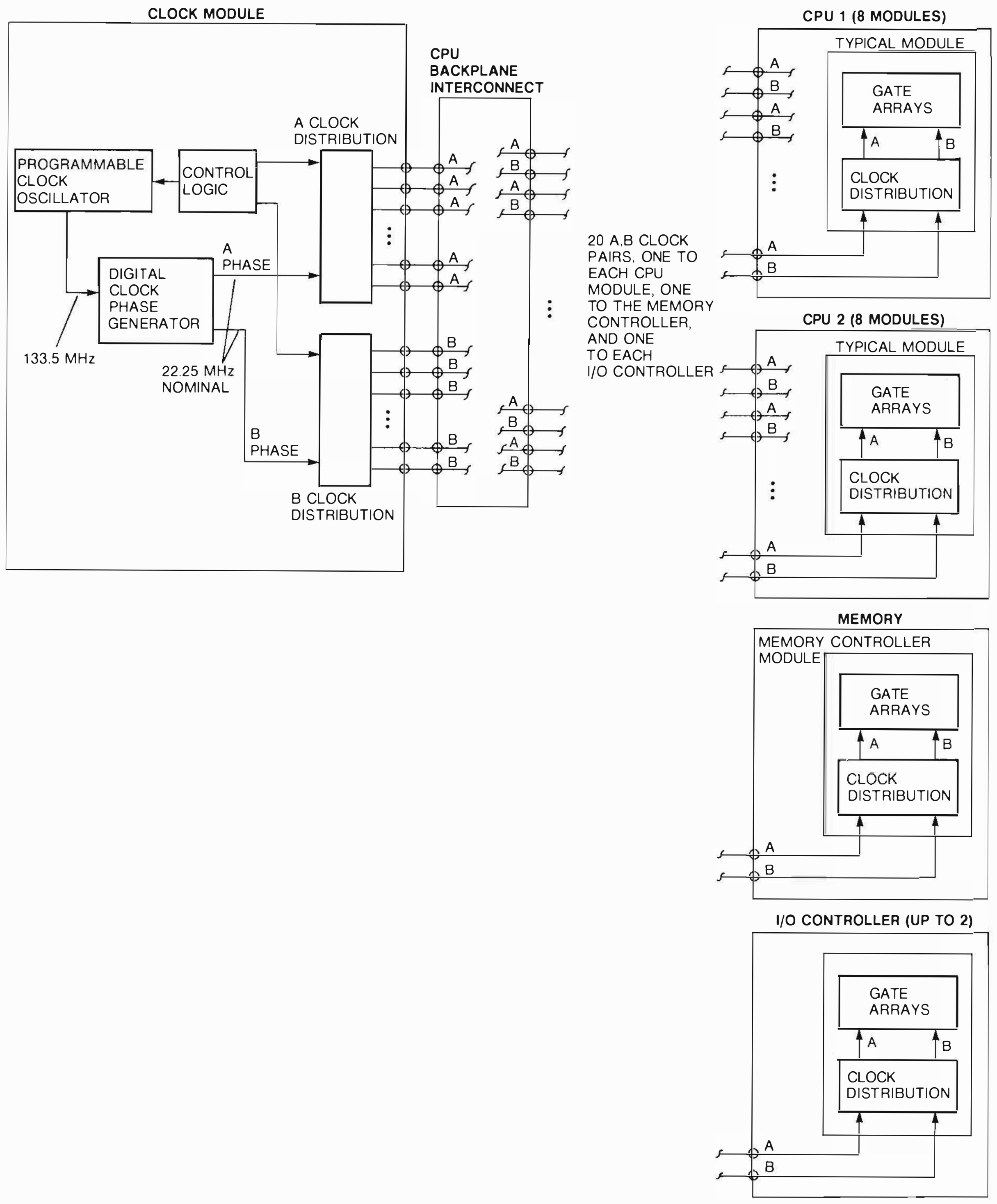


Figure 1 Clock System in VAX 8800 Family

heavily influenced by our desire to use an automatic timing verifier. The following discussion of the timing verification environment gives a clearer view of the reasoning behind the clock distribution scheme.

Clock System and the Timing Verification Environment

Traditionally, timing verification was accomplished by hand calculations using component specifications. A designer would simply add all the component propagation delays in a particular path and determine if all timing criteria were met. In the past, this method worked fairly well for several reasons. First, the designer usually knew which paths in a circuit were critical and could give special attention to them. Second, components generally behaved better than their worst-case vendor specifications.

Marginal timing problems, or ones that were simply overlooked, would often be less serious than the difference between the worst-case specifications and how the components actually worked. Finally, timing errors were *expected* to appear during the hardware debug phase of a project. Therefore, timing errors that were blatantly missed during the design could be corrected (with a lot of hard work) during that phase. That was possible because the overall complexity of the design could be comprehended by the designers.

From the beginning of the VAX 8800 design effort, we knew that the timing of the design would be difficult to analyze manually. First, the sheer complexity of the machine created over four million different timing paths. It was impossible to analyze every path manually or to discover every "critical" one with either manual or intuitive analysis methods.

Second, hardware circuit loops are widely used in the design; these are circuits that feed signals back to themselves during a later machine cycle. These circuits are very difficult to analyze, especially when loops cross physical boundaries or are nested within other loops. Just thinking about the timing ramifications of nested loops taxes the mind. Manually analyzing thousands of these cases would be impossible.

Finally, the hardware design made heavy use of gate arrays, which contain most of the logic. Our ambitious development schedule and the large number of gate array designs simply could not tolerate unanticipated timing errors. A timing error in a gate array meant that a new gate array must be produced to fix the problem. The fabrication overhead for another semiconductor device, usually taking months, was not consistent with our development schedule. Moreover, while that new gate array was being fabricated, the debugging of the entire system could be jeopardized since it was just not possible to "fix" an LSI chip.

Therefore, the hardware design group wanted to design the processor with the aid of an automatic CAD tool for timing verification. Such an automatic method for verifying the timing was essential to the success of the project. Since the entire design was to be "soft" (the schematics were contained in computer databases), it seemed logical that some type of software tool for automatic timing verification could be applied.

We decided that the most appropriate timing verifier for this project was produced by Valid Logic, Inc. Although this automatic tool solved the problems caused by manual timing verification, it also created some very special new restrictions.

It was apparent from the beginning of the design effort that some restrictions had to be placed on the design styles of individual engineers to reduce the timing-analysis problem to a manageable level. CPU hardware designers, like any other creative persons, often assume large degrees of freedom in their work. Usually, no two designers will arrive at the same solution to a problem, although all solutions may be acceptable. When ten or more designers work independently, as happened on this project, it is likely that ten unique design styles will emerge.

Therefore, we placed restrictions on the timing environment for the following two reasons:

- Some standardization of timing had to take place for electrical signals to communicate properly between designs generated by different people.
- Since the automatic timing verification software was new, several important features were lacking.

The usefulness of an automatic timing verifier depends largely on how well timing-rule violations are reported. Knowing that a design contains timing errors is useful only if it is easy to find them. One way to aid the reporting of timing errors is to create an environment that clocks all state devices in the processor the same way. This means that all logic designs in the processor must follow consistent and strict rules for the clocking of state devices. That was the method we decided to pursue in this design project.

The Timing Environment

The clock system needed strict constraints on its circuit design and physical layout to guarantee accuracy. Therefore, the generation and use of clocking signals were tightly controlled to minimize the different ways in which the circuits could communicate. The timing control of state devices had to be consistent throughout the design. Moreover, any arbitrary timing control of the state devices would have been an impossible task for the timing verification software.

The timing signals in the VAX 8800 processor were carefully distributed to every state device. This distribution was accomplished by carefully

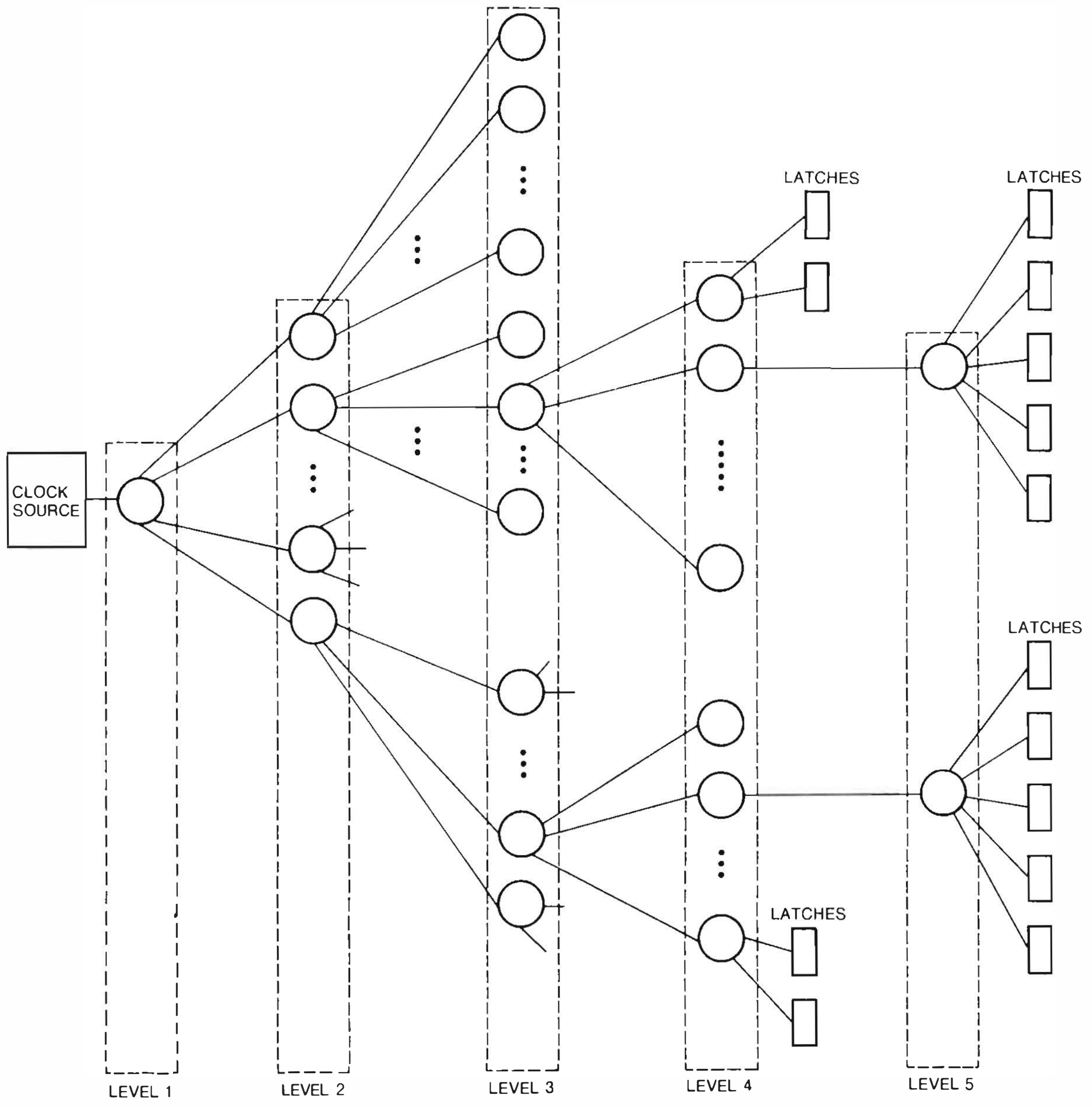


Figure 2 Clock Expansion Groups

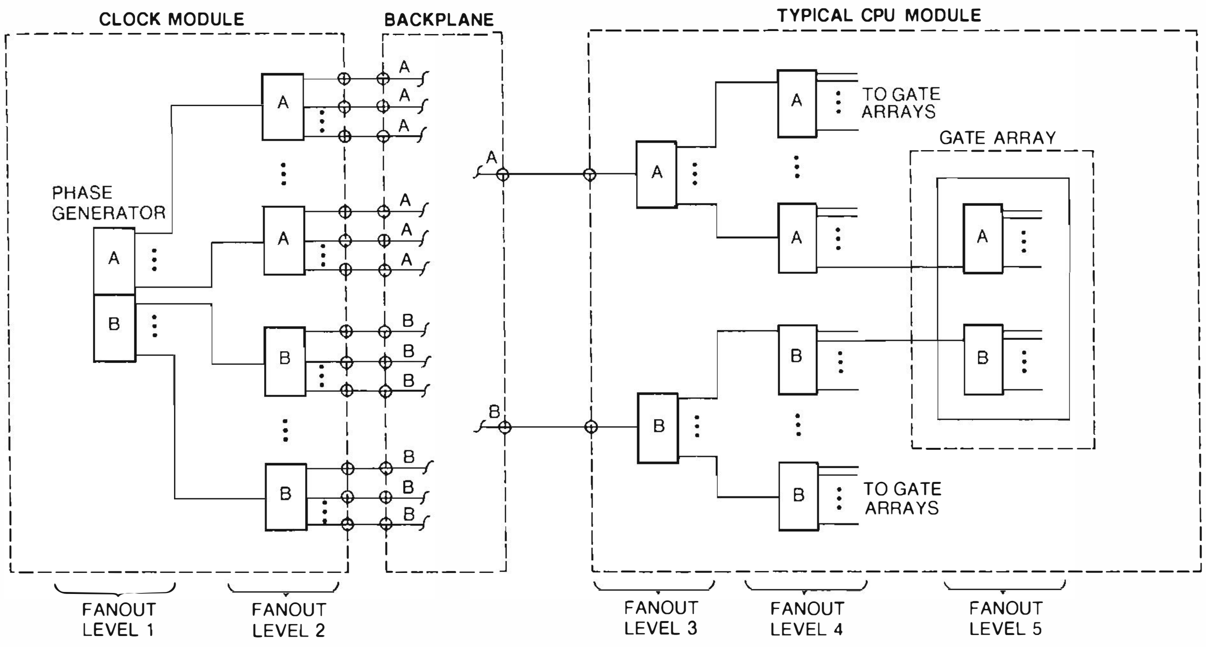


Figure 3 Minimized Global Skew Distribution

expanding the clock signals at strategic physical positions in the processor. A simple example of this expansion, or fan-out, is shown in Figure 2.

Each time the clock signals are expanded, more timing uncertainty is introduced into the resulting signals. The 8800 design required up to five levels of expansion to produce enough clock signals for every state device. As shown in Figure 2, some signals are in common distribution groups. Signals existing in the same group will have low timing uncertainty between them, a characteristic called skew correlation. The timing uncertainty between signals in different distribution groups has no correlation; therefore, these signals have the highest skew. Signals from the same group have a skew, called local skew, lower than the overall group-to-group skew, called global skew.

It is very tempting for designers to take advantage of the lower local skew, which is often only half that of the global skew. Each clock distribution group is usually contained entirely on one logic module due to the natural physical partitioning of the hardware. Therefore, communication between circuits on any particular module can take advantage of the lower local skew. If all signal communication occurs within the local-skew environment, the timing analysis can be consistent and easily managed. However, complications arise when trying to analyze signals that cross from the local-skew environment to the global-skew environment. Signal communication between logic modules will have to pay the penalty of using the higher global skew because the timing signals at each end of the communication are derived from different distribution groups. Managing the timing interface across this partition between local and global skews was beyond the capabilities of the timing verification software.

As discussed earlier, a timing analysis of the entire processor was beyond human capacity; therefore, it had to be performed with timing verification software. The timing verification tool chosen for the 8800 development had no facility for distinguishing between local and global skews. Moreover, we wanted to use the timing verifier to analyze the timing of the entire CPU as one entity. This decision forced us to disallow the use of any local-skew computations in our timing analysis. Now, from a design point of view this decision made the environment very easy to work with. All timing transactions anywhere in the CPU could be analyzed the same

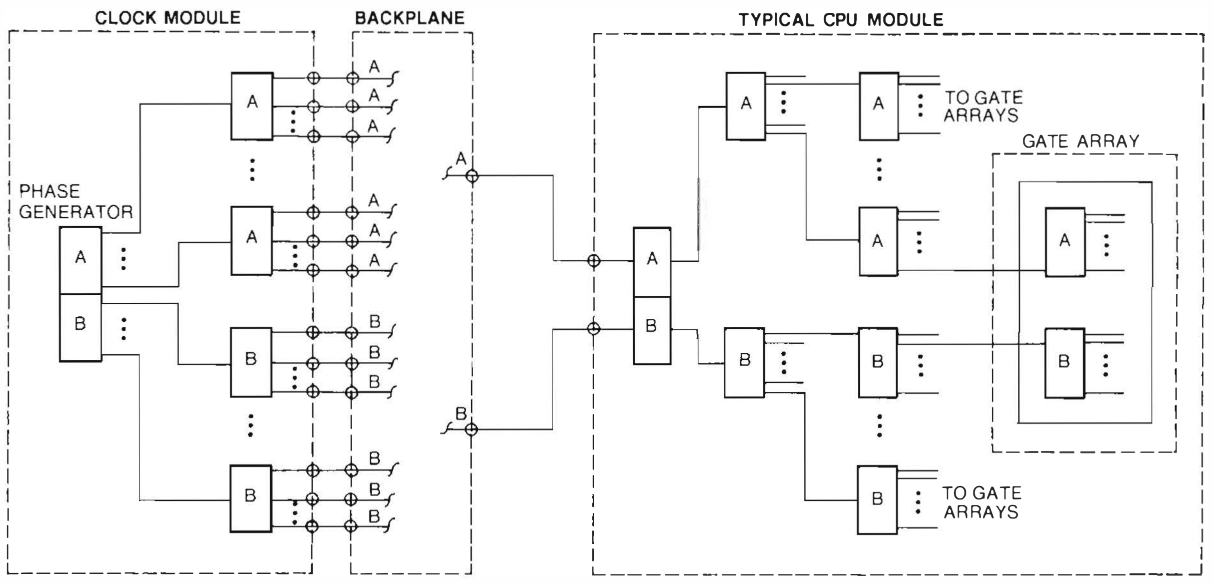


Figure 4 Minimized Local Skew Distribution

way with the same set of specifications. Everything comes at a price, however, and the obvious negative side of this decision was the loss of the ability to apply the lower local skew. At that point, some performance of the processor seemed to be compromised just to simplify the timing analysis. The following discussion explains how this problem was solved.

The Clock Distribution Solution

Since we wanted to time the CPU as one entity, we had to make the global skew as small as possible to maximize CPU performance. In the actual implementation, the global skew was lowered by removing one gating level from the clock distribution. The gating level removed was necessary for producing low local skew. Figure 3 illustrates the five levels of fan-out that were required to produce enough signals when the global-skew distribution was minimized. Figure 4 shows the same fan-out to produce enough signals in the case in which the local-skew distribution would be minimized. Table 1 illustrates the impact of this optimization for global skew.

Table 1 Distribution Changes

	Global Skew	Local Skew
Optimized Local Skew	9 ns	2 ns
Optimized Global Skew	7.5 ns	7.5 ns

Although using the lower local skew would have been valuable, it was sacrificed by making it equal to the global skew.

In short, the hardware of the clock system was designed to allow the maximum exploitation of the timing verification software. Of course, hardware and software trade-offs are a common occurrence in any design project. In this case, however, the value of the hardware involved with operating the machine was balanced against the software analysis needed during the design phase of the machine.

Summary

Producing the clocking system for a high-speed computer is best described as an exercise in minimizing and managing skew. In the VAX 8800 project, we avoided exotic hardware techniques so that we could gain the benefit of using an automatic timing verifier. The resulting skew of 17 percent of the cycle time was a figure that could be tolerated. This balance was a fair trade-off since the simplicity of the timing environment allowed us to decrease the time to design and build the VAX 8800 family of systems.

John Fu
James B. Keller
Kenneth J. Haduch

Aspects of the VAX 8800 C Box Design

In each processor in the VAX 8800 family, instructions and data are supplied to the execution units by the C Box. Employing a simple structure with a translation buffer, cache, and address and data buffers, this logic unit is an integral part of the processor's five-stage pipeline. The no-write allocate cache uses a write-through scheme featuring a unique delayed-write algorithm. The C Box has control logic to accommodate pipeline stall conditions caused by memory accesses. The C Box also maintains data coherency within a processor and between processors. A dynamic priority-arbitration scheme solves the lock-out problem between I/O and processor requests.

The performance of a high-speed computer depends to a large extent on how fast data can be passed from its memory to its execution units. If the computer is pipelined, the unit responsible for memory accesses may have to handle pipeline stall conditions. And if the computer is a multiprocessor, that unit in each processor may also have to handle data coherency problems. In processors with the VAX architecture, data accesses are further complicated by the fact that virtual addresses are normally specified. These addresses require translation to physical addresses before a data access can even be attempted.

In the VAX 8800 system, which is a multiprocessor with pipelined CPUs, the unit that performs address translations and data accesses is the C Box.

C Box Description

The C Box consists of three subunits: the translation buffer (TB), the cache, and the NMI interface. Figure 1 is a schematic diagram of this unit.

The translation of a VAX virtual address to a physical address is a complicated process.[1] Accesses to system and process page tables are required, and shifting and adding must be done to obtain the final physical address. Performing this address translation process for every data reference significantly increases the data access time and reduces the read bandwidth. One way to avoid that is to store the result of this address calculation in a small, fast memory called a translation buffer. Since each translation can access a page of data (512 bytes in the VAX architecture), it is likely that the translation will be used again in the program being executed. Rather than recalculating the physical address (PA) on those subsequent accesses, it can be retrieved from the TB.

The translation buffer in the VAX 8800 processor holds 512 system and 512 process address translations. The following summarizes the characteristics of the TB.

Characteristics of the Translation Buffer

- Direct Mapped
- 1024 Lines
  - 512 System Lines
  - 512 Process Lines
- Allocation on Translation Buffer Miss

A common approach to the problem of data access latency for high-speed processors, and the one used in the VAX 8800 CPU, is to use a cache.[2] A cache is a small, fast memory located between the processor and the main memory system. If the data requested by the CPU is not contained in the cache, that data is accessed from main memory and loaded into the cache.

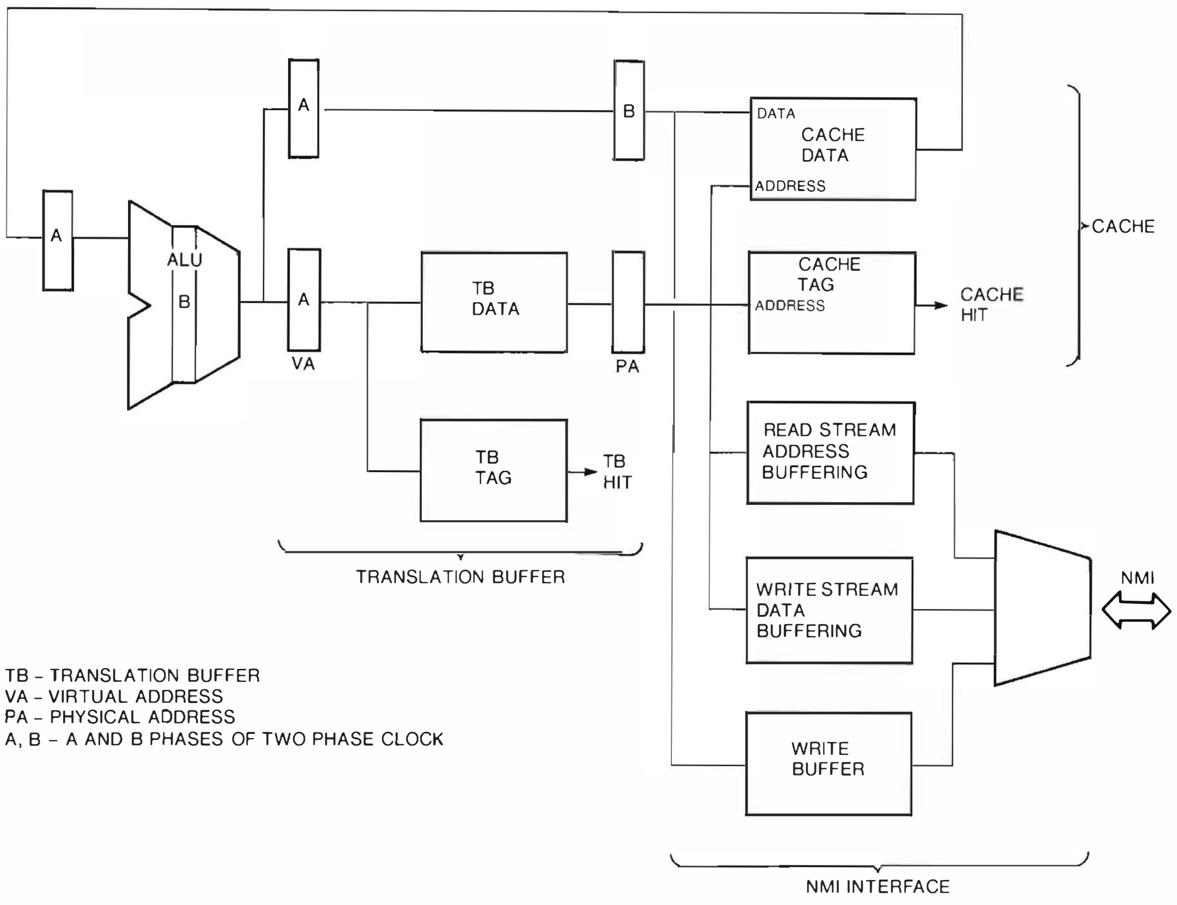


Figure 1 Block Diagram of C Box

Thus, in the majority of cases, the cache will contain recently referenced data items, and future references to those data items will be fetched from the cache. The intent is to minimize the number of longer latency accesses to the main memory subsystem. The success of a cache memory relies on the locality of references in both time and space.

The data cache in each VAX 8800 CPU holds 64 kilobytes (KB) of both data and instructions. The list on the right summarizes the characteristics of the cache.

The TB and the cache are very similar in concept and structure, except that the TB is used to accelerate address translations and the cache to accelerate data accesses. Each consists of a tag section and a data section. The tag section holds the unique identifier, or tag, for the data item held in the corresponding data section. The TB and the cache are direct mapped, meaning that each address can point to only one location; however, each location can potentially be allocated to one of many addresses. A tag permits the identification of a data item in either the TB or a cache location. The tag in the VAX 8800 processor is an unmodified selection of bits

Characteristics of the Cache

- Direct Mapped with Physical Address
- Read Allocate Only
- Delayed-Write Cache Update
- Write-through Memory Update with Write Buffering
- 1024 Blocks
- 64-byte Block Size
- 4-byte (one longword) Line Size
- 32-byte (one hexword) Cache Refill Size

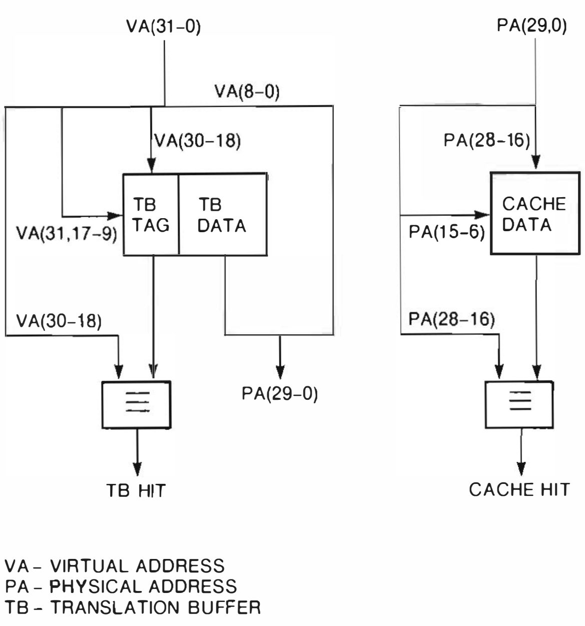


Figure 2 Translation Buffer and Cache Address Mapping

from the address of the data item being accessed. This concept is depicted in Figure 2.

As mentioned earlier, a memory access is required if the cache does not contain a requested data item. In the 8800, both processors are connected to the memory and the I/O subsystems through the NMI bus. All read and write references that go to these subsystems are processed by the NMI interface. This interface maintains a set of buffers for both read and write reference streams. For the read stream there are actually two sets of address buffers: one for data reads, the other for instruction reads.

C Box Operations

A C Box reference consists of a function code, an address, and in the case of writes, 32 bits of data. In general, that address is a 32-bit virtual address (VA). The VA translation process begins with a check to see if the PA is available in the TB. If the PA is available, called a TB hit, the data is read out and concatenated with the lower nine bits of the VA to form the PA. As part of the translation process, the TB also performs page access checking. If the PA that pertains to the VA is not in the TB, called a TB miss, then microcode must perform the translation. The microcode then writes the data into the TB for subsequent use. (If the address supplied is already a PA, then the TB is not used.)

Only physical addresses access the cache. If the data referenced is contained in the cache, called a cache hit, then the data can be accessed from there. If the cache does not contain the data, called a cache miss, then the data must be accessed from memory.

Read Operations

Cache-miss addresses for reads are passed to the NMI interface, where they are held in the read address buffers. A hexword read request (32 bytes), with the address of the missed location, is then made to memory. The memory data is passed to the requesting unit, and the address held in the read address buffer is used to update the missed cache location. A read miss is the only occasion upon which a cache location is allocated.

There are two read streams in the C Box for requests to memory: the data stream, called the d-stream, and the instruction stream, called the i-stream. The i-stream requests the memory to send data destined for the instruction unit (I Box), which interprets that data as macroinstructions. I-stream fetches are initiated by microcode, which loads a C Box register called the physical instruction buffer address (PIBA). The PIBA holds the address of the next longword of the i-stream to be fetched. If the execution of macroinstructions is sequential (i.e., there are no branches, page crosses, etc.), the C Box can increment the PIBA contents automatically after each fetch. However, should the program branch or a page cross occur, microcode must be used to reload the PIBA. D-stream fetches are made only by the microcode, which must specify one of eight memory data (MD) registers as its destination. D-stream data is always returned to the execution unit.

Write Operations

In general, the performance of a cache is measured by its hit rate when reading data. The selection of the update mechanisms for both cache and memory, however, can have a major influence on the design of the cache. There are two well known strategies for updating a cache: write allocate, and no-write allocate. A write-allocate scheme updates a cache location whether or not the write is a hit or a miss. This scheme is generally implemented with a write-back memory arrangement (discussed later). In a no-write allocate scheme, the cache is updated only if the write was a hit. The VAX 8800 processor uses a no-write allocate scheme.

The no-write allocate scheme does, however, present a problem. Since only writes that hit will update the cache, cache updates take two pipeline cycles in the C Box — the first to check for hit or miss, the second to update the cache for a hit. The C Box was designed to enable one read reference to complete in each cycle. If two consecutive cycles are needed to update the cache, the second cycle could block a read reference, thus causing a pipeline stall.

To solve this problem, the C Box implements a delayed-write algorithm. This mechanism delays writes that must update the cache from doing so until the first cycle of the next write reference. The second cycle of the delayed write does not need to be the next consecutive cycle.

The delayed-write algorithm in the C Box takes advantage of the fact that the first cycle of a write utilizes only the tag section of the cache to determine whether a hit or a miss has occurred. The second cycle uses only the data section. A write that must update the cache has its address and data placed into the delayed-write address and data buffers respectively. On the next write access, during the cache-tag look-up cycle, the data section of the cache will be updated from the address and data contained in those buffers, but only if the previous write access was a hit. Since reading a data item after one has been written is common, this design significantly reduces the potential for stalls.

Write Buffer

All write references, whether or not they hit in the cache, must eventually go to memory. There are two general strategies in cache design with respect to memory updating: write-through, and write-back. In the write-through approach, write references are sent to the memory system immediately. Conversely, in the write-back approach, writes are held until the cache block is deallocated (made ready to receive different data).

There are several major problems with a write-back strategy. First, it requires either microcode or hardware to accomplish all the write-back functions. Adding that code or hardware to the C Box would have considerably increased its complexity.

Second, if there is a write miss with this scheme, a cache block that might be full of valid data could be displaced by a block whose only valid data was that just written to the cache. For a cache having a large block size, like the 8800 has, this action is undesirable. Moreover, in most cases microcode reads data before it is written; therefore, writes will generally hit in the cache.

Finally, the write-back strategy requires a complex algorithm to maintain coherency between caches within a multiprocessor system. Therefore, for all those reasons, we chose to use the write-through approach in the cache.

One disadvantage of write-through is that it tends to generate a lot of write traffic to the memory. In a shared-bus system like the 8800, this traffic can limit performance. To reduce memory-write traffic, writes in the VAX 8800 processor are buffered in a write buffer contained in the NMI interface. This write buffer is really a one-line, octaword, write-allocate cache. A write going out to the NMI bus is held in the write buffer. Subsequent writes to the same octaword update only the write buffer so that no memory requests are sent on the NMI bus. A write that is outside the octaword currently in the write buffer deallocates it; that is, the contents of the write buffer are sent to memory, and the next write replaces those contents in the buffer.

Like the cache, the success of the write buffer in reducing bus traffic relies on the locality of programs in space and time. For example, sequential writes, such as pushes to the stack, will get collected in the write buffer even if the writes occurred in different macroinstructions. This collected "package" of writes can then be sent to the memory more efficiently than can individual writes.

Another advantage of the write buffer is that it decouples the processor from memory activity. When the memory is busy processing transactions from the other processor or from the I/O subsystem, a processor will not stall due to writes. The write buffer is actually implemented as a two-deep buffer, which further reduces the potential for stalls.

Pipeline Stalls

In a pipelined implementation, how well the pipeline performs is determined both by how often it is flushed clear and how often it is stalled. Stall conditions are generally related to the lack of some physical resource or data.

In some implementations, some pipeline stages can take more cycles to complete than others for certain functions. If a shorter stage precedes a longer one, the longer one will be unable either to accept fresh data or to pass its result to the next stage until finished with its cycle. In turn, other portions of the pipeline cannot proceed with their operations; therefore, the pipeline will stall. In this stalled condition, all stages preceding the "bottleneck" maintain their input and output conditions until the stage responsible for the stall completes its function. Some implementations have a combination of stages that may exhibit these characteristics, leading to complex pipeline stall conditions.

In the VAX 8800 CPU, the design simplicity of the pipeline ensures that each pipeline stage — except the C Box — always completes its function in one cycle.[3] Since the C Box also controls data accesses, all stalls in the 8800 are related to the operation of this unit. The pipeline will experience two types of stalls: the MD stall, and the VA stall.

MD Stalls

When making a read reference, a microinstruction must specify one of eight MD registers to be used as its destination. When data is made available, either from the cache or from memory, it is written into the specified MD register. Subsequent microinstructions then use the data from this register. If a microinstruction attempts to use an MD register that is not "valid" (i.e., the data has not yet been fetched by the C Box), the pipeline will experience an MD stall.

The MD stall condition is a data-dependency type of stall that is generally seen in pipelined machines. On the VAX 8800 processor, certain steps are taken to either avoid such stalls or reduce their effects. For example, consider two consecutive microinstructions, R and S, as illustrated in Figure 3. R is a microinstruction that performs a read and puts data into an MD register. S then accesses and uses the data fetched by R. If R and S are adjacent, the pipeline will stall in the 8800. The reason for the stall is that the pipeline stage accessing the MD data and the stage fetching that data (the C Box) are separated by one other stage, the arithmetic and logic unit (ALU). When S tries to use the MD data, R is just starting to make the read reference in the C Box. S must therefore stall the pipeline, waiting for data to be supplied by R.

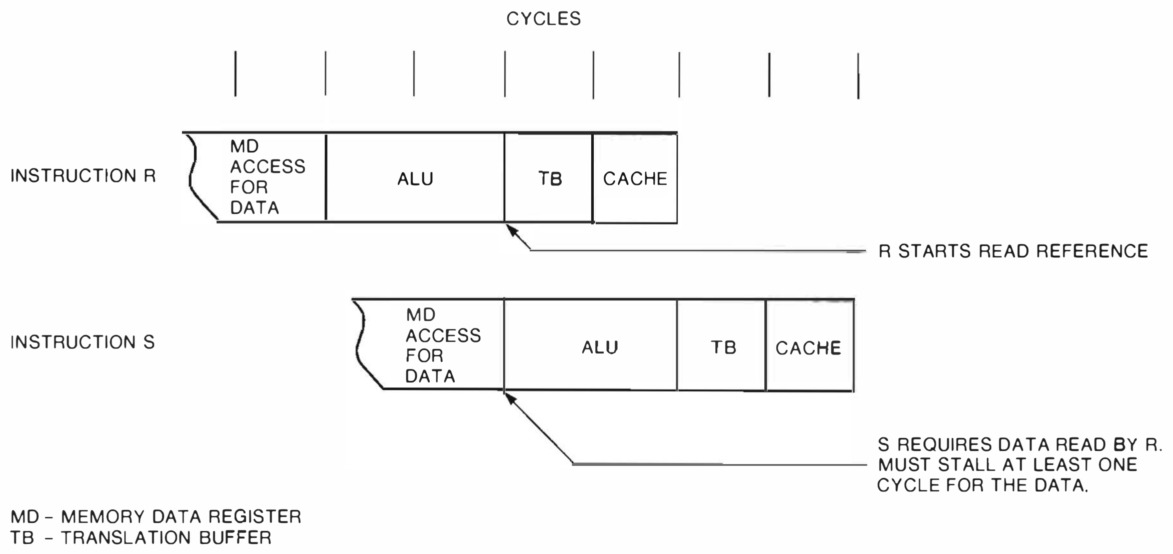


Figure 3 Instructions R and S Are Adjacent

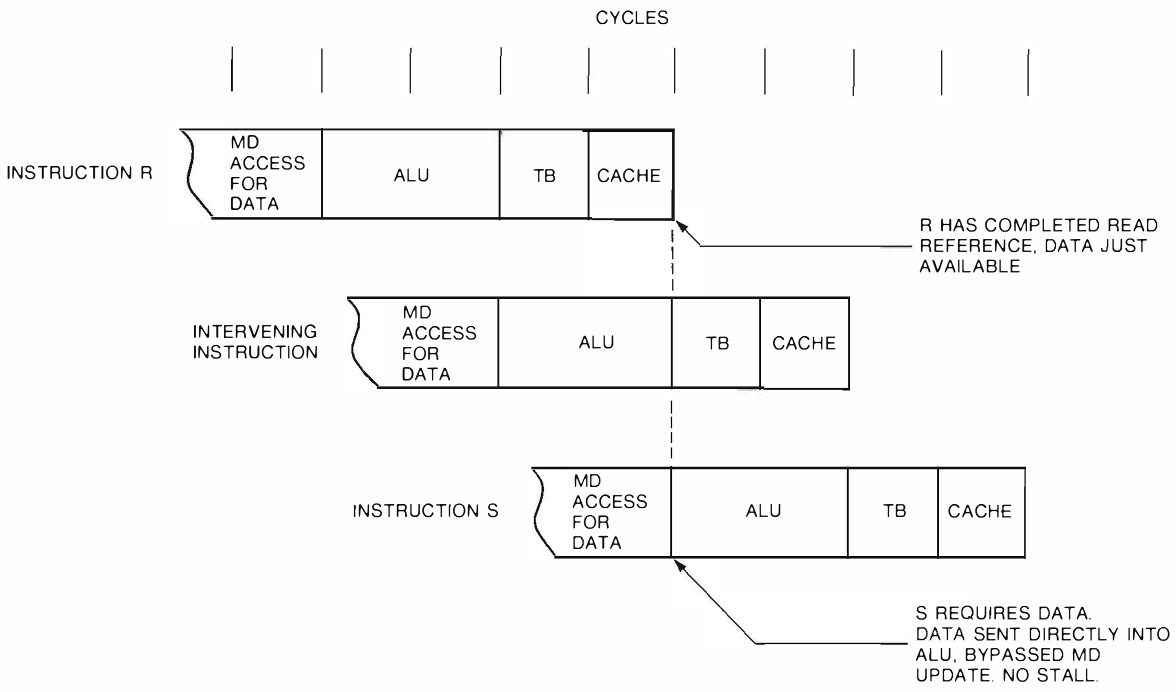


Figure 4 Instructions R and S Separated by Another Instruction

On the other hand, if R and S are separated by one other instruction, then when S attempts to use the data read by R, that data is just being made available by the C Box (assuming, of course, a read hit in the cache). If S were to wait for the MD registers to be updated before using the data, the pipeline would stall. To eliminate that type of stall, a path has been designed from the C Box directly into the input of the ALU, bypassing the MD registers. Therefore, the data coming from the cache is sent both to the MD registers for updating and directly to the ALU, where S can use the data. The net effect is that this bypass path removes the one-cycle latency that S would have experienced had it waited for the data to come out of the MD registers. Figure 4 illustrates these concepts.

Had R caused a read miss, S would still cause an MD stall since the C Box must make a memory fetch for the data. Notice that an MD stall happens only when S attempts to use an MD register. Therefore, a general rule for making microcode accesses to the C Box is to make read references early and to use the MD registers late. Should the read reference miss, some part of the memory-fetch latency will be hidden by the microinstructions between the read and the MD register access. When data returns from a read miss and the pipeline is either undergoing or about to undergo an MD stall, the bypass path can be used to reduce the effects of the stall or even prevent it.

VA Stalls

A VA stall condition occurs when the C Box cannot process a requested reference. This can be due to either an invalidation cycle in the C Box (discussed in the final section of this paper) or the capabilities of the address and data buffers in the NMI interface being exceeded.

As mentioned earlier, for reads there is a set of buffers for d-stream and i-stream references. The d-stream buffering is one deep, meaning there can only be one read miss outstanding in the C Box. However, the implementation will not allow the pipeline to stall should subsequent reads hit in the cache. I-stream reads never stall the pipeline as do VA and MD stalls, which stop the clock. The instruction buffer can "stall" if it does not have enough data for the decoder to complete the decode of the current VAX instruction operand. This condition causes the CPU to perform a no-operation microword. That does not stop the clock, however, and thus is not a pipeline stall.

The C Box can still receive commands even if it contains one read miss. Of course, there is the potential that the command being received will miss in the cache. That will require the NMI interface to request the data from memory, thus resulting in a VA stall. That stall lasts from the time the command is received until the time the previous read-miss data returns from memory. If the second command is a read that hits in the cache, a VA stall will be generated for the one cycle that it takes to determine whether or not there is a cache hit. The read data will then be taken from the cache and returned to the MD, after which the stall will be released.

Since writes go to memory more than reads, the buffering for writes is more extensive. The delay-write buffer and the double buffering in the write buffer are used to reduce the possibility of write stalls. These buffers enable the C Box to hold a maximum of nine longwords of data before the pipeline will experience a VA stall on a write.

Stalled and Unstalled Logic in the C Box

If an instruction is stalled, the C Box has either not returned the data or cannot take another reference. Therefore, all stages prior to the C Box (the I Box and the E Box) must be stalled. The TB is part of the last stage of the pipeline; therefore, it must be capable of being stalled. When the pipeline stalls, the TB holds the address of the stalled reference. Only the NMI interface can resolve a stall, either by supplying the read-miss data or by freeing up its buffers. Thus this interface can never be stalled. However, the cache, being part of the last stage of the pipeline, is also the path for supplying data to the stalled instruction. This situation leads to an interesting control characteristic of the C Box. One of its sections, the TB, can be stalled; another, the NMI interface, must never stall; and the third section, the cache, must remain unstalled but maintain stalled input and output conditions in its logic. Figure 5 depicts the logic for stalled and unstalled conditions in the C Box.

Coherency Problems in the C Box

In general, data coherency means that a read should always get correctly modified data when a series of reads and writes is made in any sequence. One way to maintain coherency is to perform all reads and writes to completion in a purely sequential manner, thus strictly maintaining their sequence of reference. However, in a pipelined machine, not only can there be several sources of read and write references, but there can also be more than one copy of the data item. This duplication often leads to very complex solutions to achieve coherency.

This complexity has been simplified somewhat in the VAX 8800 pipeline by having the C Box both control and sequence all data accesses. The C Box itself, however, is pipelined, having a d-stream and an i-stream for reads, and a stream for writes. This fact also presents some coherency problems. Coherency for the C Box means that two conditions must be met.

1. After a sequence of reads and writes has completed, any valid blocks in the cache must match the data in the memory.
2. Whenever the processor writes to a location in memory and then reads that location, the data has to be what was written.

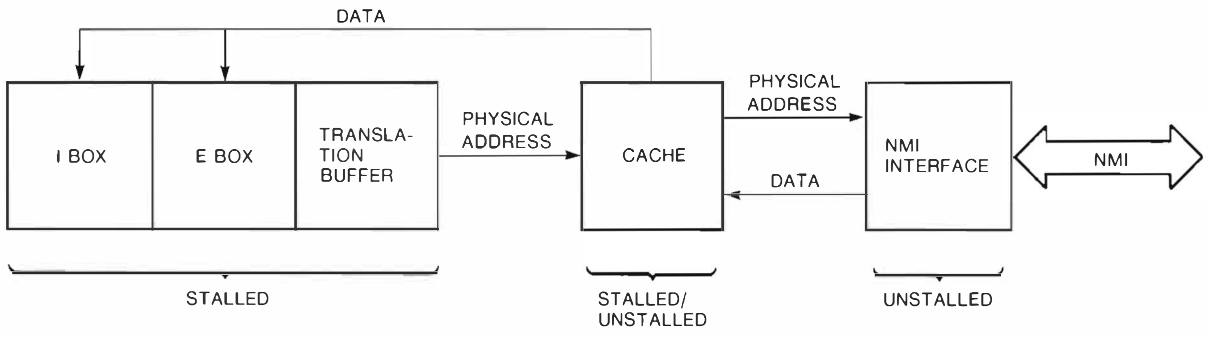


Figure 5 Stalled and Unstalled Logic in C Box

Two types of coherency problems exist in the VAX 8800 system: coherency within a processor, and coherency between processors.

The first type of problem in the C Box arises from the implementation of the delay-write algorithm discussed earlier. A problem occurs when a read is attempted to the cache location waiting to be updated by the write held in the delay-write buffers. The read will hit, but the cache data will be stale. One solution to this problem is to stall the pipeline while the cache is updated, performing the read for the correct data. The trouble here is that the sequence of writing to and reading from the same location is a common occurrence. Thus to stall would significantly reduce the read bandwidth.

The C Box solves this problem by comparing selected bits of the read and write addresses in the delay-write buffer. If the bits match, then the data content of that buffer is used as the read data. This solution works because, to the read, the delay-write buffer appears to be an extension of the cache. Since the read address matched the address in this buffer, the data can be taken directly from it. Coherency is thus assured, and no stall penalty is incurred.

The second type of coherency problem occurs when the read is a miss and thus goes to the NMI interface. To assure high performance, the NMI interface maintains two streams of data requests, the read and write streams. The buffering and the control of these two streams operate independently. If made to different data items, read and write requests can be processed to memory as quickly as possible, even out of sequence. The coherency problem is to make sure that subsequent reads and writes to the same data item result in its correct state.

If a read request occurs that was a miss, the cache will send it to the NMI interface upon discovering that fact. Once in the NMI interface, the read address is compared to the address of the octaword in the write buffer. If those addresses are different, the cache will send the read directly to memory. Thus the data in the write buffer will be unaffected. If the addresses match, however, the write data will be sent to memory, followed by the read request. Since the memory subsystem processes references in a sequential manner, the read will always access the correct data. (Of course, this case is fairly simple. A more complicated one is that in which a read is sent to memory, and the processor performs a write while waiting for that read.)

If the addresses of the read and write match, the cache can give the processor the requested data but cannot mark the returned data valid in the cache. This situation occurs because the read-miss data being fetched from memory has been made stale for subsequent reads.

The microcode is designed so that it will never read a data item and then write to it without first accessing the MD registers. However, a cache block is 64 bytes long. The microcode could write to any other data item in the block before coming to the missed data item. There can be as many as three writes and two reads (one each for the d- and i-streams) buffered simultaneously in the C Box, all referencing the same cache block. Even worse, the C Box can send an arbitrary number of writes to memory while waiting for the data returned by the read to memory. To maintain coherency, the C Box performs a set of address matches between the read and write streams. Then it "remembers" whether or not any write addresses matched the outstanding reads and marks them invalid as appropriate.

C Box Design for a Multiprocessor System

The VAX 8800 system consists of two identical VAX 8800 processors on the NMI bus connected to the memory and I/O subsystems. Within a processor, only the design of the C Box has been affected by the requirements of a multiprocessor arrangement. That is because the C box is the CPU's interface to the NMI bus and contains the central arbitration logic for that bus.

There are three key issues in designing a memory interconnect for a multiprocessor system: bus arbitration, bus bandwidth, and data coherency between processors.

Bus Arbitration on the NMI Bus

Two major problems were encountered in the design of an arbitration scheme for the NMI bus. The first was the fact that between the CPUs and the I/O subsystems, called the NBIs, there was a possibility that a high-priority device could lock out a low-priority device from the bus. This is certainly possible with a fixed priority-arbitration scheme. To address this problem, the C Box implements a dynamic priority-allocation scheme that causes priority to be assigned between two groups: the I/O devices, and the CPUs. Within these groups, the priority shifts between the two CPUs and the two I/O devices. For example, if all four devices wanted to use the bus all the time, the order in which the bus would be granted to the devices would be

first CPU, first I/O, second CPU, second I/O,

first CPU, first I/O, second CPU, second I/O, etc.

This scheme guarantees that all devices on the bus will have nearly equal access to the bus, thus solving the lock-out problem.

The second problem involves the "memory busy" situation. Whenever the memory subsystem cannot process more requests, it sends a "memory busy" signal. It could happen, for instance, that a CPU accesses the bus and attempts to write to memory. Upon receiving a memory-busy signal, the CPU will abort the write. When memory is released, some other device will access the bus and perform a write, thus filling the write queue in memory. Once again, the first CPU re-arbitrates, accesses the bus, and tries to write. Once again, that CPU receives a memory busy signal. And so on.

The NMI arbitration scheme mentioned above solves this problem in which a device might get locked-out of memory. As implemented, the arbitration scheme saves the priority state at the time before the memory-busy signal was asserted. The arbitration logic then restores that state so that the device that received the signal will get the bus when the memory-busy signal is deasserted.

Bus Bandwidth

For the processors on the interconnect, bus bandwidth involves two components: read bandwidth, and write bandwidth. The problem of inadequate read bandwidth is addressed by having a high hit-rate cache. The higher the hit rate, the fewer the requests to memory. The problem of inadequate write bandwidth can be treated in two ways. The first way is to have a write-back cache like the one on the VAX 8650 processor.[1] Such a cache writes a block to memory only when the cache block is deallocated. This technique can significantly reduce the write bandwidth requirements.

In multiprocessor systems like the 8800, however, in which each processor has an internal cache, this technique becomes complicated. In these systems, a data item can exist not only in memory but also in all the caches. To maintain coherency, each write-back cache would have to notify the other cache when the first cache writes. This technique usually leads to a complex protocol and design implementation.

Another approach in a multiprocessor system, the one used in the 8800, is to implement write-through caches. In such an approach, all write references go directly to memory so that each cache on the bus can "see" all write activity. The caches can then be invalidated. Such an approach greatly simplifies the protocol for cache coherency but, as discussed earlier, generates a high degree of write traffic. The unique design of the write buffer helps to reduce this traffic, although not as much as a write-back cache would. In the 8800 processor, however, the write buffer reduces traffic enough so that the two VAX 8800 processors can write at their maximum bandwidths on the NMI bus.

Coherency in a Multiprocessor System

A multiprocessor system, with internal caches, presents a number of interesting coherency issues when sharing data. Ideally, if one processor writes to a location and the other processor reads that location, the read will always get the data that was written. In practice, achieving this condition is difficult. Several major questions arise: Did the read happen before the write or after it? What happens if both processors write to the same location at the same time? Unless controlled, these situations can produce unpredictable results.

If programs on the processors want to share data, they must use the interlock instructions in the VAX architecture.[5] Only after an interlock instruction is processed will the memory location be guaranteed to have the correct data. The general method is as follows. Processes must decide to share a block of memory. One memory location is called the software lock, and only one process at a time is allowed to write to (or lock) that location. This is accessed with an interlock instruction, for example, the branch on bit set and set interlocked (BBSSI) or the add aligned word interlocked (ADAWI) instructions.

Upon gaining the software lock, a given process can proceed to write any location in the shared block. Read-write coherency will be assured only if the other processes sharing that data observe the protocol of obtaining the software lock before modifying the data structure.

The VAX interlock instructions are implemented using interlock microinstructions. These enable a processor to lock and unlock the memory subsystem. Once locked, this subsystem excludes further attempts to lock it until an unlock has occurred. Thus only one processor or I/O system can lock the memory subsystem at any one time.

When each processor has an internal cache, there is one more mechanism that keeps the two processors coherent. While one processor is performing a write to memory and while the write command is on the NMI bus, the other processor will examine its cache store to see if it contains a copy of that data. If the data is there, it is marked invalid. The next request for this data will then result in a cache miss and a subsequent fetch to memory. This simple approach is possible because the VAX 8800 caches are write-through. Although all writes are seen on the bus, the write buffer packs together consecutive writes within an octaword. Therefore, the number of invalidation cycles performed by a processor will be reduced. When an interlock write is performed, the contents of the write buffer are sent to memory. Thus the interlock mechanism ensures that data coherency will work under all conditions. Figure 6 illustrates the events that achieve coherency in the 8800.

Summary

The general concepts used in the design of the C Box are well known to computer designers. Our goal was to achieve a simple yet high-performance design that avoided unnecessarily complex solutions that did not give comparable increases in performance. The choices made

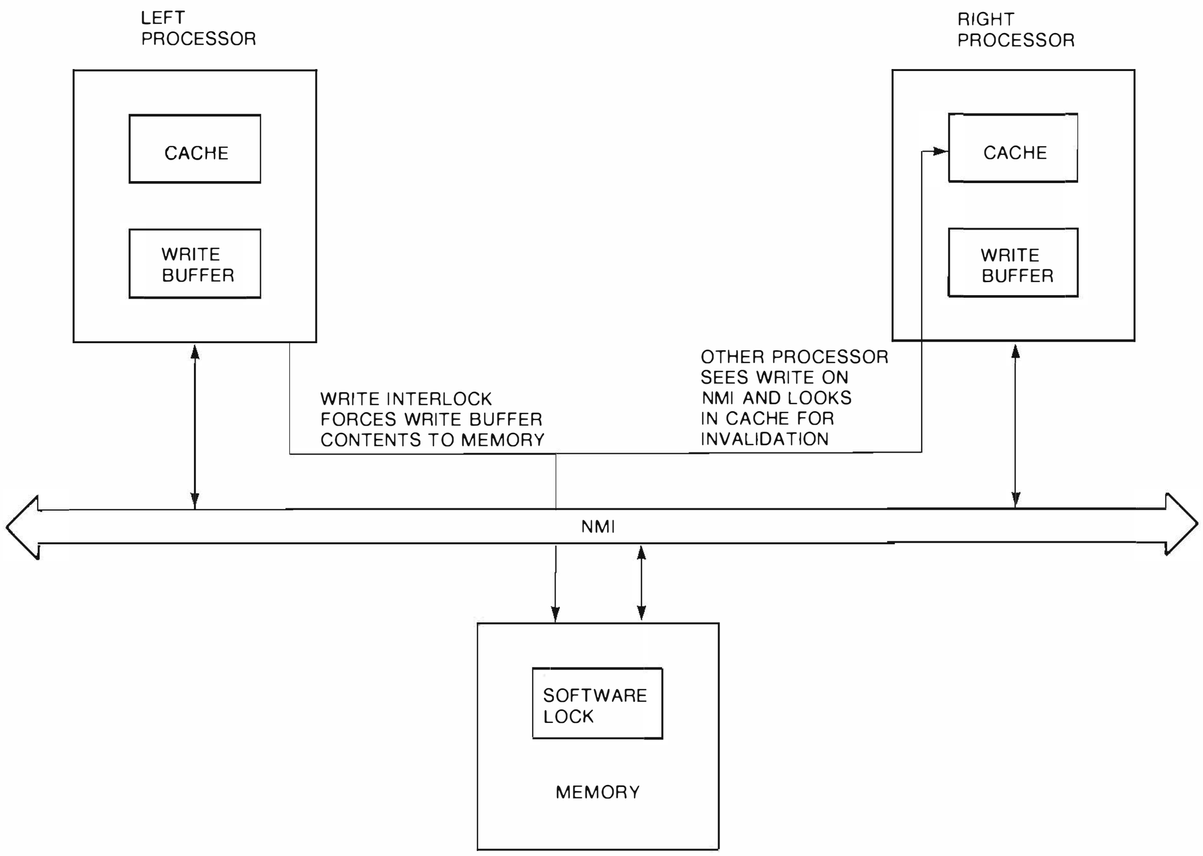

LEFT PROCESSOR
CACHE
WRITE BUFFER
RIGHT PROCESSOR
CACHE
WRITE BUFFER
WRITE INTERLOCK FORCES WRITE BUFFER CONTENTS TO MEMORY
OTHER PROCESSOR SEES WRITE ON NMI AND LOOKS IN CACHE FOR INVALIDATION
NMI
SOFTWARE LOCK
MEMORY

Figure 6 Multiprocessor Coherency

have yielded a design that fully supports the multiprocessor concept. The VAX 8800 system can translate addresses and access data faster than any previous VAX processor.

Acknowledgments

All those who worked on the VAX 8800 system contributed to the thinking that went into the C Box design. Special thanks go to Dave Sager for keeping things going.

References

1. *VAX Architecture Handbook*, (Maynard: Digital Equipment Corporation, Order No. EB-26115-46, 1986): 7-11 to 7-19.

2. A. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, (September 1982): 473–530.

3. S. Mishra, "The VAX 8800 Microarchitecture," *Digital Technical Journal* (February 1987, this issue): 20–33.

4. T. Fossum, J. McElroy, and W. English, "An Overview of the VAX 8600 System," *Digital Technical Journal* (August 1985): 8–23.

5. S. Farnham, M. Harvey, and K. Morse, "VMS Multiprocessing on the VAX 8800 System," *Digital Technical Journal* (February 1987, this issue): 111–119.

Paul J. Natusch
David C. Senerchia
Eugene L. Yu

The Memory System in the VAX 8800 Family

The memory system in the VAX 8800 family can send data at 71MB per second and receive it at 59MB per second. The 8800 and 8700 CPUs can contain up to 128MB of memory, the 8550 and 8500 up to 80MB. Commands, addresses, and data flow between the memory interconnect (NMI bus) and the memory controller, array bus, and array modules. Read, write, and masked-write commands are executed. The designs of the NMI bus and write-through cache affected the memory system design. Although ECL is used in the controller, TTL is used in the array bus. The array modules of 4MB and 16MB contain 256K MOS dynamic RAM chips.

All members of the VAX 8800 family of processors (the 8800, 8700, 8550, and 8500) use the same type of memory system. Since the VAX 8800 system is a multiprocessor, that memory system must connect to both CPUs and both I/O adapters, called the NBIAs. The bus connecting these devices is called the NMI bus, and each connection on the NMI bus is called a nexus. These connections are illustrated in Figure 1, which shows five nexuses: one for each CPU, one for each NBIA, and one for the memory system.

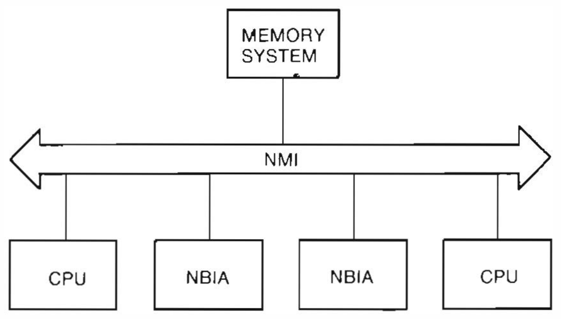


Figure 1 Memory Interconnect Structure

The memory system itself consists of three major parts, as depicted in Figure 2:

- A memory controller based on ECL technology
- A high-speed TTL bus connecting that memory controller to a maximum of eight array modules
- The array modules themselves

The memory system can deliver 71 megabytes (MB) per second of read bandwidth and 59MB per second of write bandwidth.

Since the VAX architecture has a 32-bit format, all datapaths in the memory system must also handle 32 bits. These datapaths are combined by pipelined and parallel operations to produce the read and write bandwidths. The most significant occurrence of parallel operations is two-dimensional interleaving. The first dimension interleaves between longwords (32 bits) of data on a single array module; the second interleaves between octawords (4 longwords) on different array modules. As many as three array modules can be active simultaneously with either a read or a write. There are three cases:

- Each module can do one read.
- One module can do a read while the other two can do as many as four writes.
- Two modules can each do a read while the third can do as many as four writes.

The selection of the array modules can be programmed from the console when the system is powered up. Thus the memory system can support a variety of array module sizes and speeds without the need to modify the hardware in the memory controller. Moreover, the memory controller can address 512MB of physical memory, the limit of the VAX architecture. The 8800 is the first VAX system to be able to address this much physical memory.

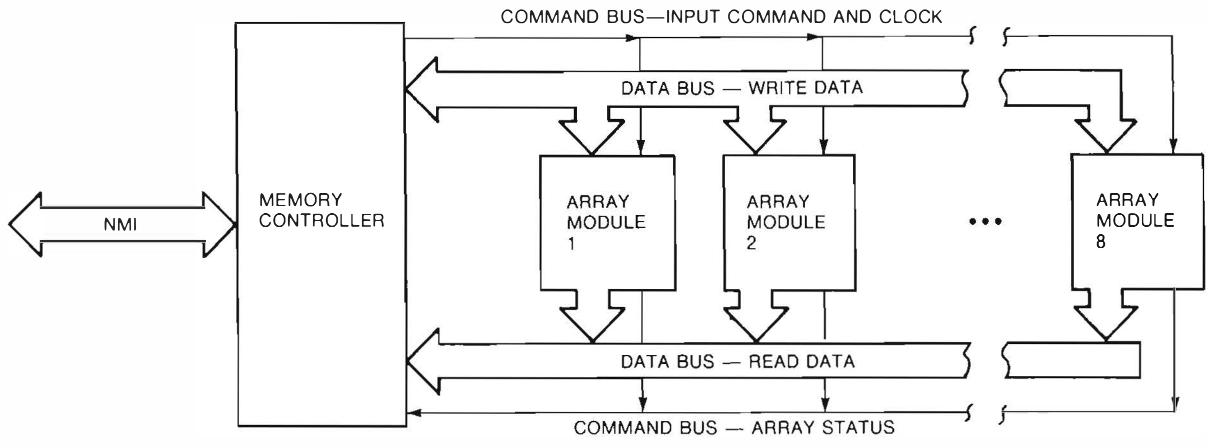


Figure 2 Plan of Memory System

Owing to the limits of the existing technology, however, the initial machine was introduced with 32MB for the 8800 and 8700 systems, and 20MB for the 8500 and 8550 systems. The 32MB configuration consists of eight 4MB modules with 256K MOS dynamic RAMs packaged in DIPs. To increase the density of the machine without using a different semiconductor technology, a 2MB daughter module was developed after the initial announcement. This module uses double-sided surface-mount technology and plastic leadless chip carriers. Eight of these daughter modules are mounted on a mother module to produce a 16MB array module. This new module has increased the machine's memory to 128MB for the 8800 and 8700 systems, and to 80MB for the 8550 and 8500 systems.

Memory System Architecture

As shown in Figures 1 and 2, the memory controller communicates with the CPUs and the NBIAs over the memory interconnect, called the NMI bus. Commands, addresses, and data requests are all first received by the NMI interface and then passed to other sections of the memory controller. Addresses and data are stored in custom multiport RAMs, where eight locations are reserved for addresses and eight for data. The NMI interface encodes command information, passing it to the command-control portion of the memory controller.

Since the memory controller communicates with the NMI bus and the array bus, the NMI protocol has to be changed to that of the array bus. Reads and writes of data fields with various sizes are received by the NMI interface. The NMI bus supports a very robust set of commands. Reads and interlocked reads are supported for longwords (4 bytes), octawords (4 longwords), and hexwords (2 octawords). Masked writes and masked-write unlocks are supported for longwords, quadwords (8 bytes), and octawords. Writes are supported for longwords and octawords.

The read-interlocked and masked-write unlock commands are used to implement VAX instructions in which mutual exclusion is required. For example, the VAX instructions ADAWI, BBCCI, BBSSI, INSQHI, INSQTI, INSQUE, REMQHI, and REMQTI all need these commands. Since an interlocked instruction locks the entire memory system, the interlock bit must reside in the memory controller. This bit restricts the execution of subsequent interlock commands until the lock has been released by a masked-write unlock instruction.

After receiving a memory request from a nexus, the memory controller must transfer that request to the appropriate array module. This transfer is accomplished using the array bus. This bus consists of

- A unidirectional set of command and address lines from the memory controller to the array modules
- Another unidirectional set of data lines from the memory controller to the array modules

- A set of data lines (capable of assuming three states) that can be driven by any one of the array modules and received by the memory controller
- Various status and control lines that communicate in both directions

The array bus has a minimal repertoire of commands, consisting of longword reads, octaword reads, and longword writes, but not hexword reads. Since the NMI supports hexword reads, the memory controller must convert them into two octaword reads and then send them to the array modules. Thus the two octawords of a hexword read can reside on different array modules. That fact increases the memory bandwidth because parallel accesses can be executed. The array bus supports only longword writes; therefore, octaword writes must also be converted. As mentioned earlier, the array bus has one line for commands and addresses and another for data. Therefore, an octaword write, which takes five cycles to transfer on the NMI (one for the command, four for the data), can be transmitted in five cycles on the array bus to an array module. Figure 3 shows the corresponding actions during each cycle on the NMI and on the array bus.

In addition to commands, the memory system must also execute maintenance tasks, including memory refresh, error reporting, and battery backup.

Since physical memory is implemented with MOS dynamic RAMs, every array row must be refreshed once every 4 milliseconds. This function can be done by refreshing one row every 14 microseconds. To facilitate this activity, the memory controller sends signals to each array module from a 14-microsecond oscillator. Upon receiving a refresh signal, an array module will handle the refresh arbitration and execute the operation.

Occasionally, a bit will be lost due to either alpha particles or a device failure. In that case the memory controller must handle those errors and other types in a graceful manner. To do that, the memory system uses a 7-bit modified hamming code to generate the ECC, which allows all single-bit errors to be corrected and all double-bit errors to be detected. After correcting each error the memory system logs the error's physical page address and the bit. The memory system then interrupts the CPU to call an error service routine, which logs in a VMS file the necessary information to isolate the failure. The memory system can also interrupt the CPU to handle internal parity errors and interlocked time-outs. An interlocked time-out happens when a nexus executes a read interlock but never issues a masked-write unlock. The system software can enable or disable these interrupts.

Battery backup, standard equipment on both the 8800 and 8700 systems, can power the refresh operation when the system is down. That power allows the memory system to continue to refresh the RAMs so that data will not be lost. Note that the entire system is not backed up;

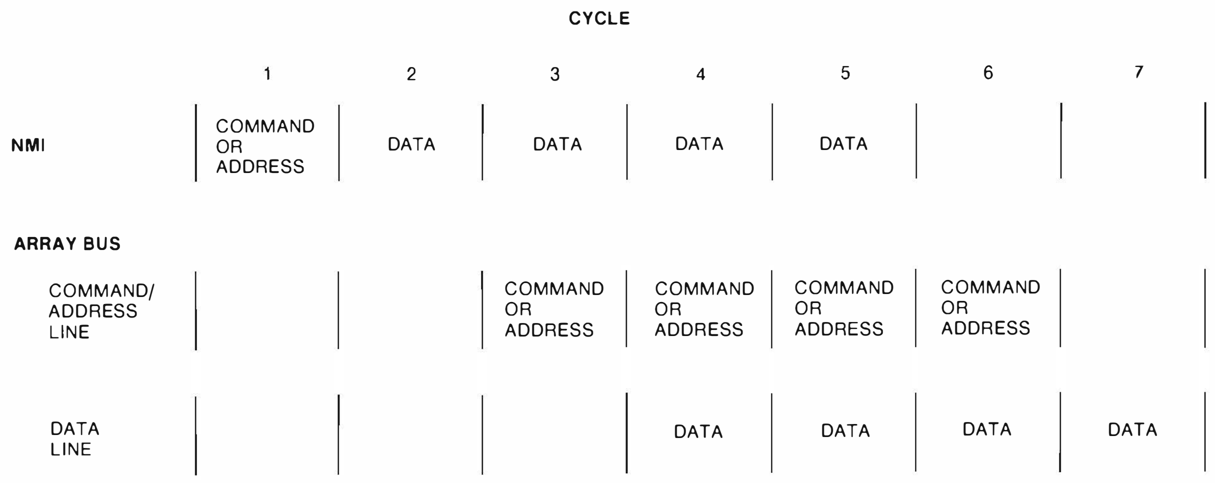


Figure 3 Cycles on NMI Bus and Array Bus

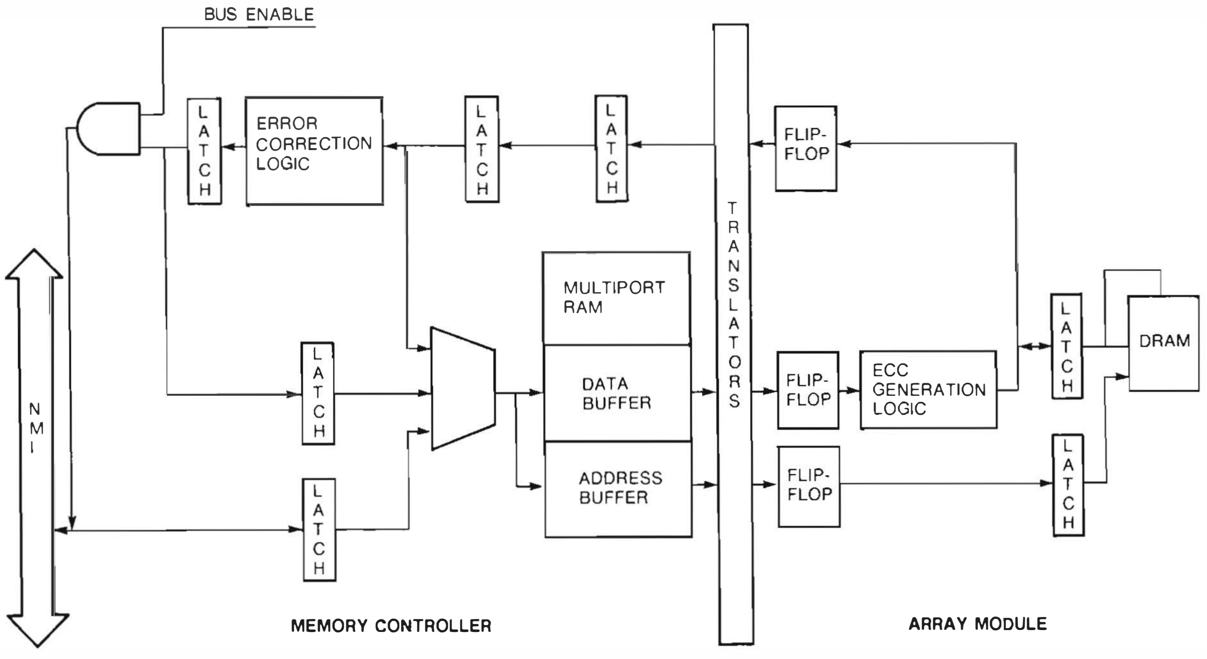


Figure 4 Datapaths in Memory Controller and Array Modules

therefore, all components must be in quiescent states before the memory system enters battery mode. Upon sensing that power is eroding, the 8800 will write all its data to the memory system. The memory controller will then complete all commands and send signals to the array modules informing them to enter battery mode. In this mode only five MSI chips on the memory controller and approximately half the control logic on the array module will be active.

Command Execution

The execution of any command received by the memory system is a joint effort between the memory controller and the array modules. Figure 4 depicts the datapath in each memory component. After a nexus places a command on the NMI bus, the interface in the memory controller ascertains if the command is a valid memory reference and, if so, decodes it. The interface then places the command in a queue of commands waiting to be executed.

Since one array module can execute multiple write commands simultaneously, and since multiple array modules can also execute commands, the memory controller must maintain the status of the array modules. The status control logic to monitor activity must "remember" which portions of which arrays are "busy." This status control logic can best be described by showing how the three basic operations, writes, reads, and masked writes, are executed.

Write Commands

For a write command, the control portion of the memory controller performs only three actions: it determines the capability of the array module to accept the command, it sends the command, and it waits for the array module to signal its readiness to receive another command.

The write datapath is that portion of the logic responsible for the flow of data from the NMI bus to the array modules. This path comprises both electrical interconnects (buses and cables) and a considerable amount of logic. The major storage element for the datapath is a 9-bit by 32-location custom multiport RAM (MPR) with two ports for reads and two for writes. Data received from the NMI bus is placed in the next available location of the MPR. Upon determining that the required array module is available, the control logic sends the data from the MPR to that array module over the array bus. Each array module holds the data until it is strobed into the dynamic RAMs

(DRAMs). The array module can load four longwords of data with their associated ECC bits on four consecutive cycles.

Some writes are called masked because there is a 4-bit byte mask associated with each data word. The byte mask informs the memory system as to which bytes are to be written. The memory system executes this command by first doing a read and correcting any single-bit errors that may exist. It then merges the memory data with the data received from the NMI bus, and finally does a write command. This sequence easily allows the implementation of longword and octaword masked writes. Masked writes for quadwords (8 bytes) are executed by performing an octaword masked write in which the data of two of the longwords remains unchanged.

Read Commands

For read commands, the memory controller performs four actions: it determines if the selected array module is ready to accept the read, it sends the command, it waits for a data-ready response, and it transfers the data from the array module. Imbedded in the command field of the read are address bits that select the longword of the octaword that is required first. This action allows wrapped reads to be implemented. (Wrapped reads are described later in the section "Impact of the Cache.")

The read datapath originates at the DRAM, which sends the requested data. As in the case of write commands, each array module stores an octaword of read data. Once the data has been loaded into the latches, the array module signals to the memory controller that the data is ready. As mentioned earlier, the read datapath between the array module and the memory controller is tristatable. Therefore, the memory controller must ensure that only one array module at a time drives this datapath. Once the data has been requested by the memory controller, the array module must send the longwords sequentially, beginning with the starting address that was sent with the command. This action allows the memory controller to request any one of the four longwords as the first to be read. The array-module portion of the read datapath can transfer one longword of data during every cycle.

The error-correction logic in the memory controller receives each longword of data plus the seven ECC bits. This logic detects single- and double-bit errors, but only single-bit errors can be corrected. A significant feature of this process is that error detection and correction is performed as the read data is pipelined through the memory controller. Thus no additional cycles are needed to correct read data.

Masked-write Commands

The execution of a masked write involves both a read and a write sequence. The memory controller executes a masked-write command by first issuing a read to the selected array module. Assuming that there were no memory errors, the data returned is sent to the MPR, where the bytes are merged with those sent to the memory controller over the NMI bus. The memory controller must ensure that no commands to the same array come between the read and write portions of a masked write. After all the bytes have been merged into the data buffer, the memory controller will write the data to the array module. The array module then generates new ECC data, adds it to the other data, and strobes the composite data into the DRAMs.

If a single-bit error is detected, the process is quite similar to the one with no errors, except that the data must be corrected. Since corrected data and NMI traffic both share the same datapath on the memory controller, the NMI interface must be free to correct errors found during masked writes. This freedom is ensured by asserting a signal that stops all activity on the NMI bus. Once activity has stopped, the data can be routed through the NMI interface, corrected, and then merged with the NMI data in the data buffer. The process then continues as it would have if there were no errors.

If a double-bit error is detected, the process is similar to the case in which no error occurred, except that the write is prevented from happening. When the array location is read the second time, the double-bit error will still be present, thus alerting the system that the data is unusable.

Memory Address Path

The memory controller continuously latches all addresses from the NMI bus. Once an address is latched, the memory controller must verify it as a valid memory address. That verification is done by comparing the address to valid addresses of both the control status registers (CSRs) and physical memory.

The CSR addresses are hardwired into the NMI interface logic; therefore, only a simple compare of the addresses is required. The compare for a valid memory address requires a reference to a "decode" RAM. This RAM is loaded by console software when the system is powered up and is used to configure memory. Loading the RAM from software allows the memory controller to support several different sizes of array modules without modifying any hardware.

Once the address has been verified as being valid, it is placed in one of eight storage locations allocated to address buffering in the MPR. The address remains in that buffer until its command is sent to an array module.

Even though eight locations are allocated to address buffering, only seven of them can be used for temporary storage. One location is reserved for the error's page address, a pointer to a physical page of memory containing an error. Since the location of the error page-address buffer is not fixed, the control logic for the address-buffer control must look ahead and not allow a new address to overwrite that error page address.

The control of the address buffer is further complicated by masked writes and error logging. Since a masked write is implemented as a read followed by a write, the address in the buffer cannot be overwritten until the write has completed. A similar situation exists for error logging on read transactions. Since an error is not detected until the read has completed, the address cannot be overwritten until the data has been checked.

Design Requirements of the VAX 8800 System

Impact of the NMI Bus

As stated earlier, the VAX 8800 memory system interfaces with the CPUs and I/O systems through a synchronous bus called the NMI bus. This bus is highly efficient and operates in a pended fashion similar to the synchronous backplane interconnect (SBI bus) in the VAX-11/780 processor. The NMI bus allows several transfers to be in progress simultaneously.

There are four nexuses in the 8800 system that can require memory: the two CPUs, and the two NBIAs. Each nexus is allowed to have two commands outstanding at any time. The protocol supports this arrangement by allocating two codes in a 4-bit ID field to each nexus.

The CPUs use one of their references for program data, called the d-stream, and the other for instructions, called the i-stream. The CPUs always request a hexword of data; the NBIAs may request either longwords or octawords. Thus there can be as many as eight simultaneous requesters of memory data. These simultaneous events require that the memory system buffer several commands while executing. In the 8800 implementation, the memory system can access three array modules in parallel and store two commands.

Moreover, since the memory system can accept multiple read commands, it must store the identification of the requester and the length of the transaction. The NMI interface does the actual storing and returns the identification with the correct data. This action is possible because all commands are processed in sequence; therefore, the read returned first is the one stored the longest. However, hexword reads are returned to the NMI interface as two separate octaword reads; therefore, that interface must ensure that both octawords have been returned before discarding the identification.

To prevent a deadlock condition, the memory system is given the highest priority during arbitration. This priority guarantees that the memory system will be able to return data to a requester. When full, the memory system notifies any potential requesters that it cannot process any more commands and to try again later, thus preventing the memory system from overfilling.

Impact of the Cache

The design of the cache affected the design of the memory system. The write-through design of the cache guarantees there will be a large number of longword writes directed at memory.[1] A write buffer was installed to bundle a series of longword writes into octaword writes; however, the write buffer is only effective if multiple longwords are written in the same octaword.

Extra logic is always required to increase performance. The extra write bandwidth for this memory system, however, required more logic than what would have been required to implement extra read bandwidth. The added complexity was needed to facilitate interleaving on longword boundaries for write operations.

When the 8800 project was first initiated, the goal of the memory system was to maximize read bandwidth, thus producing a relatively simple array-module design. In that design, any operation, regardless of its size, kept an entire array module busy until the operation completed. The control logic on the array module was simple and required a reasonable amount of board space and power. When the design changed to the write-through concept, however, higher write bandwidth was required. Therefore, the control logic in each array module had to be replicated for each bank (longword) of memory to allow independent write operations. This replication permitted four longwords to be written on four consecutive cycles to the same array module.

This increase in design complexity was not limited to the array module. In the initial design, when maximum read bandwidth was critical, the memory control logic was relatively simple. It had only to track the state of an array module as being busy or not. However, with the interleaving capability required for the increased write bandwidth, the memory control logic now has to track simultaneously the status of as many as eight write operations in progress on two array modules.

Although maximizing the longword write bandwidth was important, minimizing the read latency to the first longword required was critical. Wrapped reads were implemented to reduce this latency. A wrapped read is a hexword or octaword command that requests a specific longword to be returned first, with other longwords in that block to follow in "wrapped" fashion.

Other Design Trade-offs and Options

As in all design processes, we considered many trade-offs and options before committing to a particular design architecture. One area with several alternatives was the interconnect between the memory controller and the array modules. The array modules and the controller reside in physically separate backplanes interconnected by a cable. We had to decide whether to make this interconnect with ECL or TTL.

The overall project goal was to make the 8800 an all-ECL machine. Therefore, our first choice for this interconnect was ECL, which provides enhanced signal integrity, reduced skews, and overall speed advantages over TTL. As the system and memory design progressed, however, some real problems arose that altered our opinion. The first problem became apparent as the array-module design coalesced enough to allow some accurate power estimates to be made. We found that, with an ECL bus, the array module would require -5.2 V in excess of its allocation. The next problem surfaced in response to an architectural requirement that the memory system function with less than eight array modules and, preferably, without load cards. This requirement made it difficult to implement a termination scheme for an ECL interconnect.

With these problems in mind, we investigated a TTL interconnect, which clearly offered some design challenges, the least of which were speed and skew. Using the SPICE simulator, we constructed an accurate model to verify that a TTL electrical interconnect could indeed meet our signal integrity, speed, and skew requirements.[2] While the simulation results showed that a TTL interconnect could work, the associated skews certainly increased the complexity of the memory design. While alleviating the problems of limited -5.2 V power on the array module and the termination of varied loading, this TTL scheme required ECL-to-TTL translators in the memory controller to drive the array bus. We finally decided to accept the added complexity and use TTL for the interconnect. The sole exception was the clocks, which were differential ECL, received and translated on the array module.

There were logical trade-offs as well as electrical ones. The original specification for the NMI did not support quadword masked writes. They were added after the implementation of the memory system had progressed considerably. Since the array bus supported only longword and octaword reads, there were three options to support this change:

- The first was to change the array bus protocol, the command generator on the memory controller, and the array module.
- The second was to execute the command by performing two longword masked writes. This option would take almost twice as long as a quadword masked write if implemented like the first option, yet still require changes to the command generator in the memory controller.
- The third was to execute an octaword masked write in which the data of two of the longwords remains unchanged.

Since the design was well advanced, we chose the last method to ease the problems of implementation; this decision actually has little impact on system performance. The logic to accomplish this addition already existed on the array module. Only small changes were required to the command generator of the memory controller and the datapath control. In practice, the frequency of quadword masked writes is extremely low since they are executed only by the NBIAs.

Technology Description

A number of different module and component technologies were used for the memory controller, backplane, and two array modules.

Memory Controller

The memory controller is a 9-layer, controlled-impedance, extended hex module (15 inches by 11 inches). The lay-up consists of 6 routing layers, 2 power layers (− 5.2 V and − 2 V), and a ground plane. Since there is a minimal amount of TTL, both the +5 V power and the +5 V battery are run on the surface with 50-mil etch. With the mixed technology on the module, we took special care to keep the TTL signals properly spaced from the ECL signals to avoid signal integrity problems.

The logic on this module is implemented using nine unique macrocell-array designs from Motorola, Inc., and one custom ECL multiported RAM. There are 16 custom and semicustom devices on the module. It also contains some 10KH MSI logic, some ECL-to-TTL converters, and some CMOS logic used for operating with battery backup.

Array Module Backplane

The array module backplane in the VAX 8800 and 8700 CPUs is a 12-layer, 8-slot pressed-pin backplane. The one in the VAX 8550 and 8500 CPUs is a 5-slot backplane. Since a TTL bus was chosen to communicate between the memory controller and the array modules, a good termination strategy had to be developed. Using the SPICE simulator, we evolved the termination strategies shown in Figure 5.

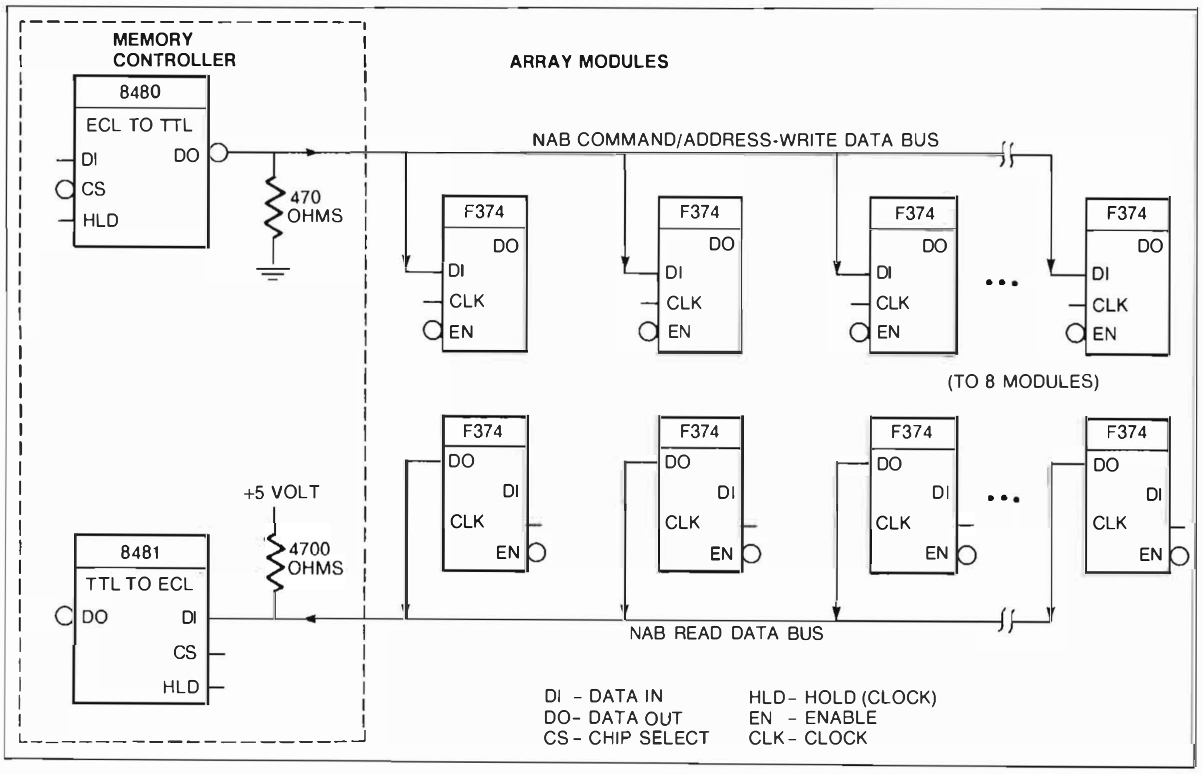


Figure 5 Termination Strategies in Memory Controller and Array Modules

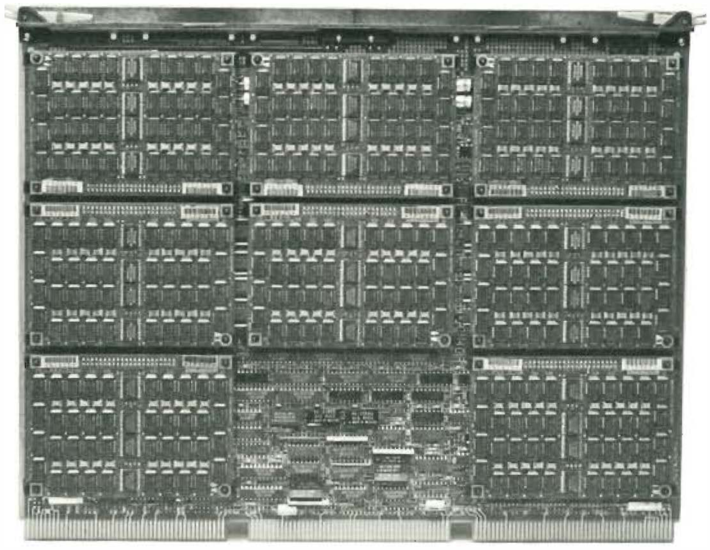


Figure 6 Sixteen Megabyte Array Module

Four Megabyte Array Module

The 4MB array module was designed using an 8-layer, controlled-impedance, printed circuit board. The lay-up consists of 4 routing layers, 2 power layers, and 2 ground layers. To support battery backup, the module has separate power planes for +5 V power and the +5 V battery. Since only a limited amount of −5.2 V and −2 V power is needed, these voltages share space on the other power planes. To eliminate discontinuities that could cause unwanted reflections, we ensured that signals did not cross the power-plane splits by surrounding the power planes with solid ground planes.

Approximately half of the logic technology on the array module consists MOS dynamic RAMS; the other half is FAST MSI logic. The clock system is implemented in ECL to minimize the skew.

Sixteen Megabyte Array Module

A 16MB array module was developed to increase the available memory to 128MB for the 8800 and 8700 systems and 80MB for the 8550 and 8500 systems. This array module consists of an 8-layer mother board (similar to the 4MB module) and eight 2MB surface-mounted daughter boards. The 16MB array module is pictured in Figure 6.

Summary

The VAX 8800 memory system was designed to provide 71 MB per second of read bandwidth and 59 MB per second of write bandwidth to the multiprocessor system. The system architecture, processor performance needs, and high I/O activity combined to make a high-performance memory a requirement.

Since the 8800 contains ECL components, the memory system has to provide a high-speed path between the ECL logic in the CPUs and the high-density dynamic RAMs used for main storage. Although the memory system does not play a direct role in the execution of a VAX instruction, its performance has to match closely that of the multiprocessor system. If the memory system were under designed, the processors would stall frequently, thus reducing their usable performance. If the memory system were over designed, it would contain extra complexity, with the attendant extra cost, that could not be used by the system. Thus the memory strategy played an important role in the price/performance trade-offs that had to be made.

Acknowledgments

Although done by a small group of engineers, the design of the memory system was greatly influenced by the efforts of many people from the Electronic Storage Development Group and the Advanced VAX Engineering Group. We would especially like to acknowledge the creativity, leadership, and energy level of the late John Henry, Jr.

References

1. J. Fu, J. Keller, and K. Haduch, "Aspects of the VAX 8800 C Box Design," *Digital Technical Journal* (February 1987, this issue: 41–51.

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

John H.P. Zurawski
Kathleen L. Pratt
Tracey L. Jones

Floating Point in the VAX 8800 Family

The processors in the VAX 8800 family were designed with particular emphasis on cost-effectiveness. These CPUs do not contain separate floating point accelerators. Their performance is not compromised, however, especially for the double-precision instructions. High performance is achieved, in part, by a custom ECL multiplier and divider unit and by specific hardware for exponent manipulation and normalization. The main advantages of this integrated approach are less hardware to replicate and a tightly coupled interface to each CPU, thus less time is wasted fetching the operands. Microcode branch problems are minimized by using a prediction strategy and extensive hardware assistance.

Unlike other VAX families, the processors in the VAX 8800 family do not contain separate floating point accelerators (FPAs). Instead, their FPA is integrated into each processor's main datapath. Therefore, no distinction is made between instructions that are executed in the FPA and those that are not: the hardware is available to be used for all functions. For example, the extended arithmetic logic unit (XALU) is also used as a counter for the move character instruction (MOVC). This usage differs from that in the VAX 8600 and VAX-11/780 systems, where the XALU is used only for floating point instructions. Furthermore, all the floating point instructions, from the most complicated (POLY and EMOD) to the simplest (MOVF), have access to the FPA hardware.

There are a number of advantages to this approach. First, logic is not duplicated; only one arithmetic logic unit (ALU) and one shifter unit is shared between the floating point and the normal arithmetic. Second, the design is tightly integrated with the rest of the computer; there is no overhead involved in starting the floating point computation.

Clearly, since all other VAX families use FPAs, there are also disadvantages with our approach. Shared logic is more complex than specialized logic. Performance may also suffer since the design cannot be optimized toward one class of problem. Those disadvantages can be overcome, however, as we shall relate in this paper. The problem of optimization was ameliorated by providing dedicated hardware for the main operations of multiplication and addition. A custom multiplier and divider chip is provided together with exponent manipulation logic and a shifter unit optimized for floating point. These logic elements handle those floating point operations that take the longest times to execute.

The floating point logic resides in the execution unit, the E Box, of the VAX 8800 CPU. That logic is controlled by microcode in the instruction unit, the I Box.[1]

VAX Formats and Instructions

The VAX architecture supports four floating point formats: F, D, G, and H. These formats are discussed at length in references 2 and 3. The F format is 32 bits wide, the D and G formats are both 64 bits wide, and the H format is 128 bits wide. Although the D and G formats have the same width, the exponent field is larger in the G format, and its fractional field is commensurately smaller. This format allows a larger range but with slightly lower precision. The fractions are always normalized and the leading bit — the hidden bit — is not stored.

E Box Operation

Physically, floating point operations are performed on three modules: two slice modules and a shifter module. The slice modules contain the cache, the main ALU, and a register file. The shifter module contains the custom multiplier, the shifter unit, the exponent manipulation logic (the two ALUs), and the priority encoder. Figure 1 shows this partitioning. To a large extent, the shifter module strongly resembles an FPA but without the ALU and register file.

The source operands are fetched from either the 64 kilobyte (KB) cache or a general-purpose register (GPR). The operands are sent on the A and B ports to the ALU on the slice modules and to the shifter module. All the components on the shifter module are driven in parallel by the A and B ports.

From Figure 1 it is clear that the datapath is highly parallel; the shifter, XALU, multiplier, and ALU can all operate simultaneously. This parallelism is used extensively to gain performance and to save cost. For example, in multiplication operations, the XALU determines the exponent of the result, the multiplier multiplies, and the shifter absorbs the low-order bytes of the product that are discarded each cycle by the multiplier.

The main problem with designing an integrated FPA is that the VAX formats for integer and floating point numbers must all be handled by the same shared units. Figure 2 shows the different bit orderings for two VAX formats, the F floating point and the integer. In the integer format, the bit ordering is from right to left. In the F format, the mantissa begins at bit 16 and increases in significance to bit 31, then continues from bits 0 through 6. The remaining bit positions are used to hold the exponent and the sign.

This requirement for shared handling complicates the carry path of the ALU. The carries out of the 16-bit word boundaries have to be switched into the appropriate places, as shown in Figure 3. The problem with shifting is similar to the carry problem, except that now the carry path of Figure 3 represents the flow of the shifted bits.

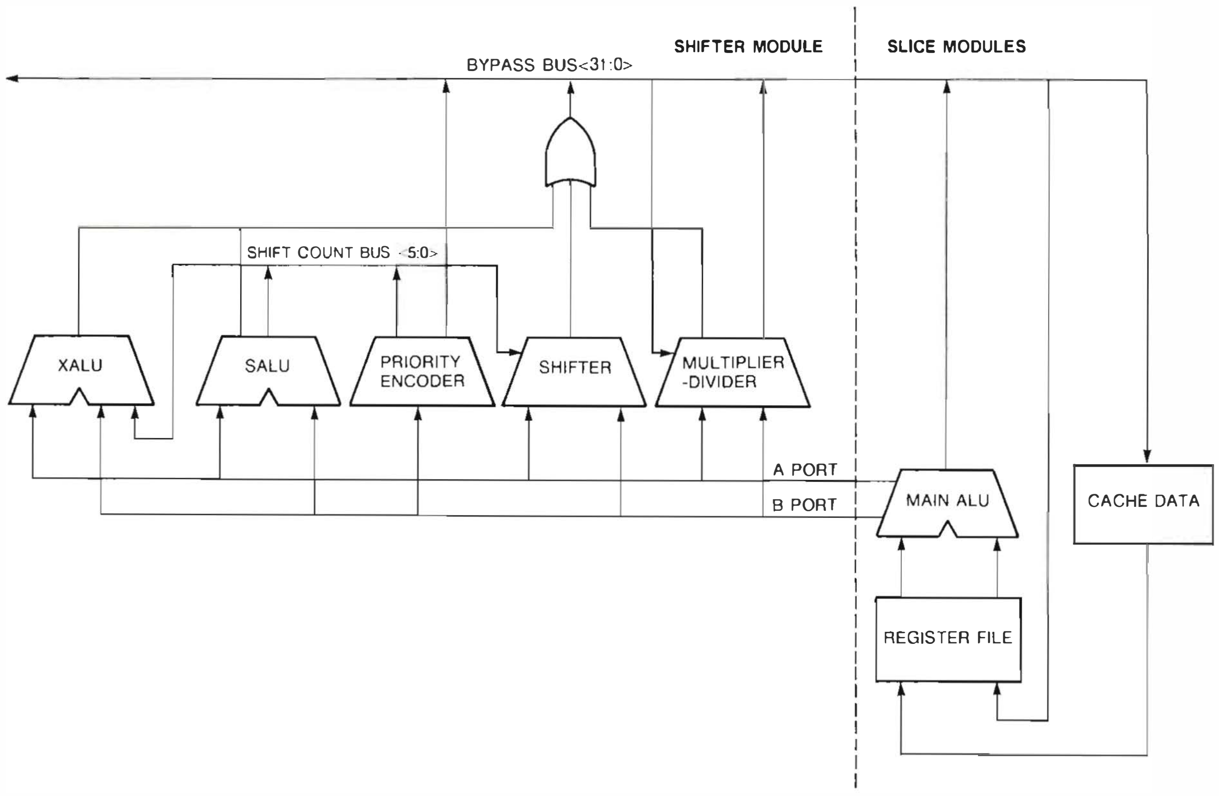


Figure 1 Block Diagram of the E Box

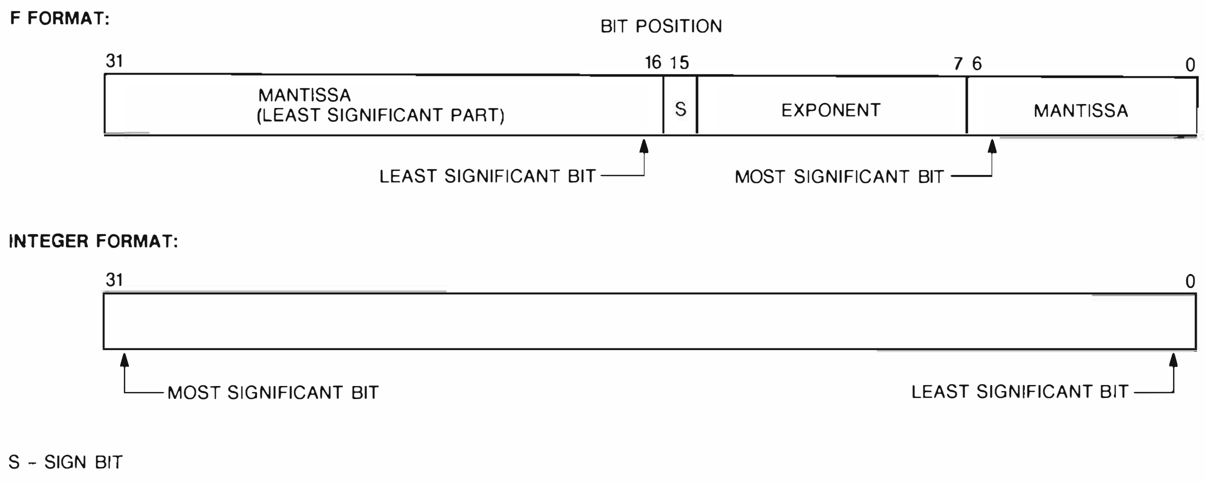


Figure 2 Two VAX Formats

The ALU and the shifter unit are both designed to handle all integer and floating point formats. The multiplier expects operands to come only in a floating point format. Therefore, for integer multiplications, the data must first be converted into a pseudo-floating point format by swapping the places of 16-bit words within the integer format. This operation is performed by the shifter unit.

Table 1 gives the execution times for the most common floating point instructions. These times include the overhead for fetching the operands. The VAX 8800 processor is designed so that there is little, if any, difference in performance between register and memory operands. The execution times vary from 2.25 to over 5 times the performance of the VAX-11/780 CPU with an FPA for the F and D formats. For multiplies, one 8800 CPU is 2.5 times faster in F format and 4.8 times faster in D format; divides are 3.0 times faster. The gain is even more substantial for the G and H formats since they are not accelerated on the 11/780.

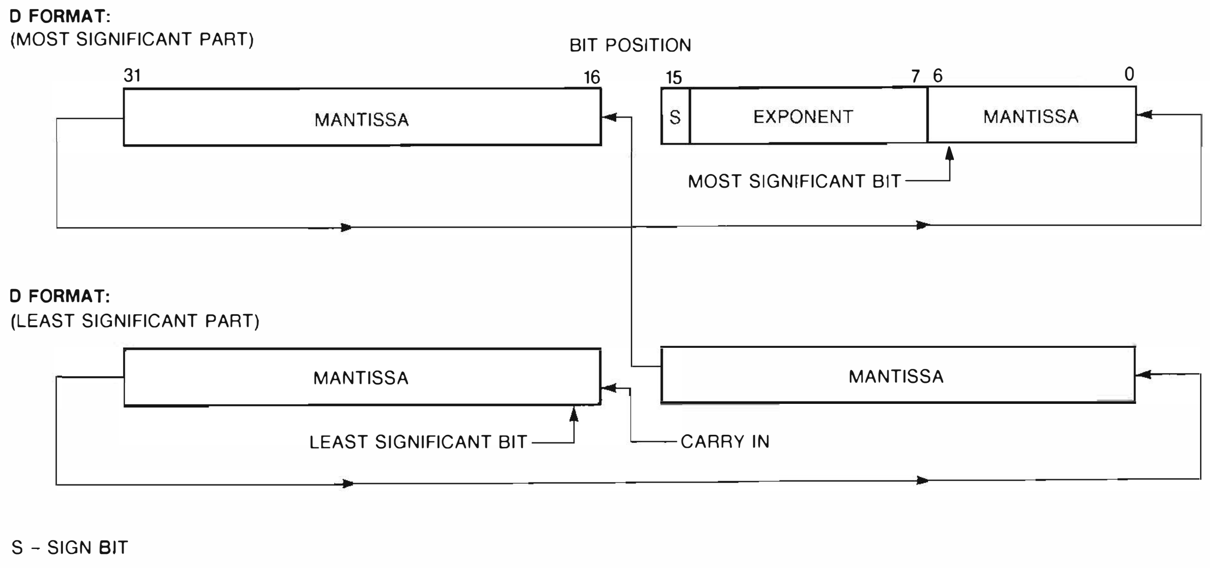


Figure 3 Floating Point Carry for D Format

Table 1 Execution Times

Instruction Register to Register	Execution Time (Nanoseconds)			
	F	D	G	H
ADD	315	495	540	3314
MUL	450	675	842	6306
DIV	1607	3197	3107	21649

In the 8800 the D format is slightly faster than the G format with its longer opcode, which requires an extra cycle in the decoder. The single-precision F format executes the fastest, and the larger 128-bit H format executes the slowest. However, the H format is intended as a backup for intermediate calculations in the D and G formats. Used thus, the H format ensures that the final calculation result has sufficient precision and avoids overflow or underflow problems. Little hardware assistance is provided for the H format; it is driven mostly by microcode.

Technology

Component technology used in the VAX 8800 processor is an enhanced version of the macrocell array (MCA) used in the VAX 8600 CPU.[2] This technology provides about 1,200 gate equivalents with a typical gate speed of 1 nanosecond (ns). MCAs utilize emitter-coupled logic (ECL) in a 72-pin package that is 1 square inch with a maximum power dissipation of 5.5 watts. The GPR and the multiplier are made with custom technology, which uses the same package as the MCA but contains a more advanced process. Around 1,800 gate equivalents are provided, and the gate speed is 50 percent faster than the MCA. This higher performance is achieved by using the following features:

- Smaller transistors and metal-oxide-walled resistors
- Current mode logic instead of the slower ECL
- Four-level logic instead of the two-level logic of the MCA

At 300 by 260 mils, the size of the custom chip is larger than the dimensions of 221 by 252 mils for the MCA.

The shifter module contains 12 MCAs and 8 custom multiplier parts. Some 10KH parts are used for clock distribution and for driving the bidirectional bypass bus.

Arithmetic Algorithm Processing

Addition and Subtraction

For an addition operation, the 32-bit words containing the exponents are sent to the main ALU. There they are passed to the A and B ports, which feed the shifter module. These ports drive all the gate arrays in parallel.

The exponents are then loaded into the XALU and the shift-amount ALU (SALU), which computes the alignment shift amount sent to the shifter. The SALU also generates some 20 branch conditions for the microcode. These conditions indicate the size of the alignment shift and whether any source operand is zero or a reserved operand. They also help to optimize the microcode flow.

The XALU, which selects the larger exponent and saves it for later use, has a 12-bit datapath and a register to hold the exponent. The size of this datapath is sufficient for the F, D, and G formats plus a guard bit for overflow or underflow detection. An ALU is provided to perform arithmetic operations on the exponent. The SALU, with an 11-bit datapath, subtracts the exponents to determine the alignment shift amount, which is always positive. The sign manipulation logic also resides in the SALU.

Next, the fractional part of the smaller operand is aligned by the shifter. This operation involves either one CPU cycle for F format operands or two CPU cycles for the D and G formats. The shifter unit shifts in the floating point format and can do a full 64-bit shift. The logic that determines the round bits is related to the alignment shift operation but is physically located in the priority encoder gate array. This gate array also contains some of the shifter functionality.

Nine gate arrays are used for the shifter unit. Of those, eight make up the datapath, the ninth is the control device. The shifter can accept either a 64-bit operand on the A and B ports or a 32-bit operand on either port. The shifter generates a 32-bit result that can be either the high-order or the low-order part of the answer. The shifter datapath gate arrays are identical: each effectively constitutes a byte slice of the design and performs a bit shift of up to seven places. Byte shifting is then performed by sending the correct shifter output to the correct byte position. This operation is facilitated by having all the outputs wired to the OR gates at all possible byte positions and by enabling the correct output.

The shifter performs floating point, integer, and logical shifts, as well as a number of miscellaneous functions. These include converts from decimal-format data into integer format and vice versa. The masking of the exponent field and the insertion of the hidden bit are also done by the shifter.

After the alignment shift, the output of the shifter is directed to the main ALU on the bypass bus. There, the output is added to or subtracted from the fraction of the larger operand. The output of the ALU operation is now ready to be normalized in the shifter. In most cases a small normalize shift of at most one bit position left or right will be sufficient. The specialized hardware in the shifter handles this case and then rounds the result. Should a larger shift be required, then microcode will first direct the ALU result to the priority encoder gate array. There, the position of the leading 1 is found and used to determine the normalize amount for the subsequent cycle.

The rounding operation in the VAX 8800 CPU is unusual in that it is limited to the low-order eight bits. Therefore, a small 8-bit adder can be used for this operation. This adder is both faster and cheaper than the usual method of using a full 64-bit adder. The 8-bit adder is also sufficient to calculate the correct answer in over 99.5 percent of the addition operations. Should a carry-out be generated by this 8-bit rounding add, then clearly the result created is incorrect. In that case the computer is trapped and microcode invoked to correct the result.

Multiplication

As mentioned earlier, the 8800 contains a high-performance, custom-designed multiplier and divider unit. A number of factors impelled us to use such a unit. First, multiplication is a very frequent operation that is used extensively in matrix manipulation. For example, in the LINPACK benchmark, the time-critical routine contains an even mix of addition and multiplication operations.[4]

Second, it was not possible to succumb to the temptation of using the main ALU to provide the division operation. This desire was natural since division is an infrequent operation, and the use of an ALU in a repeated subtract and shift mode was appealing. For example, the VAX 8600 uses the ALU for just that purpose. In the 8800 the main ALU also computes the virtual address. Since this datapath is very time-critical (in the 8800 as well as in most other computer designs), it cannot be allowed to go any slower. Including an extra path to accommodate division would have slowed down this critical path by around 5 ns, resulting in a 10 percent performance degradation for all operations.

Moreover, the available space for the multiplier and divider unit was limited since floating point operations are integrated with the rest of the machine. Approximately one-third of a module (12 inches by 16 inches) was available. In contrast, the VAX 8650 CPU dedicates a full module to multiplication.

The custom design of the multiplier and divider unit is basically a byte slice of a large word-sized multiplier and divider unit. The multiplier handles 8 bits per cycle, the divider handles 1 bit. Figure 4 shows the complete 56-bit by 8-bit multiplier with its eight byte-slice custom chips. Eight chips are used to form the required word size of 64 bits (56 data bits plus 8 guard bits). This arrangement is sufficient to handle F, D, and G format operations. H format operations are performed by partitioning the problem into many smaller 56-bit multiplications under microcode control.

The multiplicand is loaded into the MD latch after passing through the mask logic, which clears the sign and the exponent field and inserts the hidden bit. The PR latch and the PRGB are cleared at the start of the multiply. The PRGB contains the guard bits for the PR latch. At the end of a multiply, this latch will hold the bits required for a possible normalization shift and also for a rounding operation. The least significant eight bits of the multiplier are loaded into the multiplier latch. The first multiply cycle is now ready to be performed.

A 56-bit by 8-bit multiplication is performed between the contents of the MD and multiplier latches. The result is then added to the contents of the PR latch (which is initially zero) and then written back into it with a right shift of 8 bits. The PR latch is thus an accumulating latch and

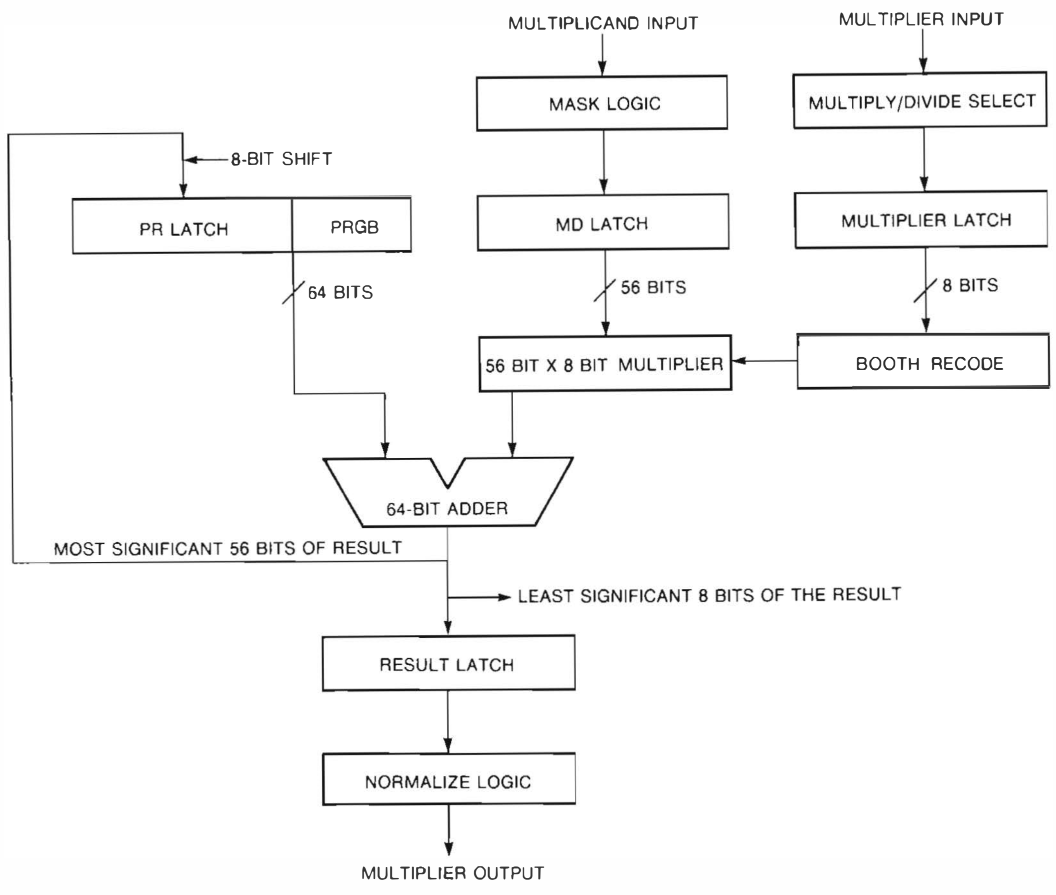


Figure 4 Multiplier and Divider Unit

contains the 64-bit partial product of each multiplication operation. The next 8 bits of the multiplier are loaded into the multiplier latch, ready for the next cycle. This cycling continues until the multiplicand has been multiplied by all the multiplier bytes. This algorithm is similar to the one used in the VAX 8650 scheme, except that that processor has a narrower datapath of 32 bits.

Notice that the least significant byte of the partial product is discarded after each cycle and absorbed by the shifter unit. These bytes are required only for the H format multiply.

Once completed, the result is sent out through the result latch, then normalized and rounded. The rounding carry is only propagated into the least significant byte of the result. This procedure uses less logic since only an 8-bit instead of a 64-bit incrementer is required. The 8-bit incrementer will be sufficient for most multiplies. Should a greater increment be required, then the multiplier will trap the rest of the machine, and the correction will be performed by the main ALU. This scheme is similar to the one used for addition.

The provision of a 64-bit adder inside the main multiply path is unusual in a high-performance machine. High-speed multiplier designs typically use carry-save adders, which do not propagate the carry signal but save them so they can be absorbed by the subsequent cycle. This form of adder is indeed used in the custom multiplier to perform the 56-bit by 8-bit multiply function illustrated in Figure 4. However, the 8800 also uses a full 64-bit adder for the following reasons:

- A 64-bit adder has to be provided somewhere to propagate the carries from the carry-save adders.

- With the 45-ns cycle time, the 64-bit adder fits in the main datapath. A faster clock for the multiplier would have complicated the clock distribution and been difficult to generate with low skew.
- A full adder in the datapath allows the use of a simple nonrestoring division algorithm.

The multiplier and divider chip contains a 12-bit by 8-bit multiplier, two 8-bit adders, six latches with a total size of 72 bits, as well as the rounding, normalizing, and control logic. A comparable MCA design would require between three and four of these elements.

Alternative Designs for the Multiplier

An MCA design was certainly possible and could have been made to fit in the specified space. The performance of such a design, however, would not be as good as the custom design for multiplication but comparable for division. An MCA design would be 1.7 times better than an 11/780 with an FPA for a multiply in F format, whereas the custom logic chosen is 2.5 times better. The performance would be 2.5 times better for the D format, whereas the custom design is 4.8 times better.

Another alternative was to use a commercially available multiplier. That was tempting because such a product has the advantage of being readily available and tested. Using it would have circumvented the high risk of a custom design. However, there are a number of disadvantages to using general-purpose multipliers:

- Extra logic is required to mask out the sign and exponent of the data and to insert the hidden bit. The output of the multiplier would have to be masked.
- Most available products cannot handle division. Thus a separate divider would have been required, which was expensive. Even division algorithms using multiplication require a large amount of ROM to contain the approximation constants.
- Many of the available designs are intended for integer applications, such as FFT butterflies and digital signal processors. Hence, the designs are optimized for those applications. Extending these 8- or 16-bit multipliers to a larger word length, as required for the VAX architecture, was neither straightforward nor cost effective. Moreover, the normalization and rounding of results entails either extra logic or additional cycles if the floating point hardware in the E Box is used.
- Most designs have a clock system not consistent with the rest of the machine. This fact introduces the complication of a special clock distribution and difficulties in verifying the design.
- Very few designs are based on ECL technology. Other technologies, such as TTL, would require a different power rail and thus an extra power supply.

The closest available multiplier to the 8800 requirements is the 10901 made by Motorola, Inc. This MCA implementation contains an 8-bit by 8-bit multiplier together with a 16-bit adder. However, no latches are included; they must therefore be provided externally, thus increasing the cost substantially. On the other hand, division could be provided by repeatedly using the 16-bit adder of the 10901.

Division

The multiplier performs a nonrestoring division algorithm, 1 bit per cycle, for the F, D, and G formats. The divider can accept a new dividend bit during every cycle, thus permitting a 128-bit by 56-bit divide. A divide of this size is used in the H format algorithm to form the starting approximation.

The booth recode of the multiplier is modified slightly to accommodate the division decode.[2] In the case of multiplication, the multiplier recode selects the correct multiples of the multiplicand to add to the partial product during each multiplication operation. In the case of division, the divisor is loaded into the MD latch, and the booth recode selects either $+1$ or -1 times the divisor for each division step.

In the nonrestoring division algorithm, the sign bit of the previous result selects the correct divisor multiple for the next cycle. This selection is facilitated by feeding the sign signal into the modified booth recode so that it will select the multiples of either $+1$ or -1 times the divisor.

The quotient bit generated every cycle is sent to the shifter unit to be absorbed. The first quotient bit generated corresponds to the most significant bit of the answer. That bit is then normalized and rounded by the shifter.

Microcode Design

Being integrated into the logic in the main machine, the floating point logic is also controlled by the main microcode. The VAX 8800 CPU is an extensively pipelined design.[5] Although pipelining is a well known technique for improving performance (for example, the VAX 8600 CPU), it comes at a price: the microcode branch latency will increase. By that we mean that the microcode cannot branch on a condition or flag in the very next instruction; instead, it must wait a number of cycles. This delay is a consequence of the overlapping of the microinstructions; each successive microinstruction starts before its predecessor has completed.

Figure 5 shows a typical pipeline similar to that used in the VAX 8800 system. The microinstruction is subdivided into five components:

- In NEXT ADDRESS, the address for the next microinstruction is computed, as well as those for any selected branch conditions.
- In LOOK-UP, the microcode RAM is accessed to fetch the microinstruction specified by the current NEXT ADDRESS.
- In READ, the register file is read to fetch the specified operands (e.g., fetch R0 and R1).
- In ALU, the operation in the arithmetic logic unit is performed (e.g., R0 + R1).
- In WRITE, the result of the ALU operation is written back to the register file.

Thus when the next-address cycle has completed for the first microinstruction, A, the next-address cycle for the microinstruction, B, in the subsequent cycle is started. This cycle now overlaps with the look-up cycle for A. As many as five operations can proceed simultaneously in this manner.

The branch latency of this pipeline is governed by the first microinstruction that can "see" a branch condition set in an earlier cycle. For example, if the ALU cycle of A sets a carry condition, then the first instruction that can possibly use this signal in its next-address cycle is E. Thus the branch latency is three microinstructions, as shown in Figure 5.

Naturally, this branch latency influenced the way in which we designed the logic to perform floating point operations. Clearly, we had to avoid branching whenever possible as this would result in an excessively slow algorithm. Instead, we had to adopt a strategy based on prediction and provide extensive hardware assistance.

Prediction is based on the fact that the speed of algorithms for floating point adds are usually data dependent. For example, for certain data values, the result of a floating point add will require considerable normalization. That requirement is always present when two values

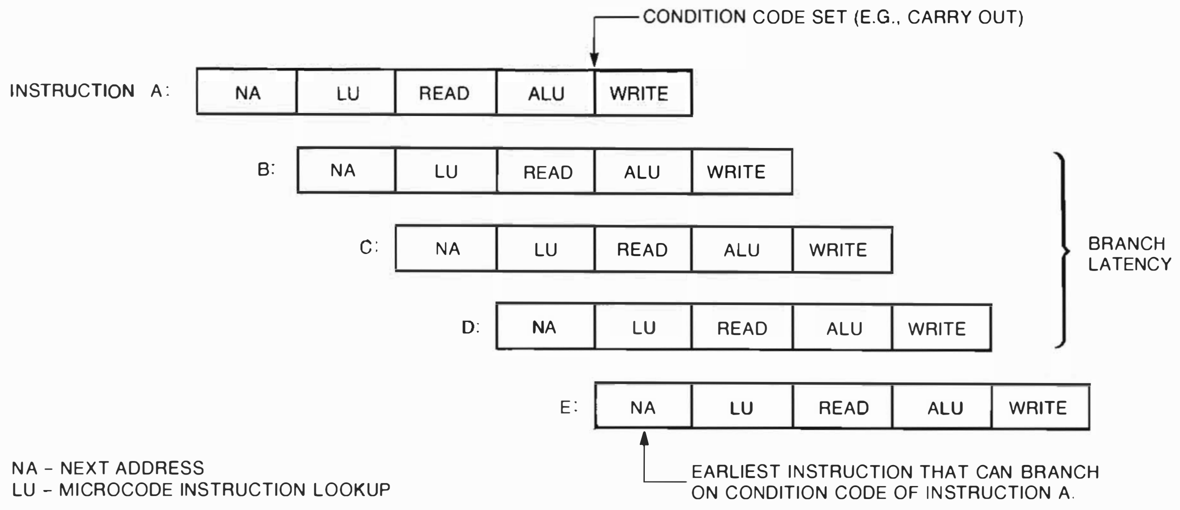

CONDITION CODE SET (E.G., CARRY OUT)
INSTRUCTION A:
NA
LU
READ
ALU
WRITE
B:
C:
D:
E:
BRANCH LATENCY
NA - NEXT ADDRESS
LU - MICROCODE INSTRUCTION LOOKUP
EARLIEST INSTRUCTION THAT CAN BRANCH ON CONDITION CODE OF INSTRUCTION A.

Figure 5 Five-stage Pipeline

of similar magnitude and large cancellation are subtracted. In other cases little or no normalization is required. It is clearly preferable not to pay the penalty of unnecessary normalizations.

The approach we took in the 8800 is to proceed down the most likely path, assuming that a small normalization will be required while waiting for the result of the branch signals. The add and subtract algorithms in particular are structured that way. The SALU examines the exponents of the operands and other signals; then it sets approximately 20 branch conditions in the first two cycles of the add/subtract datapath.

In certain situations all paths may be equally probable. In these cases the microcode enables hardware signals to control the datapath. A good example of this processing is the selection of operands. For a floating point add, it is natural to think in terms of the larger and the smaller operands. For example, the smaller operand is the one that is always aligned. However, the microcode does not know which register location holds the smaller value, and it does not want to wait for the whole branch-latency period to find out.

Therefore, the microcode will assume that the larger operand is in a particular register. Should this assumption be incorrect, then the SALU will swap the register file read addresses (thus sorting the operands). Not all locations have their addresses modified in this manner since the microcode still needs to be able to read and write to specific locations.

Similarly, the SALU determines if the main ALU is to do an add or subtract operation. At this point in the computation the microcode is unaware of which operation will be required. The pipeline is still within the long branch latency of the 8800 and cannot branch until this latency delay has elapsed. Note that one of the most frequently performed instructions is ADDF. That instruction will have just completed by the time the microcode can finally branch. Therefore, the ADDF cannot execute any faster since it is limited by the branch-latency delay. Consequently, those instructions that are the most probable cases are completely hardware driven.

To allow fast paths in the add algorithms, it is necessary to know that the result cannot possibly overflow since overflowed results must never be written. To prevent overflow the SALU examines the exponents of the operands. It then determines if the exponent of the result could possibly overflow or underflow, taking into account any possible normalization shift. There is also the added complexity of a rounding operation provoking an extra normalization step. That would happen when the rounding increment caused a carry to propagate throughout the whole fraction.

Consequently, the use of a small 8-bit incrementer for the round operation is possible only if it is known that an overflow cannot happen. The reason for this is that halting (trapping) the machine is not instantaneous (for the same reason that branch latency exists); therefore, the result will always be written. Thus, although the microcode can eventually correct the result, it cannot prevent that result from writing.

Performance Issues

When a program with many floating point instructions — such as LINPACK — is run, its performance is not totally dictated by the raw floating point speed of the CPU. Having a more profound effect are other factors, such as

- The size and organization of the cache — This factor is particularly important for programs with large amounts of data because the operands will reside in memory. Having superior register-to-register performance will not help in this type of program. Clearly, the larger the cache, the greater the chance that the required data will be quickly available, thus avoiding a lengthy transaction with memory.

- The performance of the integer and control instructions — Even programs performing extensive floating point operations still have significant amounts of integer and control instructions. Doing these quickly can contribute substantially to the program's performance.

To illustrate the effect of these factors, compare the performance of the VAX 8800 system with that of the VAX 8650 when both run LINPACK, as shown in Table 2.[3] The 8650 has faster raw floating point speed, especially for the F format (over twice as fast). Yet the two systems run this benchmark with almost the same performance. Clearly, in programs with these characteristics, factors other than raw speed will have a greater influence on performance. Of course, in applications without them, the raw speed advantage of the 8650 will be more pronounced.

Table 2 LINPACK Performance

	Performance (MFLOPS)	
Computer	F Format	D Format
VAX 8800	1.35	0.99
VAX 8650	1.30	0.70

Summary

The architecture of a processor like the VAX 8800 CPU is all a matter of trade-offs. Where does the performance make a difference? For example, we could have supplied the 8800 with a separate floating point unit to achieve faster performance. Doing that, however, would have required at least one extra module. To keep the cost of the system constant, this extra module would have entailed removing a module of logic from some other part of the computer. Perhaps removing that module would have resulted in a smaller cache or a simpler decoder with no optimizations for the frequent instructions. In any case the net effect would have been to sacrifice the performance of the computer in some other area. All things considered, we feel that the design is well balanced for the multitude of different computing tasks that customers will perform with the VAX 8800 system.

Acknowledgments

The authors would like to thank Ron Melanson and his team for the circuit design of the custom multiplier. In addition, we would like to thank Dave Sager for his help and guidance.

References

1. R. Burley, "An Overview of the Four Systems in the VAX 8800 Family," *Digital Technical Journal* (February 1987, this issue): 10–19.

2. T. Fossum, W. Grundmann, and V. Blaha, "The F Box, Floating Point in the VAX 8600 System," *Digital Technical Journal* (August 1985): 43–53.

3. *VAX Architecture Manual* (Maynard: Digital Equipment Corporation, Order No. EB-19580, 1981).

4. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software in a FORTRAN Environment," Argonne National Laboratory (May 1986).

5. S. Mishra, "The VAX 8800 Microarchitecture," *Digital Technical Journal* (February 1987, this issue): 20–33.

James P. Janetos

The VAX 8800 Input/Output System

The VAXBI bus links the processors in the VAX 8800 family to I/O devices, including clusters and networks. The VAX 8800 multiprocessor can support four of these 32-bit synchronous buses, each of which connects up to 16 I/O devices. Each VAXBI bus connects to the memory interconnect, the NMI bus, by an NBI adapter, which contains an interface chip to implement the VAXBI protocol. The NBI adapter logic handles CPU references and direct memory accesses to and from the I/O devices. The adapter has its own 200-nanosecond clock, which is completely asynchronous with the 45-ns CPU clock.

The VAX 8800 family of systems is another major step for Digital Equipment Corporation into the realm of high-performance computing. While increasing the computing capability of the VAX line for scientific and technical applications, these systems will undoubtedly play an important role in commercial and office markets. In these markets, the ability to connect to a computing cluster, service many users, and function in a network are as important as a fast CPU. Indeed, in a multiuser, multiprogramming system, the efficiency of "housekeeping" operations affects the perceived system performance as much as raw processor computing speed. These operations include sharing memory between many programs, swapping processes into and out of memory, paging, and responding to interactive user requests.

All members of the VAX 8800 family use Digital's new VAXBI bus as their communication link to clusters, networks, and interactive users. With its ability to connect to four separate VAXBI channels, the VAX 8800 system in particular offers great flexibility in configuring peripheral devices and interfaces. This paper first discusses the characteristics of the system communication buses in the VAX 8800 system. Following that is a discussion of the interface, called the NBI adapter, linking the primary system bus to the VAXBI input/output (I/O) bus. Figure 1 illustrates the various components of a VAX 8800 system.

The Processor-to-Memory Bus

The two CPUs, the I/O subsystem, and memory all share the primary system bus, called the NMI bus. This bus is a limited-length, high-speed synchronous communications path that provides the data link between these four devices. The NMI bus is completely contained in the main system cabinet; its cycle time is 45 nanoseconds (ns), the same as the CPU's. The bus protocol handles several outstanding transactions at one time, thus effectively increasing the bus's utilization. That is, once a device has issued a transaction (e.g., a read), that device relinquishes the use of the bus until the responding device is ready with the data. Other devices are then free to start other transactions.

In this fashion, the bus usage is greatly increased. The two CPUs communicate directly with memory over the NMI bus; the I/O devices connected to the VAXBI buses access memory via the NBI adapters. A device on the NMI bus is called a "nexus." Arbitration among nexuses occurs in parallel with data transfers and is handled by one CPU in a nearly round-robin fashion. This guarantees that each nexus gains its fair share of the bus resource. Data transfers on the NMI bus occur in longword, octaword, and hexaword lengths (4, 16, and 32 bytes respectively). Four levels of device interrupts are supported.

The VAXBI Backplane Interconnect

The VAXBI bus is used as the I/O bus for the VAX 8800 system. As shown in Figure 1, from one to four VAXBI buses can be interfaced to the NMI bus, depending on a customer's needs and his desired mix of peripheral devices. Each VAXBI bus is a 32-bit-wide synchronous bus that can connect up to 16 VAXBI devices. Each VAXBI device, called a "node," uses a chip called the VAXBI Interface Chip as its bus interface. This chip provides a consistent logical and electrical interface to the bus. The VAXBI Interface Chip implements most of the bus protocol for its node, including bus arbitration and error checking. The VAXBI cycle time is 200 ns, controlled by an oscillator on the NBIB.

The NBI adapter acts as both a processor and a memory on the VAXBI bus. The adapter provides the following three important functions:

1. A means for the master CPU to read and write device registers
2. A window into memory for VAXBI devices
3. The facility for VAXBI devices to interrupt the processsor

Control of Peripheral Devices

To gain an appreciation of the NBI adapter architecture, it is worthwhile to discuss the control of peripheral devices.[1] To move data from a disk into memory or to send program output to a peripheral device, a programmer must specify the operation to be carried out (read or write), a memory address to receive the data or that contains data to be output to a device, and the amount of data to be moved. In early machines, the processor was required to control the entire operation — executing instructions to move the data, waiting for the slower device to complete the operation, and then continuing in this fashion until all the data had been moved. This process wasted a great deal of processor time since many instructions could have been executed while waiting for an I/O operation to complete.

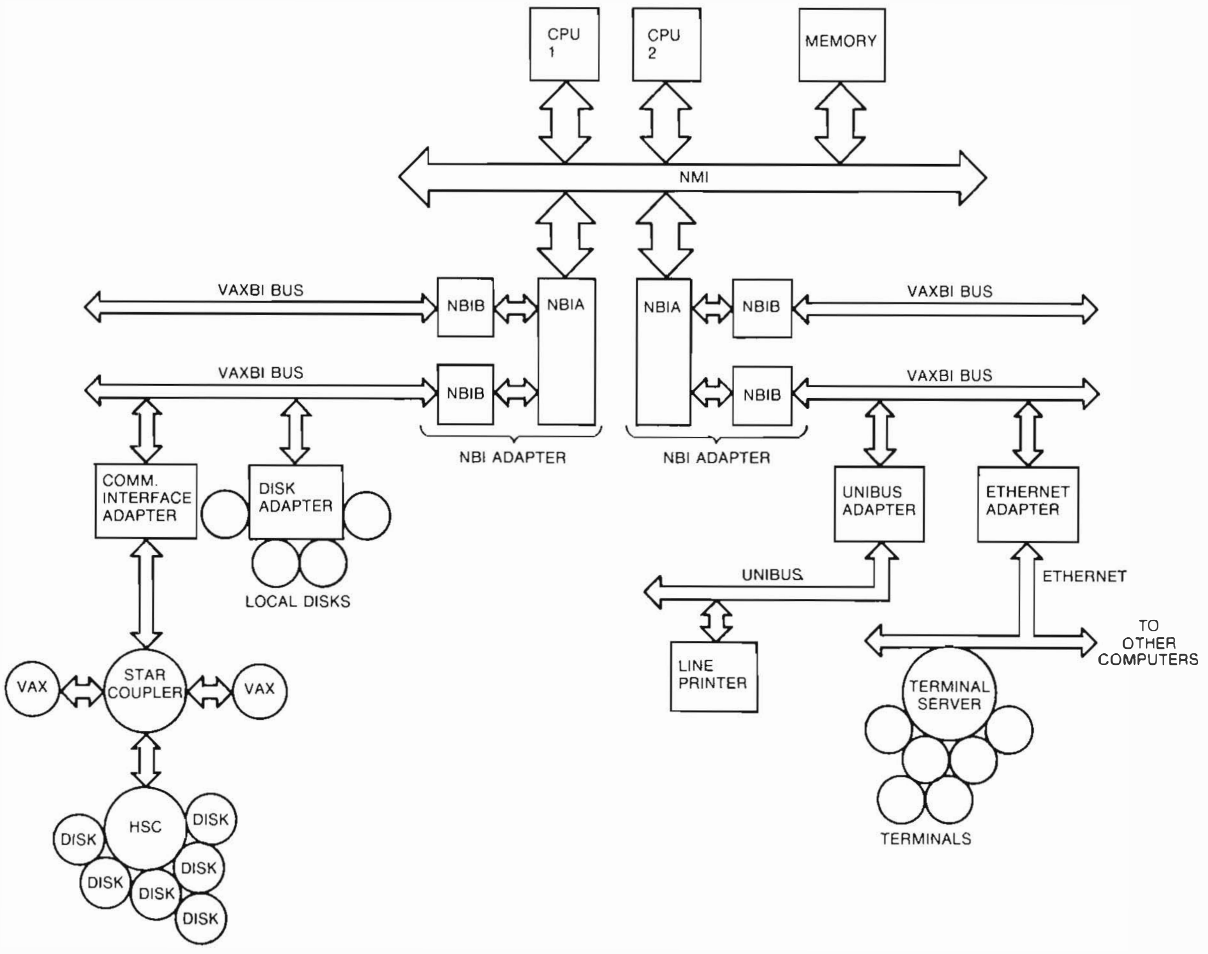


Figure 1 VAX 8800 Configuration

Modern machines have I/O controllers. which are special hardware interfaces that handle device operations. A programmer must specify to the controller the attributes of the operation to be carried out. Once the operation is accepted by the controller, the processor is freed from the details of actually moving the data. In this way processing and I/O operations can be overlapped, increasing processing utilization.

For slow devices, such as terminals, the controller usually has a small buffer to hold the data to be transferred to or received from the processor. This buffer is loaded by the processor when it has data to be transmitted to the device. The device accepts the data, then signals when ready for more. When having data to be transmitted to the processor, the device loads that data into the buffer and then signals to the processor to remove the data. This process is called programmed I/O.

For high-speed devices, such as disks, the I/O controller normally performs direct memory access (DMA) operatiions. The processor loads special registers in the controller with information about the transfer — the amout of data to be moved and its location and destination. The processor is then freed while the controller performs the transfer. In this way large amounts of data can be moved with miinimal processor intervention.

Addressing in the VAX 8800 CPU

The master CPU manipulates the I/O controllers with reads and writes of single lonwords to their control and status registers. These registers have addresses in physical address space and can be manipulated by standard VAX instructions. This technique contrasts with that used in many computers in which special instructions control I/O. The address range of the VAX architecture is shown in Figure 2, in which addresses are given in hexadecimal notation.

Physical memory occupies the first 512 megabytes of the defined address range. The I/O adapter and the I/O controller registers are located in the range from 2000 0000 to 3FFF FFFF. In the I/O space, the address range allocated for each VAXBI bus is further subdivided into space for each device on the bus.

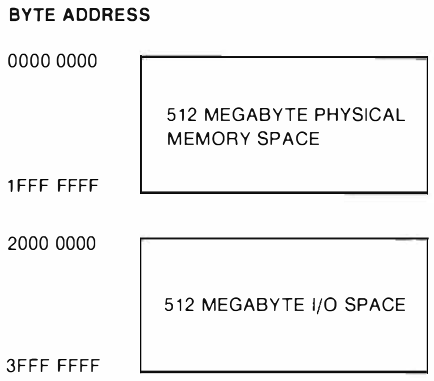


Figure 2 VAX Address Space

The NBI Adapter

An adapter provides an interface between two existing buses, each with its own addressing protocol and data-transfer protocol. The adapter is responsible for all communications between the two buses. It is a datapath for the processor to access device registers and for devices to access memory. This datapath is also ued to interrupt the processor and for initialization functions.

The NBI adapter, consisting of an NBIA module and either one or two NBIB modules, interfaces the VAX 8800 system to the VAXBI buses, which are I/O buses in this application. That is, the NBI adapter issues reads and writes on the VAXBI buses in response to reads and writes that are in the NBI address range initiated by the processor on the NMI bus. Likewise, the NBI adapter issues reads and wites to memory on the NMI bus in response to reads and writes initiated by VAXBI devices on the VAXBI buses. The NBI adapter in the VAX 8800 system supports a new generation of high-performance, native VAXBI devices.

Figure 3 contains a block diagram of the NBIA/NBIB adapter system. Basically, the datapath of the NBIA module contains an NMI interface, which provides buffering for addresses and data transmitted and received during NMI transactions. The NMI interface is connected to a transaction buffer, which is a 16-location, dual-ported ECL/TTL RAM. The transaction buffer provides five locations to buffer commands and addresses and up to four longwords of read/write data for direct memory access (DMA) transfers by devices on the VAXBI-0 bus. A second group of five locations is provided for DMA transfers by devices on the VAXBI-1 bus. Two locations are used for the command/address packet and the single longword of read/write data transferred when the processor accesses the VAXBI device registers. The NBIA/NBIB TTL datapath indicating the layout of the transaction buffer is shown in Figure 4. The TTL port of the transaction buffer connects to a set of two bidirectional latches used to buffer commands, addresses, and data for transmission across the data-bus cable to and from an NBIB module.

The datapath of the NBIB module consists of a set of four bidirectional latches used to buffer both DMA commands and addresses and CPU commands and addresses, as well as data. These latches connect to another set of latches known as the BCI data buffer (one longword deep), which connects to the VAXBI Interface Chip. (The module side of the interface chip is known as the BCI.) The interface chip controls the enabling of data onto the BCI for data transmission onto the VAXBI bus.

Data flows between the NMI bus and the VAXBI bus by moving it between these two sets of latches. Control logic moves data from stage to stage, passing control successively to the next stage as each part of the transfer completes. The VAXBI bus runs approximately four times slower than the VAX 8800 processor and is asynchronous with it. Therefore, the additional problem exists of synchronizing control between the NBIA and NBIB modules. Facilities are provided for delaying data transfer until a buffer is free, thus preventing data corruption. Another synchronization problem occurs when the master processor wants to read from or write to a VAXBI device when that device wants to make a memory access. The control logic in the NBIA and NBIB modules is carefully designed to referee such contention problems.

DMA Transfers

From VAXBI Devices to Memory

A DMA transfer to memory by a VAXBI device is shown in Figure 5.

After winning the VAXBI bus, the device wanting to make a transfer initiates a command and address cycle. In Figure 5, that device is a disk controller. The VAXBI Interface Chip in an NBIB is programmed to recognize memory addresses on the VAXBI bus. The chip "awakens" the NBIB control logic, decodes the command, and stores the command/address packet, as shown in Figure 4. Control logic on the NBIB then sends a "DMA request" signal to the NBIA. After a synchronization delay on the NBIA, the NBIA TTL controller begins to transfer the command and address from the NBIB to the NBIA.

Meanwhile, the NBIB takes the longwords of data as they appear on the VAXBI bus and stores them in the NBIB's data buffers. The NBIA stays approximately one cycle behind the NBIB, removing data from the NBIB buffers and storing

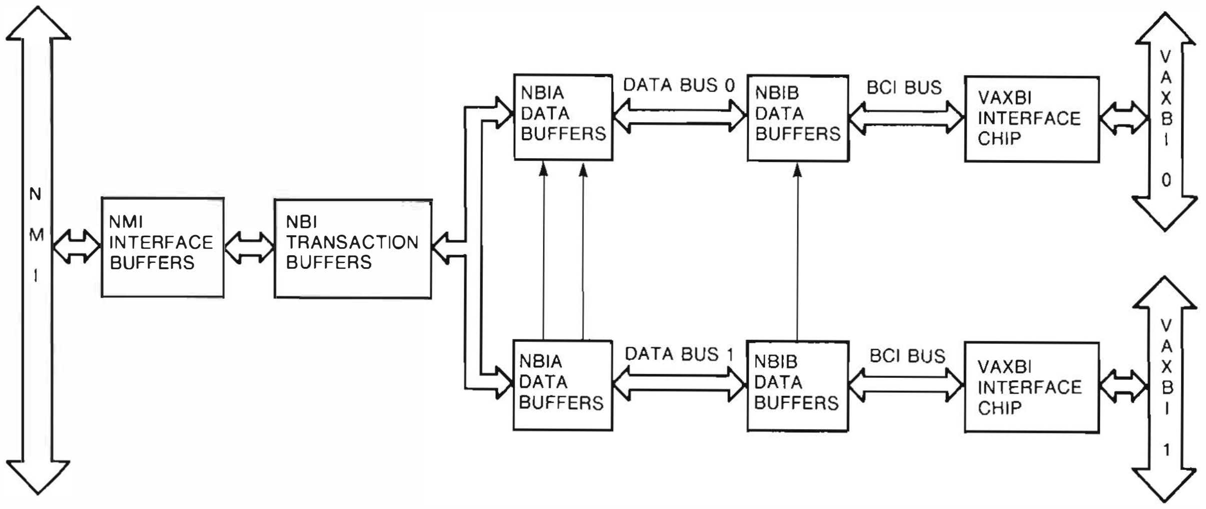


Figure 3 Block Diagram of NBI Adapter

it in the DMA locations in the transaction buffer. After successfully transferring all data into the transaction buffer, the NBIA alerts the NBIB, which, after a synchronization delay, ends the transaction on the VAXBI bus. At this time the NBIA TTL controller passes the DMA request to the NMI interface in the NBIA, which then performs the write to memory on the NMI bus.

It should be noted that a DMA write transaction is considered to be complete on the VAXBI bus before the data is actually written to memory. A VAXBI device is thus free to start another transaction immediately. This performance enhancement is known as a "disconnected write," in which the write operation is considered to be completed on one bus before that

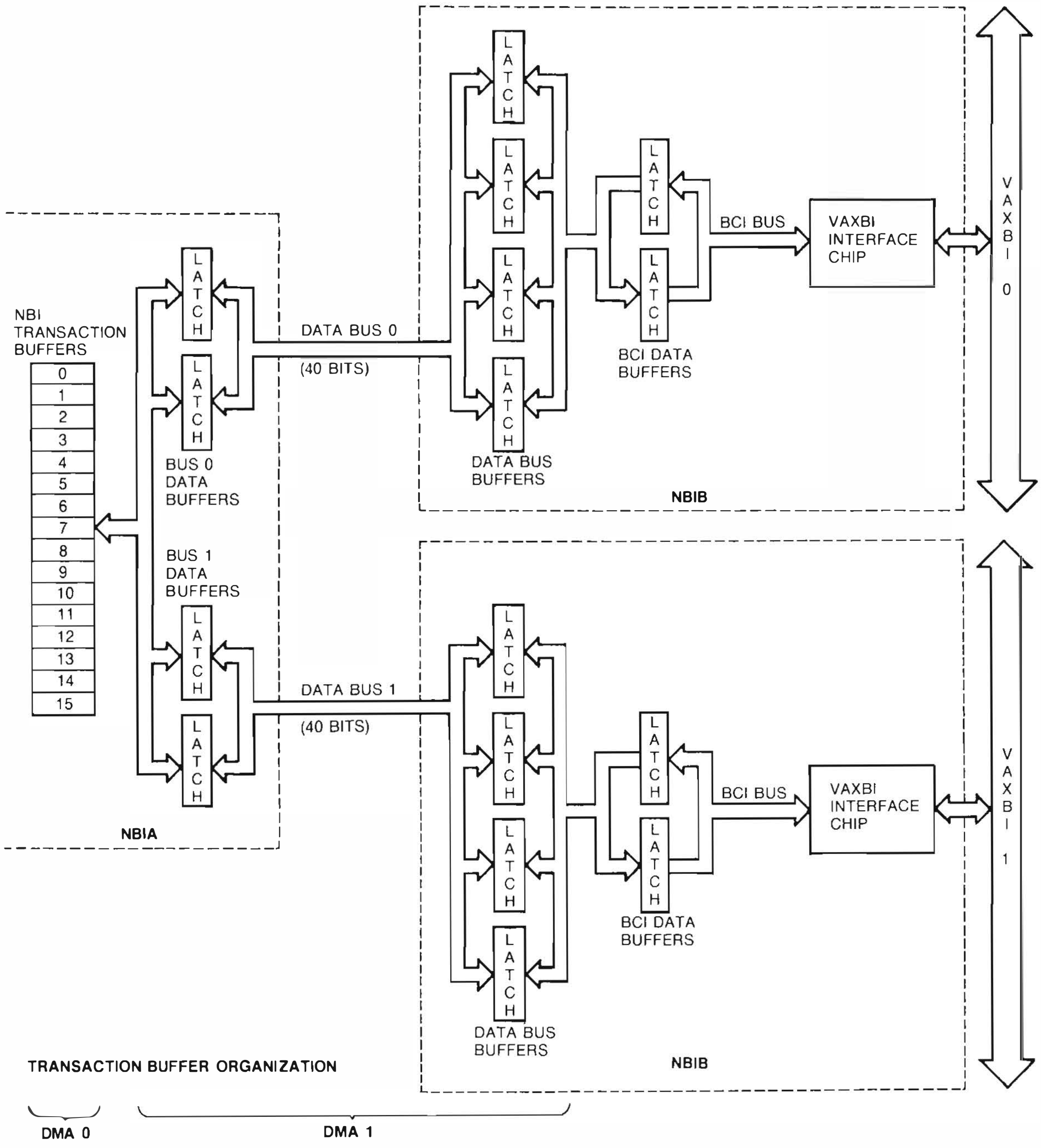


Figure 4 NBIA/NBIB TTL Datapath

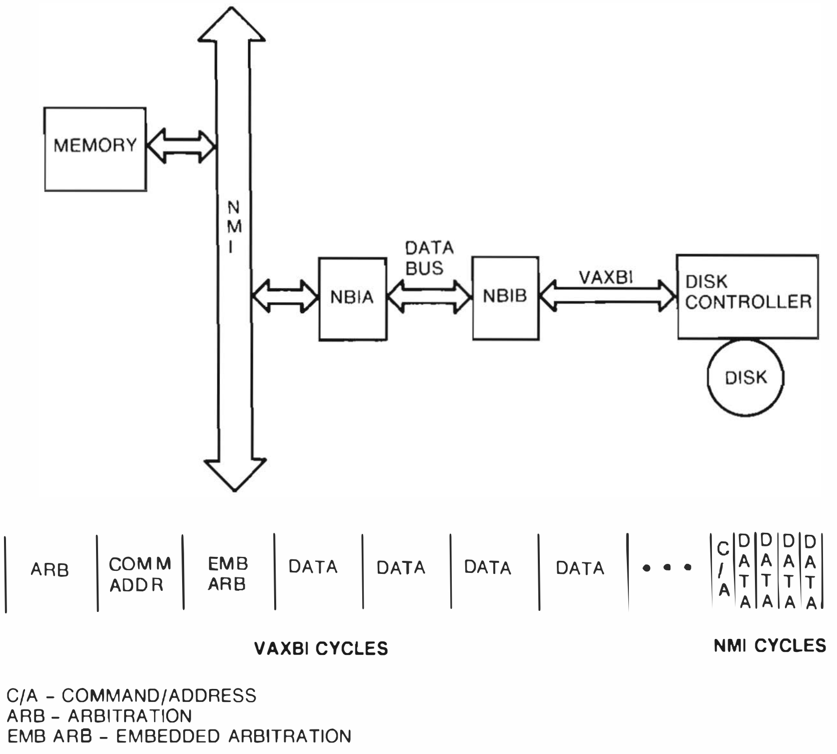


Figure 5 DMA Transfer to Memory

operation has actually taken place on the target bus. The NBI adapter is designed in such a way that a write transaction could be waiting in the transaction buffer (e.g., while the NMI interface controller services the other VAXBI bus) while a second transaction waits in the data bus transceivers. Using two levels of buffering and the disconnected write technique allows the NBI adapter to support a write bandwidth of 8 megabytes per second.

It is interesting to note that during the data transfer from the NBIB to the NBIA, the NBIB notifies the NBIA TTL controller of the DMA request immediately after storing the command/address packet. However, the NBIA TTL controller does not pass the DMA request to the NBIA NMI interface controller until the command/address packet and all the write data have been loaded into the transaction buffer. The reason for this delay is that the NMI interface controller runs at the same speed as the NMI bus, or 45 ns per cycle.

The NBIA TTL controller runs four times slower, or 180 ns per cycle, to closely match the VAXBI cycle time of 200 ns per cycle. Thus if the NBIA TTL controller were to signal the DMA request after loading only the command/address packet into the transaction buffer, the NBIA NMI interface would attempt to read data from the transaction buffer before that data had been loaded. That is obviously a bad thing to do. Indeed, the NMI interface of the NBIA can empty the transaction buffer in approximately the time it takes for the NBIA TTL controller to load one longword.

From Memory to a VAXBI Device

A write request from a VAXBI device is similar to the DMA operation just described. After winning the VAXBI bus, the device wanting to read data from memory on the NMI bus transmits a command and address on the VAXBI bus. Figure 6 depicts this transfer.

The interface chip awakens the NBIB control logic, which then decodes the command and stores the command and address in a data-bus buffer location. The NBIB then passes the DMA request to the NBIA immediately after the command/address packet is loaded. Again similar to the write operation, the command or address is transferred to the appropriate location in the transaction buffer by the NBIA TTL controller. However, a DMA read is unlike a write operation, in which the data is ready for transmission, in that the data must be fetched from memory. The DMA request is first passed to the NBIA NMI interface controller, which arbitrates for the NMI bus. Upon winning the bus, the interface controller initiates a read request to memory. When the the data is ready, the memory returns it on the NMI bus to the NBIA. Thence the data is transferred into the DMA locations in the transaction buffer, and the NBIA TTL controller is notified by the NBIA NMI interface that the data is ready. The controller then begins to transfer data to the NBIB, loading it into successive locations in the NBIB buffers. This process is illustrated in Figure 4. A "DMA Done" notification is sent to the NBIB after the first longword of data, rather than all the data, has been transferred. That maximizes the read bandwidth on the VAXBI bus. The NBI adapter has a maximum DMA read bandwidth of four megabytes per second.

The DMA read transfer illustrates one fundamental difference between the NMI bus and the VAXBI bus. Referring to Figure 6, one can see that the VAXBI bus is unusable while the NBIA and memory complete the read operation. (The NBIB issues stall signals to the requesting device during this time.) The NMI is a pended bus, but the VAXBI bus is nonpended, or interlocked. That is, the NMI bus is immediately available for use once a command has been transmitted and acknowledged, whereas the VAXBI bus must wait. Thus "pending" transactions are allowed on the NMI bus. Indeed, the NBIA NMI interface can respond to requests from the other VAXBI bus while also having an outstanding read to memory on behalf of the first VAXBI bus.

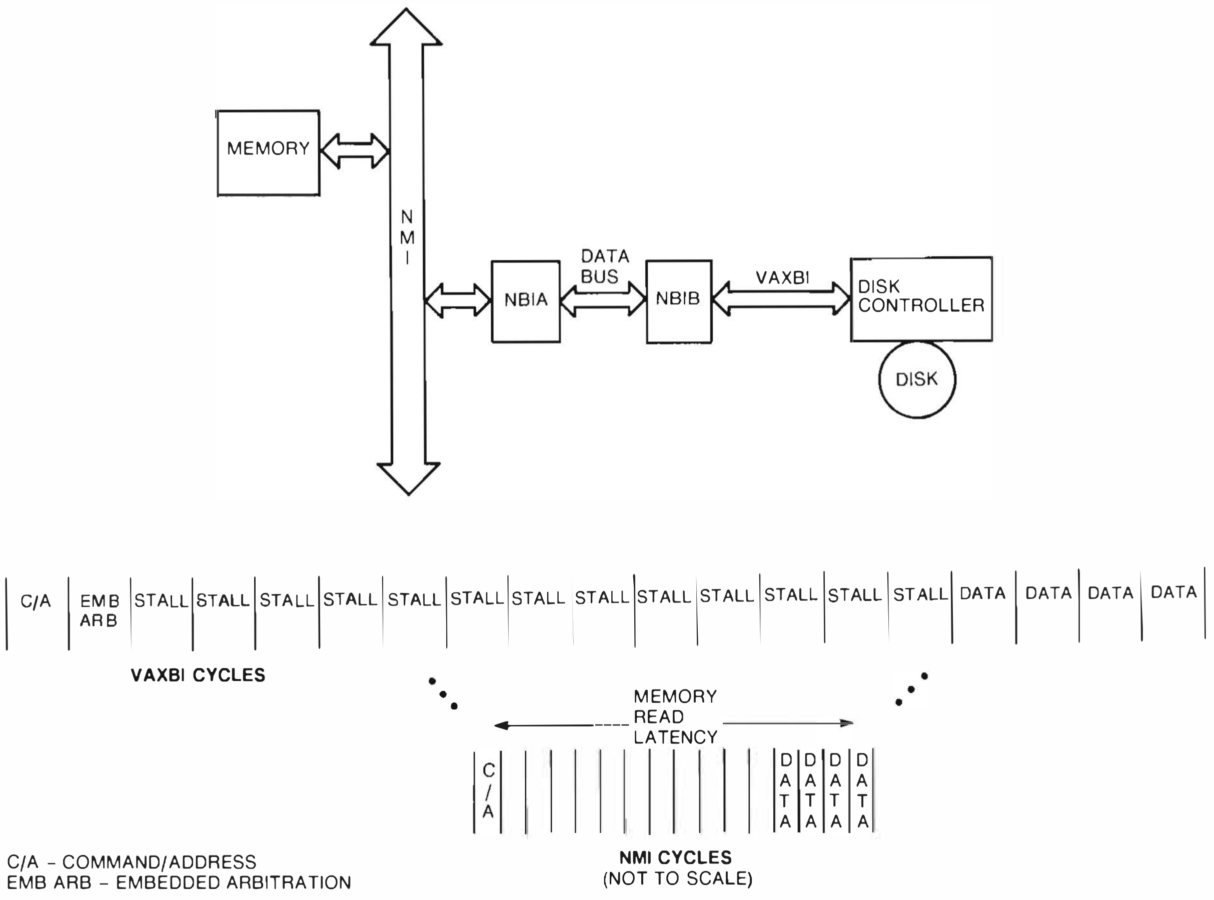


Figure 6 DMA Transfer from Memory

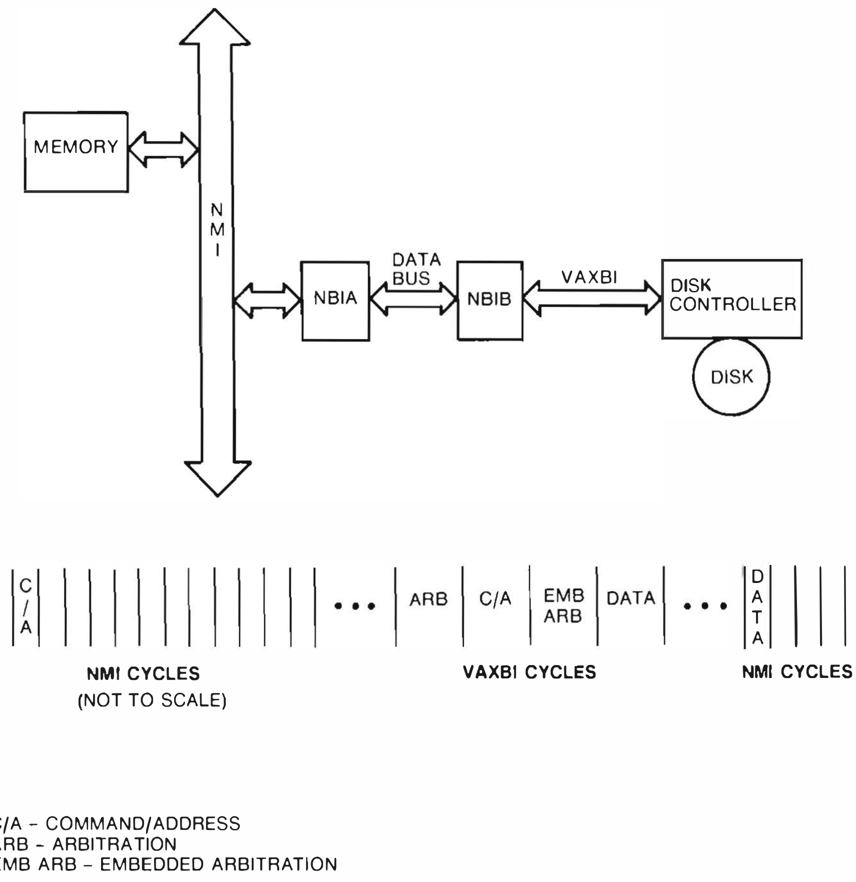


Figure 7 CPU Transfer from VAXBI Device

CPU Transfers to and from VAXBI Devices

CPU transfers to and from VAXBI devices are similar to VAXBI transfers to and from memory, the obvious difference being that the transaction is initiated on the NMI bus. CPU transfers are shown in Figure 7.

Another difference is that CPU transactions are limited to longword length when accessing VAXBI devices. Since there is only one location for a command/address packet for CPU transfers and one location for read/write data in the transaction buffer, the NBI adapter can handle only one CPU transaction at one time. These limitations lower the CPU-to-VAXBI bandwidth as compared to the DMA bandwidth. An analysis of bus traffic, however, has shown that CPU-initiated transactions account for under 10 percent of the VAXBI traffic in a VAX 8800 system. This finding could be anticipated since the CPU must make only a small number of accesses to a VAXBI device controller to cause it to transfer large amounts of data.

Synchronization

In the earlier discussions of data transfers, the term "synchronization delay" was introduced. In general, some type of synchronization is required whenever more than one independent clock exists in a system. This is the case in the VAX 8800 system. Timing for the processors, memory controller, and NBIAs is derived from a sophisticated clock module that provides two-phase, nonoverlapping clocks with a basic period of 45 ns and tightly controlled skew.[2] The VAXBI timing, on the other hand, is derived from an oscillator and a clock-driver circuit on the NBIB. This timing has a basic period of 200 ns, completely asynchronous to the VAX 8800 kernel. The synchronization of control signals is thus necessary for data transfer between the NBIA and NBIB modules. A DMA read transfer involves the synchronization of a "DMA request" and a "DMA complete" signal. Therefore, the synchronization overhead can account for approximately 5 to 15 percent of the time it takes to complete the operation.

Summary

The performance of the I/O subystem is critical to the operation of high-performance systems like those in the VAX 8800 family. The I/O adapter provides a communication link between the each processor, the memory, and the I/O devices. The NBI adapter is this link for these systems, providing access to a new generation of VAXBI devices and high-performance I/O operation for these important new machines.

References

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

2. W. Samaras, "The CPU Clock System in the VAX 8800 Family," *Digital Technical Journal* (February 1987, this issue): 34–40.

Paul C. Wade

The VAXBI Bus — A Randomly Configurable Design

The VAXBI bus provides a high-performance alternative to the UNIBUS system as Digital's general-purpose bus. The VAXBI design was completely specified before any hardware was built and is independent from any physical configuration. The designers had to discard the traditional small-perturbation approach and instead used many techniques to specify the bus characteristics. Two custom chips, a differential driver and receiver, are used to clock the bus. The bus designs were tested extensively with SPICE, but tests on the physical chips led to some unanticipated problems. Further analysis of waveforms, crosstalk, and switching noise led to changes that met all the original goals.

The VAXBI bus is a new, high-performance, general-purpose bus that provides a common interface to all of Digital's new VAX products, from the VAX 8200 CPU to the VAX 8800 system. This bus can also be used for future VAX systems. The VAXBI bus is a higher-performance replacement for the UNIBUS system and should have a similarly long and productive lifetime.

The UNIBUS system was enhanced many times during its long history. Since there was no formal specification for this bus until 1986, these many de facto enhancements led to numerous compatibility and configuration problems. Having learned from those problems, the VAXBI design team decided to make a complete design specification of the VAXBI bus before any hardware was built. Thus compatibility problems should not occur if all future designs comply with that specification.

One of the most important aspects of that specification — and the most difficult to implement — is that the VAXBI bus operates independently from any particular physical configuration. That is, the bus must be randomly configurable. The achievement of that specification was the most difficult part of the electrical design. The techniques and solutions involved in solving this problem should be instructive to future bus designers.

VAXBI Bus Description

There are several excellent references that describe in detail the operation of the VAXBI bus and the VLSI chip that implements the bus logic and arbitration.[1,2,3] Therefore, only a short description of the bus will be given here. The VAXBI bus is a general-purpose bus with data transfer rates high enough (up to 13.3 megabytes per second) to serve as a memory bus in mid-range VAX systems, such as the VAX 8200 CPU. All machines in the new generation of VAX systems use the VAXBI bus for all I/O, communications, networks, and connecting adapters for mass storage. Those high rates also allow it to serve as an I/O bus in all sizes of VAX systems by using multiple VAXBI channels in the largest systems, such as the VAX 8800 multiprocessor, shown in Figure 1.

All the machines in the new generation of VAX systems use the VAXBI bus for all I/O, connecting adapters for mass storage, communications, and networks. A VAXBI subsystem, consisting of two six-slot card cages and the backplanes, is shown in Figure 2. The backplanes are connected with flexible interbackplane jumpers with terminators at each end.

The key to general-purpose operation is the distributed nature of the VAXBI bus. All nodes on it contain identical interface hardware, and a

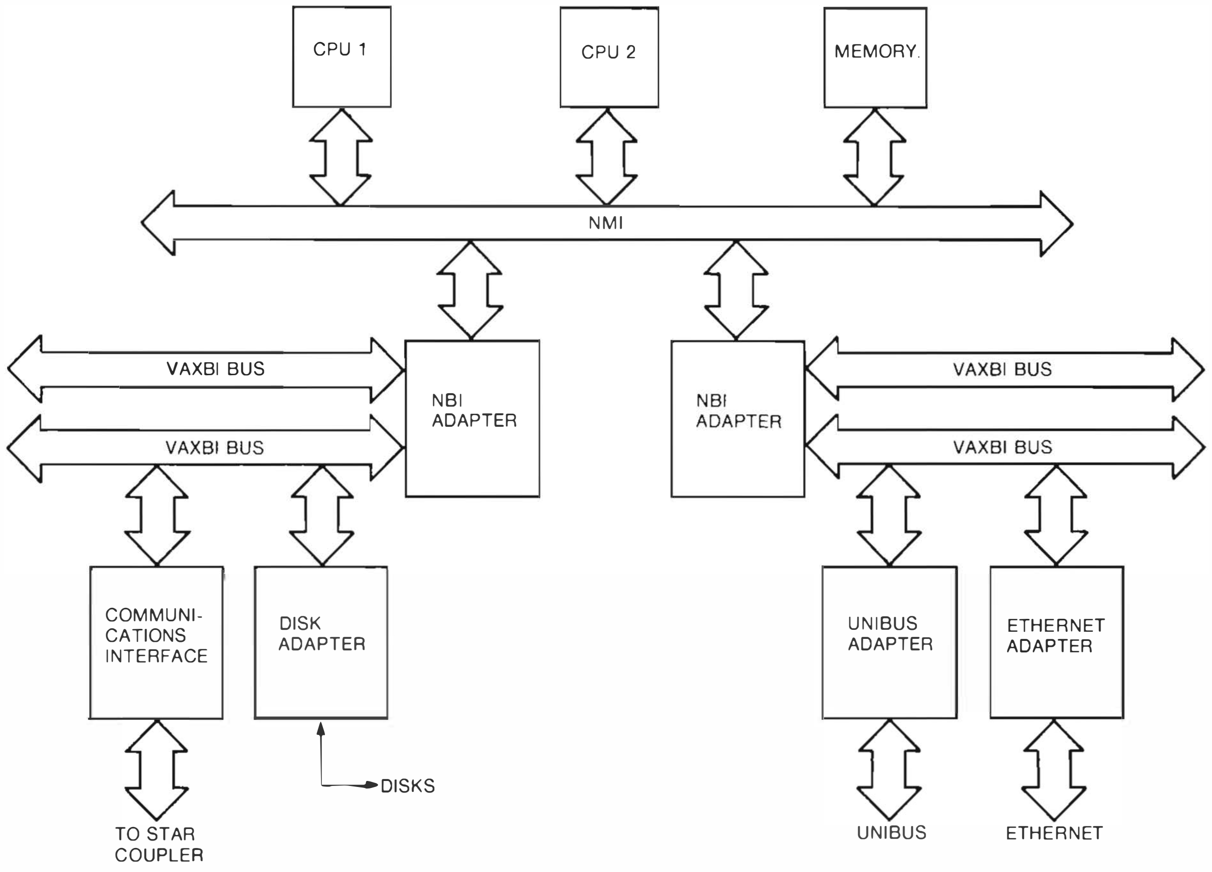


Figure 1   VAX 8800 System with Four VAXBI Buses

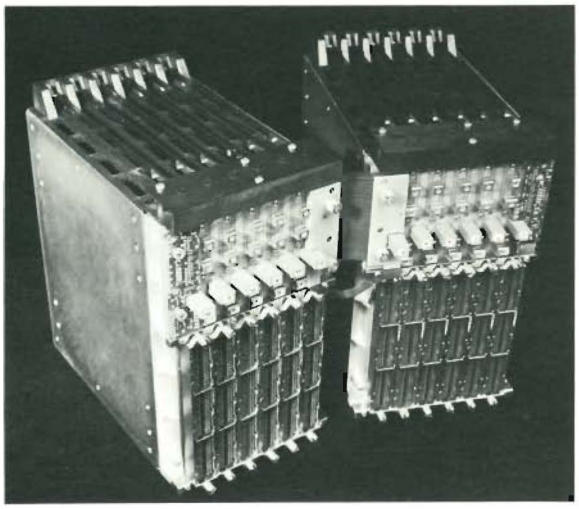


Figure 2   VAXBI Subsystem

distributed arbitration scheme precludes the need for a processor to act as a dedicated bus master. The VAXBI bus can support both multiple and networked processors, thus implementing Digital's strategy of distributed computing. The synchronous operation of the bus achieves high performance by providing predictable communication delays. The distributed arbitration is embedded within each bus transaction so that further data transactions may follow without delay.

The VAXBI bus architecture is rigorously specified, and all designs that are verified to its specification will be fully compatible with the bus. The task of system designers has been greatly eased by the incorporation of all data-handling and arbitration logic in one VLSI element, the 78732 chip, called the VAXBI Interface Chip.

That chip also performs self-test functions and bus error detection and handling to improve system reliability and robustness. The physical bus interfaces are also rigorously specified, and the bus clocking is controlled by custom clock-driver and receiver chips. Figure 3 shows the VAXBI corner of a module, with all the components required for the bus interface contained in a standardized layout. These features free a designer to concentrate on his unique design rather than on the bus details.

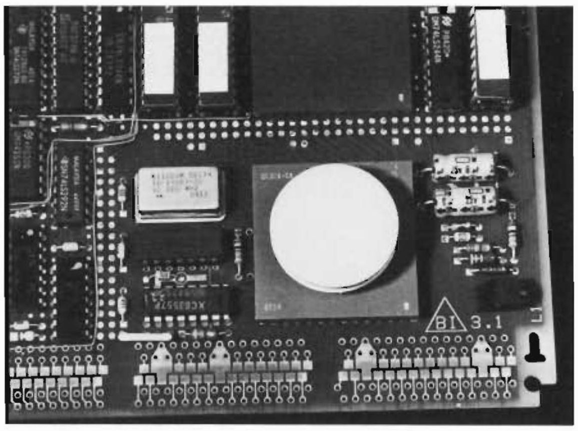


Figure 3 VAXBI Corner of a Module

VAXBI Electrical Design

A randomly configurable bus has many advantages as a data bus in general-purpose computers since their physical configurations are not known a priori and are subject to change during repair or upgrading. The previous state of the art within Digital was to use an artificial intelligence program, called XCON, to calculate a configuration for each unique set of UNIBUS options. XCON is based on an extensive set of bus configuration rules. Although it is a triumph of applied artificial intelligence, the necessity to use it for bus configurations was a bottleneck we hoped to avoid by better bus design with the VAXBI bus.

The design of a randomly configurable bus involves essentially the design of a group of aperiodically loaded transmission lines. The characteristics of regularly loaded transmission lines are well defined, but those of randomly and unpredictably loaded lines are less well understood. The design team evolved a design procedure from their work on the VAXBI bus. Although this procedure was derived from the development rather than being planned in advance, it may help bus designers with their projects in the future. Therefore, the remainder of this paper describes that procedure, especially the activities and results that proved most significant to the project.

The first step in designing this bus was the realization that the problem was not completely random but may be bounded. A bus is physically implemented as a group of transmission lines in a backplane. These lines are perturbed by the loading of connectors for modules and by the modules themselves. Each connector, or slot, in which a module may be inserted causes a small perturbation if empty and a larger one if populated. A transmission line can also continue through cabling and connectors onto another backplane. In either case the transmission line is terminated in some manner.

The classic method of dealing with transmission line loading is to make the characteristic impedance so low that perturbations will be trivial. In that case any reflections from these perturbations will be small, and the line can be end terminated in its characteristic impedance so that there is no reflection. The loading is then considered to be predominantly capacitive. Thus the loaded impedance can be calculated as

$$Z_o' = Z_o / \sqrt{1 + C_d / C_o}$$

Our first approach was to determine if the classic method could be used to deal with transmission-line loading for the modules on the VAXBI bus. Z_o, the characteristic impedance, ranges from 35 to 100 ohms for the standard dimensions of organic printed circuit boards made by Digital. Corresponding values of C_o, the intrinsic line capacitance, range from 1.8 to 0.6 picofarads per centimeter (pf/cm). However, C_d, the distributed loading capacitance, can be as much as 5 pf/cm for modules in this implementation. That capacitance means that Z_o', the loaded impedance, would be in the range of 18 to 33 ohms, clearly a major perturbation. Therefore, for modules with these characteristics, the small-perturbation approach could not be used.

In the case of the VAXBI bus, even if it were possible to produce lines whose characteristic impedances were low enough ($Z_o < 15$ ohms), massive drivers would be required to supply the necessary current. Therefore, bus power would become a significant portion of the system power dissipation, an undesirable situation. Consequently, we had to consider a design approach different from the classic one.

Our alternative design approach was more pragmatic. Significant development investments had already been made in several key components, particularly the module connector and the 78732 chip. Therefore, the rest of the design had to be as compatible as possible with the characteristics of those key components. Particular attention was paid to three areas: the physical layout, to keep capacitance within the drive capability of the 78732 chip; the clock, since it is the critical element in bus timing; and grounding, which is critical for signal integrity.

The VAXBI data lines are driven directly by the 78732 chip, which is fabricated using advanced MOS technology. MOS devices, however, are limited in their ability to drive current. Within the constraints of chip area and power dissipation, open-drain drivers of about 21 milliamperes (ma) are the only ones available. The data cycle of the VAXBI is 200 nanoseconds. Therefore, the maximum bus length of 1.5 meters (VAXBI specification) is short compared to a wavelength, and a lumped-constant approximation could be used for calculating the delays. An RC time-constant model was used for this approximation, and the voltage swing was limited to 3 V to accommodate a smaller terminating resistor for faster switching. The resulting resistance was 238 ohms (5 V/21 ma).

After calculating the tolerances and worst-case allowances, we chose a standard value for this resistance of 270 ohms. By choosing an RC time constant equal to the maximum available propagation delay (and after subtracting device delays and allowing for component tolerances and a 10 percent timing margin), we calculated the capacitance as 410 pf. This figure became the maximum capacitance for each data line, including backplanes, interbackplane jumpers, connectors, modules, and bus transceivers on the chips. Obviously, the RC time constant is applicable only on the low-to-high transition, when the open-drain device is turning off. Device turn-on, which is normally much faster, is internally compensated for by controlling the rise time to minimize the transmission-line reflections.

For the clock lines, the timing requirements are critical enough to justify the use of very large drivers since only two signals are involved. We selected a differential configuration for clock signals in order to minimize the skew, which could degrade timing accuracy. This configuration also provides noise immunity by common-mode rejection. Since the clock frequency is much higher than the data frequency, ECL was chosen for the logic technology. The maximum drive capability of standard devices is 25-ohm impedance, however, so a custom driver is required. We also chose to use a custom differential receiver, for the following reasons:

- Both parts can operate from the available +5 V supply rather than the −5.2 V supply normally required for ECL.
- The receiver sensitivity and common-mode range can be optimized for the driver.
- The receiver input can be designed for minimal bus loading capacitance.
- The receiver output levels can be standard TTL levels, thus eliminating the need for a separate integrated circuit (IC) for level translation.

Altogether, these two custom clock chips do the work of five standard ICs, thus saving power and module real estate while improving performance.

Since the characteristics of ECL drivers are well understood, we require the clock driver to use an output driver made from three standard 50-ohm ECL drivers in parallel. Thus the effective drive capability is 17 ohms (50 ohms/3). The design termination is intended to match the estimated impedance of a maximally loaded system, approximately 25 ohms differential impedance. This impedance is composed of a resistor to ground from each line and a resistor between lines, chosen to sink the appropriate high- and low-state currents. The design was extensively modeled using the SPICE circuit simulator, which indicated that the driver had adequate current capability for this load.[3] The characteristic impedance of the clock lines was made as low as possible by maximizing the line width within the space constraints of a 0.1-inch via-hole (plated-throughhole in a printed circuit board) grid. To improve the common-mode rejection, the two lines of each differential pair are located one above the other on adjacent layers with ground planes above and below the pairs.

Finally, careful attention was given to the ground return path for all VAXBI signals. Ground planes, to minimize inductance, are provided on the modules, backplanes, and interbackplane jumpers for data lines as well as the clock lines described above. The data-line capacitance was constrained within the 410-pf limit described above by controlling the line width and the ground-plane spacing. A particularly difficult problem is the ground inductance of the 78732 chip. The 78732 chip can switch as many as 48 data lines simultaneously, with a total switching current of over one ampere. The induced voltage, V, from simultaneous switching is calculated as

$$V = L \times (di/dt)$$

in which L is the inductance and di/dt is the rate of current change. For example, if the ground inductance were 10 nanohenries and the chip switched in 10 nanoseconds, 1 volt of switching noise would result. Based on these noise calculations, we designed the package with an internal ground plane and 15 ground pins to minimize inductance and switching noise.

Test Results

When the custom clock devices became available, measurements showed that the driver could not power a 25-ohm differential load and still maintain the desired 700-mV amplitude over all conditions. Therefore, we carefully measured the output characteristics in both the high and low states to calculate an optimum termination. The TK!Solver software was used to solve iteratively the driver equations for the piecemeal linear approximations of the driver characteristics, which did not fit any simple curve. We then calculated the optimum resistances and chose the nearest standard resistor values. We also recalculated the output voltages for normal tolerances of resistance, voltage, and temperature, and a $+/-50$ percent variation in the internal resistance of the driver. The minimum calculated amplitude was 695 mV, giving us a very high confidence of having at least 700 mV for any actual hardware.

The optimized termination has a differential impedance of 37.6 ohms, which turns out to be a better match for the measured impedances of the rest of the hardware. An empty backplane has a differential impedance of approximately 60 ohms, dropping to as low as 28 ohms when fully populated; a jumper cable between backplanes typically has a 45-ohm differential impedance. The various possible VAXBI configurations yield a maximum reflection coefficient at any point of 0.28; probable configurations will have even smaller reflections.

Reflections of this magnitude could cause significant timing variations in single-ended systems due to a fixed receiver threshold voltage. However, they have no effect on a differential line since the reflection is the same on both lines of the differential pair. The only variation we found was caused by the differences in impedances on different printed circuit layers. Subsequent experiments indicated that improving the matching of impedances by putting the differential pair on the same layer reduces the skew more than the common-mode noise reduction due to the mutual coupling of adjacent layers. Further experiments showed that the clock system operates at frequencies at least 25 percent higher than the design goal over all combinations of bus configuration, voltage, and temperature.

The data lines exhibited more subtle problems. Our initial testing yielded results very similar to our design predictions. As sufficient hardware was assembled for a maximum configuration with heavy bus traffic, however, unexpected waveforms were discovered. The waveforms no longer exhibited the exponential shape of an RC time constant; instead, they resembled step functions with exponential risers. After due deliberation, we realized that, although the full time constant was fairly slow, the initial slope, dV/dt, was much faster. Therefore, its higher-frequency components traveled down the line and were reflected several times during the duration of an RC time constant, resulting in the staircase effect. SPICE simulations yielded an identical waveform when a transmission line, originally considered unnecessary, was included in the model. The overall timing was not affected by the reflections. Figure 4 shows this waveform with its staircase effect caused by incomplete termination of the transmission line.

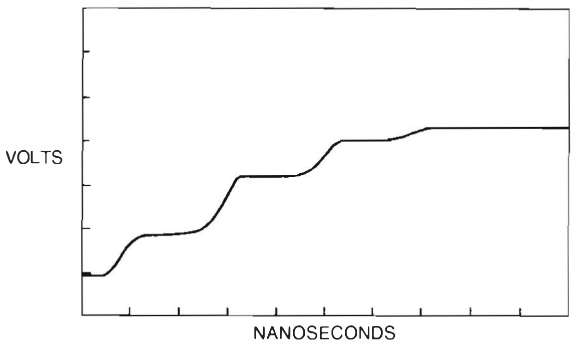


Figure 4 Simulated Waveform from SPICE

A second, more significant, effect was due to crosstalk, or coupling between the lines. To meet the capacitance budget, the original physical design aimed to minimize the capacitance to ground. An undesired result was that the mutual capacitance from line to line, while still small, became proportionally larger, thus increasing the coupling from line to line. The voltage on one line was affected by voltages on nearby lines: transitions were aided by like transitions and slowed by opposing transitions. In the worst case, the magnitude of this variation was as much as 24 nanoseconds.

This worst case occurred on a group of lines in close proximity to a "spare" line, not connected or terminated, which contributed additional mutual capacitance, thus enhancing the coupling. This spare line, included to reduce the need for engineering change orders to the backplane, nearly needed an ECO for its removal, which could have delayed several new products. However, a timing analysis showed that its removal was unnecessary. It should be emphasized that this effect was not visible until actual bus traffic, consisting of random data patterns, was being transferred on a large bus configuration. Test patterns were too small and too regular to show these significant effects.

Simultaneous switching noise, described above, was also investigated because its effect was similar to the effect of crosstalk. All VAXBI data signals except one were switched simultaneously, and the induced voltage was monitored on the remaining line, which was fixed in the high (inactive driver) state. Ground pins were then broken off one at a time, the voltage being measured after the removal of each pin. As a result the induced voltage increased from an insignificant level with 15 ground pins to more than one volt with only 3 ground pins remaining. With one more pin removed, the chip no longer passed self-test. These results showed that only a few ground pins are necessary for the chip to operate, but 15 are needed to prevent the addition of noise to the bus.

The timing analysis involved fabricating special lots of 78732 interface chips with the fastest and slowest possible process variations. From these lots chips were selected at the absolute specification limits. These chips were carefully measured in a range of configurations, including one beyond the specified limits. Then the timing margins were calculated over the specified range of operating conditions. When all possible worst-case conditions and the effects described above had been included, the calculated timing margin was reduced to 0.5 nanoseconds. Design verification testing on this worst-case system showed that it could still operate at a frequency 10 percent higher than that specified over the full operating range of temperature and voltage.

Summary

The VAXBI bus was designed to a rigorous bus-architecture specification. After minor adjustments during design verification testing, the bus met all the requirements of that specification. In particular, this testing proved that the VAXBI bus can operate independently of system configuration.

Several other points should be noted by bus designers for future products:

1. Designing a product to a rigorous specification, called top-down design, can really work.
2. Differential signals are recommended for critical timing. They are best located on the same printed-circuit layer on a module.
3. Testing should be performed on real hardware with real data, as closely as it can be approximated during the design process. Too often, the test patterns run on test structures yield nothing but the expected results. Testing should also reveal unexpected problems, not simply corroborate the design.

4. Ground return paths require careful consideration, particularly under conditions of simultaneous switching.

Acknowledgments

The following people were invaluable in the successful and timely conclusion of the VAXBI project: Dana Blanchard, Frank Bomba, Bob Chen, Norm Commo, Ron Desharnais, Rick Gillett, Glenn Herdeg, Bill Lin, Bill Schmidt, Jim Staples, Betty Ann Tyson, Bob Willard. Of course, the VAXBI bus would not have been possible without the contributions of the VLSI team responsible for the 78732 VAXBI Interface Chip.

References

1. F. Bomba, R. Chen, and R. Gillett, "General Purpose Bus Eases Interaction of Distributed Resources," *Computer Technology Review*, vol. VI, no. 2 (Spring 1986): 47–53.

2. *VAXBI Options Handbook* (Maynard, Digital Equipment Corporation, Order No. EB-27271-46, 1986).

3. R. Schumann and W. Parker, "A 32-bit Bus Interface Chip," *ISSCC Digest of Technical Papers*, vol. XXVII (February 1984): 147–148.

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

Michael W. Kement
Gerald J. Brand

A Logical Grounding Scheme for the VAX 8800 Processor

The treatment of ground as a signal conductor is crucial in achieving high-performance computer systems. The impact of system grounding on signal integrity becomes even more important as systems are connected into networks. For the VAX 8800 CPU design, the authors first identified the sources of ground-conducted noise from the four ground systems: the power and logic systems, and the safety and RF grounds. They then isolated and defined the ground elements in order to specify an interconnection strategy to guarantee the CPU's performance. Then the I/O subsystem grounding was established and finally a system-to-system grounding scheme was completed.

The design of the ground interconnection is often given little attention in system design, at least until it becomes crucial to system performance and program development schedules. The treatment of this interconnection as a signal conductor greatly affects the electrical noise levels. Ultimately, these noise levels are a critical factor in limiting the maximum clock speeds and thus machine performance.

Field service personnel have long recognized that many installation problems result from the subtleties of grounding when cabling together CPUs, mass storage devices, and peripherals. Particularly difficult problems occur when equipment comes from different vendors. The traditional approach to solving these problems has been to dispatch a seasoned field service representative to the site with an assortment of ground straps and other parts. Given the injunction to "make it work," he could, with enough ingenuity and customer patience, bring about satisfactory results.

As a consequence, early in the development cycle the VAX 8800 project team set a high priority on the logical design of the ground system. We knew that the 8800 would be used in large networks, thus intensifying any problems with ground-conducted noise. In fact, the inclusion of the backplane interconnect, called the VAXBI bus, ensured that many I/O ports with high bandwidths would exist in close electrical proximity to the logic backplane. Moreover, many of the applications targeted for the product would preclude its installation in the controlled environment of a computer room, with its traditional massive copper grounding grid beneath a raised floor. The system components would be connected for the first time at a customer's site. Our goal was to require minimum site preparation efforts; system components were designed to be cabled together in a "plug-and-play" manner.

These product goals, coupled with the EMI/RFI and system safety requirements of the international regulatory agencies, required an integrated system philosophy for grounding and shielding. The approach that we followed on the VAX 8800 project involved three separate but interrelated steps:

First, we identified the sources of ground-conducted noise within the VAX 8800 and devised ways to reduce that noise to the lowest practical level. Next, we identified the interconnections within the ground networks and connected them in ways that controlled the ground noise. There are four ground networks:

1. Power return
2. Logic return
3. Safety, or ac power-fault ground
4. Radio frequency shield and chassis ground

Finally, we extended the concept of system ground in the VAX 8800 to large-system applications and computer networks in an effort to ensure optimal overall system performance. In the majority of cases, these networks involve mature products for which it is difficult to make any internal configuration changes.

Ground Conducted Noise

Power System

The VAX 8800 power system consists of modular units of switching power regulators operating at 50 kilohertz (KHz). The total three-phase ac power required for a typical application configuration is about 5 kilowatts (KW). The hardware implementation uses units from a family of products called the Modular Power System, or MPS, designed by Digital. These units yield low and tightly controlled differential (normal mode) noise levels for the dc power that supplies voltages to run logic.

Through their high electrical efficiency of power conversion, such switching power systems have made possible the small sizes and low weights of present computers. This power circuitry, however, has current spikes (dI/dt) as high as 1000 amperes per microsecond (μs) and voltage slew rates (dV/dt) as high as 2000 volts (V) per μs. These high slew rates, a consequence of the pursuit of high efficiencies, can produce significant noise problems. The rest of this section discusses five of the most important noise sources that we identified and resolved in the power system.

Noise Currents

When high-voltage slew rates are present across parasitic capacitances (i.e., unintentional capacitance that is present as a consequence of a physical metallic structure), a noise current I_n will be generated:

$$I_n = C_p \, dV/dt$$

in which C_p is the parasitic capacitance.

One significant source of common-mode noise in the MPS regulators is the parasitic capacitance between the primary windings in the high-frequency power transformer and the solid-foil safety shield between the primary and secondary windings. The use of this shield, connected to a sheet-metal "safety ground," is one way of complying with the international safety regulations.[1]

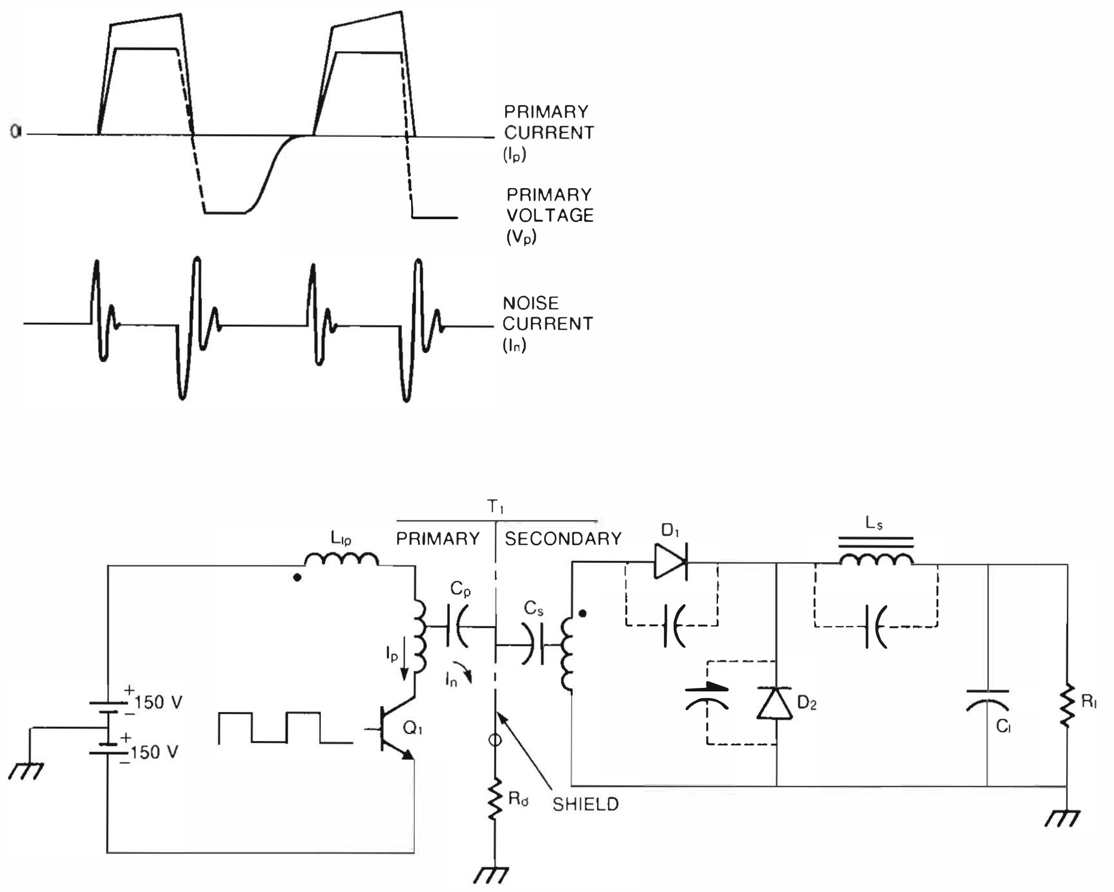
During normal switching-converter operation, voltage pulses with rise times of approximately 1000 V per μs are applied to the primary. These pulses cause capacitively coupled noise currents with peak amplitudes of approximately 200 milliamperes to be sent into the system chassis, or safety ground. Figure 1 shows a schematic representation of this process. The parasitic leakage inductance associated with the primary winding comprises a series-resonant circuit with the shield capacitance. This noise current has a decaying exponential waveform with a frequency in the range of 5 to 10 megahertz (MHz) and a repetition rate of twice the switching frequency. Since many power converters are used in the VAX 8800 system and they are all synchronized to a common clock, the noise currents tend to add. Current amplitudes as high as 2 amperes were observed.

The most practical way to reduce this noise source was to insert a damping resistance, R_d, that would reduce the Q of this resonant circuit at the specific frequency range. Q is traditionally defined as the ratio of reactive impedance to resistance, and represents a measure of resonant efficiency. The international safety regulations, however, strictly limit the fault-current impedance in this path. To meet both requirements, we inserted a ferrite bead on the shield ground lead. This bead is made of ceramic ferromagnetic material that is electrically lossy. It acts as a small inductance at low frequencies and as a nearly pure resistance at high frequencies. The bead does not block the fault currents from a short circuit but does reduce the noise current to the desired level. The noise amplitude is reduced by two to four times and the ring frequency reduced to about 1 MHz. Thus a potentially serious cause of common-mode noise current in the system is reduced at the source to acceptable levels.

In new designs, more effective schemes involving different shield configurations and interconnections could be employed.

Power Line Filter

One of the more subtle (and ironic) sources of common-mode noise current originates in the power filter designed to reduce the electrical noise emanating from the power line. Figure 2 depicts a schematic of a typical line filter,



$L_{lp} = 1.2 \times 10^{6}$H primary leakage inductance

$C_p = C_s = 200 \times 10^{-12}$ picofarads primary and secondary parasitic capacitance to shield

R_d is the damping resistance provided by a lossy ferrite bead

Resonant frequency of I_n is $F_0 = [2\pi\,(L_{lp} \times C_p)^{1/2}]^{-1} = 10.3$ MHz

Resonant impedance $R_o = (L_{lp}/C_p)^{1/2} = 775$ ohms

With $R_d = 0$, I_n (peak) = V_p (peak)/R_o = 200 milliamps

With R_d = 500 ohms @ 10 MHz, I_n (peak) = 118 milliamps

Figure 1 Parasitic Capacitance of the Power Transformer

including the parasitic, or leakage, inductance of the common-mode choke, L_l. The "Y" capacitors, C_y, are connected from either side of the power line to the chassis, forming a high-Q resonant circuit with this leakage inductance. The load current for this power filter is dominated by the discontinuous current pulses of the switching power converters, which provide excitation for this resonant circuit. The result is a resonant current pulse into the chassis with each half-cyle of current in the power line.

Other considerations of signal integrity demand that an inductor be placed in series with the power ground wire in the filter before that wire is connected to the chassis. The resulting ground impedance forces the resonant common-

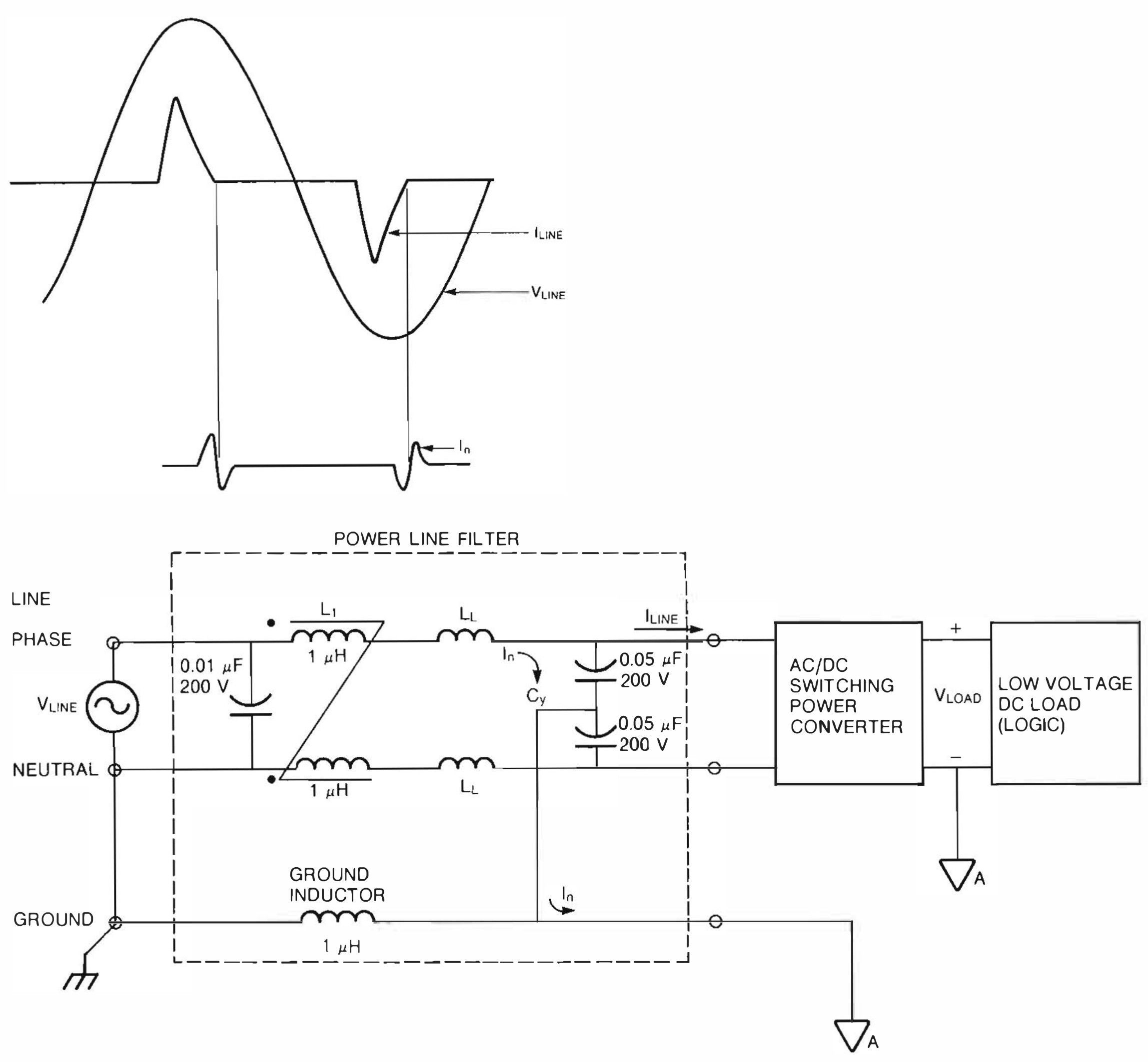


Figure 2 Power Line Filter

mode current to flow through the chassis of the system, probably through the logic returns. If the filter design has taken this parasitic resonance into account, a series resistor or ferrite bead, R_y, may be added to lower the circuit Q. That reduces the common-mode current at the expense of filter attenuation.

In the case of the 8800, many of the system components had been designed and released before this problem was fully appreciated. Therefore, our only viable strategy was to segregate this noisy ground by separating the logic returns and chassis grounds to the greatest degree possible.

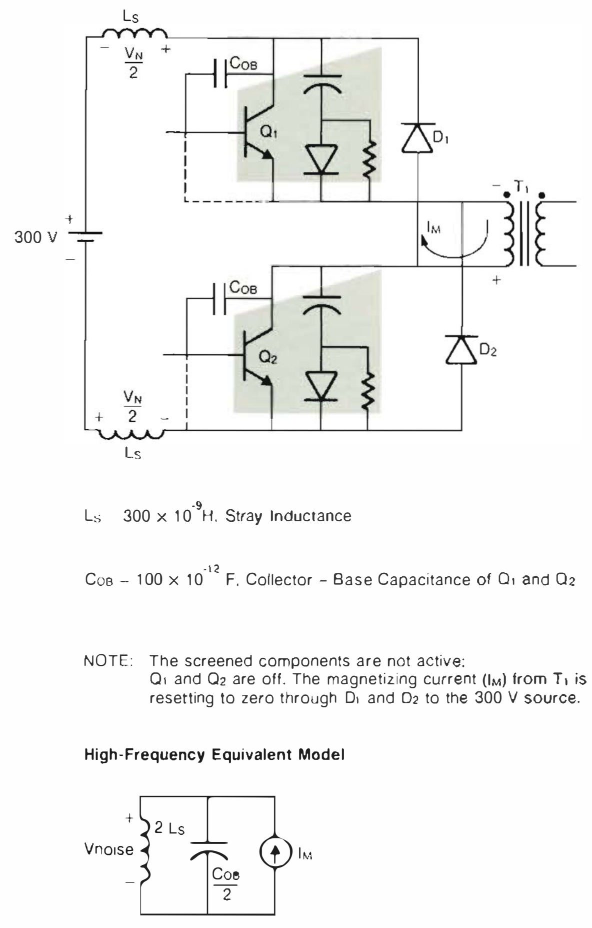
Noise Voltages

The electrical dual of the noise source just described is the generation of noise voltages across both real and parasitic circuit inductances when rapidly changing currents flow through them. This noise voltage is expressed as

$$V_n = L_p \, dI/dt$$

in which L_p is the value of inductance.

The most common source of noise voltage in switching power converters is parasitic inductances excited by the rapid rise and fall of current in the transistor power switch and by the reverse charge recovery in the rectifier diodes. These abrupt transitions between the conducting and nonconducting states generate a very high dI/dt. For example, the primary reset diodes (D_1 and D_2 in Figure 3) in the MPS converters have very fast switching times of 30 to 50 nanoseconds (ns). As the diode current rapidly goes to zero when the switch is turned



L_S 300 x 10^{-9} H, Stray Inductance

C_{OB} – 100 x 10^{-12} F, Collector – Base Capacitance of Q_1 and Q_2

NOTE: The screened components are not active: Q_1 and Q_2 are off. The magnetizing current (I_M) from T_1 is resetting to zero through D_1 and D_2 to the 300 V source.

High-Frequency Equivalent Model

Figure 3 Parasitic Inductance of the Power Switching Stage

off, the circuit parasitic inductance will ring with the capacitor in the switch-protective snubber, C_s. The frequency range will be from 10 to 30 MHz for typical circuit values. The result is a differential noise voltage at the converter output.

Our solution to this noise voltage source was to install an appropriate ferrite bead on the diode lead to damp the oscillations in this frequency range.

Radiated Magnetic Flux

A substantially more difficult problem is caused by rapidly changing magnetic fields that radiate from the high-current secondary circuits in the power converters. The output rectifiers can be conducting as much as 200 amperes when they switch off; the resulting dI/dt can easily approach 1000 amperes per microsecond. As the current dies, the magnetic field surrounding the secondary windings of these high-current conductors will collapse. That induces a voltage in other conductors enclosed by this magnetic flux. According to Faraday's Law, this noise voltage is

$$V_n = N\, d\varnothing/dt$$

in which N is the number of turns in the other conductors, and $d\varnothing/dt$, which is proportional to dI/dt, is the rate of change of magnetic flux. It is quite possible to develop volts of noise across 2 inches of circuit board etch or a sheet-metal panel through this effect.

The original designs of the MPS converter tried to minimize this noise problem by making the high-current loop areas as small as possible, thus minimizing the radiated magnetic flux. In addition, copper Faraday shields and ground-plane circuit boards were used. In spite of this care, we encountered problems with circulating currents induced in the mechanical support structure in the VAX 8800 system design. As with the power-line filter, we could not reduce the noise at its source. Therefore, the only viable solution was to take great care with the chassis ground connection of these structures so that the noise currents are directed away from sensitive circuits.

The Logic System

A significant source of noise within the logic system is the energy radiated from the interconnect cables from the I/O bus to the disk controller. This noise radiates at a fundamental frequency of about 47 MHz. The bus itself is a high-speed, mass-storage parallel interface. The interconnect cable is composed of individual coaxial signal pairs that are transformer coupled and driven differentially. However, the impedance from the coaxial center conductor to the outer overall shield is slightly different from the impedance from the coaxial shield to the outer shield. That is, both signal conductors do not have equal impedances to the outer shield, which is grounded to the chassis at each end. The result is a net noise current that flows on the outer shield. Within the VAX 8800 processor, this current can couple into adjacent cables.

The only practical method to minimize this noise coupling was careful routing and dressing of the interconnect cables relative to other communication and power cables.

VAX 8800 System Grounding

This section describes the types of ground structures present in a large system like the VAX 8800 multiprocessor. As such a computer system expands in size and complexity, its ground connections also expand and their interrelationships grow in complexity. To appreciate the grounding scheme as a total system, the various components must be isolated by function and location. In that way the ground system can be broken into its constituent elements. The individual components can then be viewed as functional blocks that require interconnection.

Although a designer can choose how to interconnect the ground elements, he is always constrained by the existing international regulations in the implementation of the grounds.

Types of Ground Topologies

There are three choices of ground interconnection topology: single point, multipoint, and hybrid. The single-point ground looks like a wagon wheel with the ground in the center and the other devices connected radially around the hub. That center becomes the absolute ground point, called the zero-voltage potential reference, for all devices. Multipoint grounding has each device individually connected to a single ground plane, all of which is at the same zero-voltage potential. The hybrid is some mixture of the single-point and multipoint topologies in which interconnections are made based on the characteristic needs of the subsystem functional elements.

The single-point topology is not practical to implement on a large system like the VAX 8800. The physical distances and associated impedances of the interconnects begin to dominate so much that an absolute ground point does not really exist. The multipoint ground requires a ground plane, or grid, to be effective. Again, in a large system, it is not practical to implement a ground plane into the physical layout. The hybrid scheme has advantages over the other two, but it requires a detailed evaluation of the characteristics of each subsystem element before an interconnection can be designed. That was the approach we followed in designing the interconnection for the different ground types in the VAX 8800 system.

DC Power Return

The dc-to-dc converters in the system required a dc current return that presented a low impedance through the frequency range of dc to 200 KHz. Our primary consideration was to specify a conductor with a sufficiently large cross-sectional area to keep the IR losses and heating effects to a minimum. A secondary consideration — often overlooked — was to minimize the physical distance between the current feed and the return. In a large system the currents involved can exceed 400 amperes. The resulting flux can produce a large magnetic field. This field is determined by the relationship

$$\text{Magnetic Flux} = I \times \mu \times A/l$$

in which I is the current, μ is the permeability of air, and A the area and l the length of the conductor. These leakage fields can couple into adjacent devices, sheet metal, and cables. If the flux has an ac component, a current may be induced in adjacent conductors, as described earlier.

A power supply in the MPS series used in the 8800 has a silver-plated bus as its main output. That bus is mated to a large connector that is mechanically mounted on the power backplane. This connector is soldered to multiple epoxy-coated copper strips that are 0.050 inch thick by 2 inches wide. These strips are fusion welded to a horizontal bar that is bolted to the inner layers of the CPU backplanes. The supply and return straps are overlapped to minimize parasitic inductance and its consequent radiated magnetic flux. The flat, wide geometry of the connection is essential to minimize that flux. (See Figure 4.) Minimizing this stray inductance is also essential to obtaining rapid power-system response to load transients with adequate stability (phase margins).

Logic Return

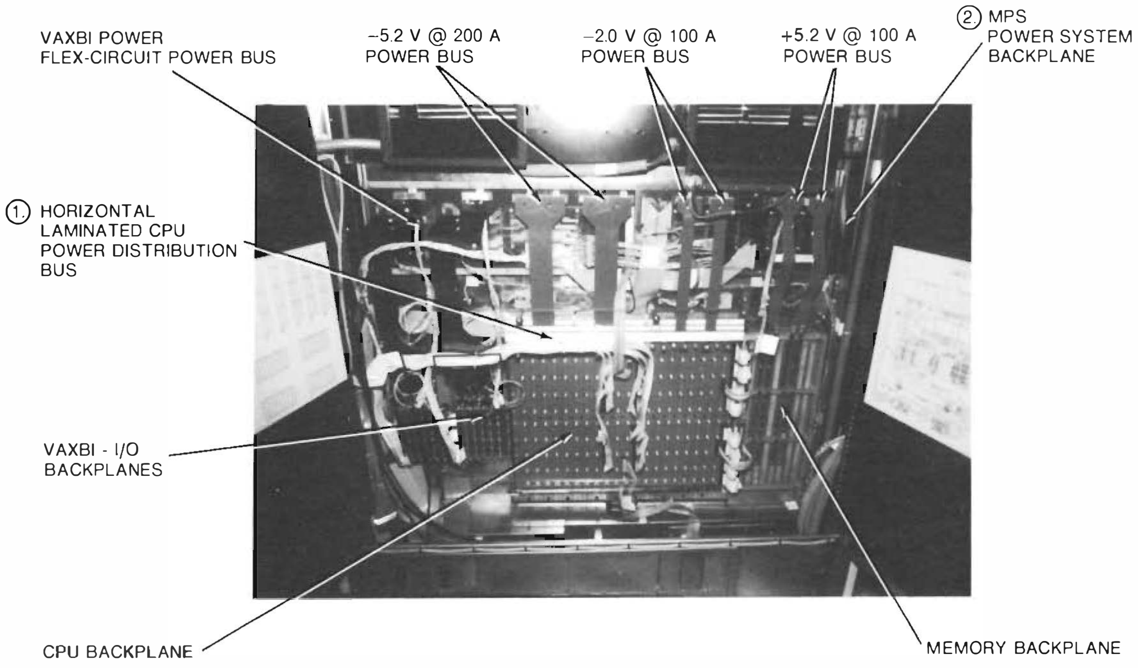
The logic return provides a common signal reference for the logic within the system. To minimize noise this reference must be designed with a low impedance at the frequency corresponding to the logic switching speed. With logic operating at rise times of 1 V per ns, or 300 MHz, this reference is considered to be a radio frequency (RF) ground and thus can be modeled as a frequency-dependent impedance. The ground impedance at these frequencies is dominated by the depth of penetration of current into the conductor. The magnetic field surrounding the current forces the density of current t● decrease from the surface value as the depth into the conductor increases. In the limiting case, as frequency becomes very high, the current will flow as a sheet of charge at the surface. The result is a steadily increasing real component of impedance (resistance) with increasing frequency. The point at which the current density decreases to $1/\epsilon$ of the surface magnitude (approximately 37 percent) is one "skin depth."

Therefore, the first step in calculating the ground impedance is to derive the skin depth, in meters, as follows:

$$\text{Skin Depth} = 1/\sqrt{\pi \times F \times \mu}$$

in which F is the frequency in Hz and μ is the permeability of air in siemens per meter. For example, for copper, the skin depth is $0.0666/\sqrt{F}$ in meters. After the skin depth has been determined, the impedance at the frequency of concern can be found using the sheet resistance of the material. The specific resistance, R, is equal to $p \times L/A$, in which p is the specific resistance of the conductor, L is the inductance, and A the area. For copper, p equals 1.673 microohms per centimeter.

Another major factor in designing a ground plane is the voltage drop across the ground layer at low frequencies (dc to 1 KHz) as the total load current is sent from the logic modules. This voltage drop produces an offset in the logic threshold from module to module that affects the noise margins, or tolerance. The voltage drop is a function of the sheet resistance of the ground layer (directly proportional to the thickness) and the method of termination of the ground layers to the return buses. The connection geometry must be chosen to ensure a safe



NOTES:

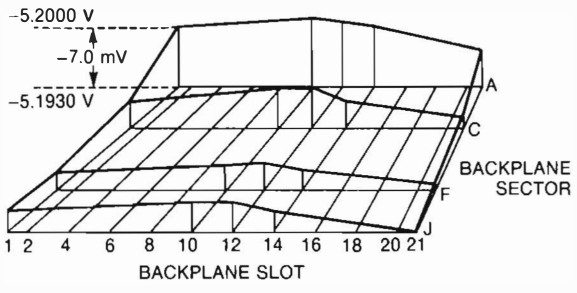
1. The return, or logic ground rail, is connected along its entire length to the system chassis and represents the system single-point connection of RF (chassis) power and logic ground.

2. MPS regulator rack is electrically isolated from chassis ground and connected through lossy RF chokes.

Figure 4 Logic Power Distribution System

maximum current density through the ground layers. Current crowding, particularly at the connection points and plated through-holes, can turn the backplane into a toaster oven.

We used the inner layers of the CPU backplane as the logic reference for the VAX 8800 CPU. There are four ground layers, each 0.003 inch thick. Figure 5 illustrates the dc voltage-potential drop as a function of geometry across the CPU backplane. The return current is approximately 500 amperes; therefore, this CPU backplane was the most challenging part of the design.



NOTE: Measurements were made from corresponding local points on the ground plane. It demonstrates the excellent control over voltage drops provided by the internal ground and power planes of the multilayer CPU backplane. Maximum current available to these −5.2 V inner layers is 400 amps.

Figure 5 Distribution of the Backplane Voltage for the −5.2 V Power Plane

AC Safety Ground

The primary function of a safety ground is to provide a low impedance at 60/50 Hz, thus allowing fault currents to follow a path with a low IR drop. The design and implementation of this path is strictly controlled by the international regulations, to which all other uses of this ground must comply. The safety ground also acts as a signal ground in that it connects products to the ground grid of the building housing the system. This connection can be detrimental to the system's I/O signals. Thus it is advantageous to add an impedance whose magnitude is frequency and current dependent in series with the safety ground. A saturating inductor meets those requirements.

For a fault condition, Digital's internal design standards require that a current of twice the product's receptacle rating flowing through the safety ground system must not result in a voltage rise of more than 4.0 V, and this level must be sustained for 10 minutes. With these requirements in mind, we used a 1.2-millihenry choke to isolate the VAX 8800 CPU from the building ground at high frequency. This choke was designed to saturate as described above if a fault occurs.

Chassis Ground

The RF shield comprises the chassis ground and the outer panels of the cabinet. The federal regulatory agencies (FCC and VDE) set and enforce the allowable limits of radiated emissions from computer equipment. Since the integrated circuits within the system are switching at high frequencies, they can be modeled as RF sources. The interconnecting etches between integrated circuits that are not tightly coupled to a ground layer can be modeled as antennas.

The faster the clock and edge speeds, the shorter the antenna needed to act as an effective radiator. The length, in meters, of a full wavelength is defined as $3 \times 10^8/F$.

Once this wavelength has been found, the outer panels of the cabinet can be modeled as an attenuator, which decreases the amount of radiated energy that can be transmitted from within the cabinet. To maintain this level of attenuation, all openings, such as doors, must be bridged with conductive gasketing or finger stock. The openings for air flow must be treated as a wave guide. The attenuation, in decibels, of the opening is related to its size by the following formula:

$$.0046 \times l \times F \times \sqrt{5900 \times F/\text{gap}^2 - 1}$$

in which F is the frequency in MHz, and l is the length and gap the width of the opening, both in centimeters.

Ground Interconnections within the System

Once the separate ground elements had been defined, we began to formulate an orderly interconnection strategy for the main computer that would not compromise the system's performance. We used the same return path for both the logic and the dc power because there was no dichotomy in the requirements for both returns. In the VAX 8800, the junction of these returns comes at the point where the horizontal bus bar (return) is bolted to the inner layers of the logic backplane. (See Figure 5.)

Digital's internal standards, which meet all the applicable international regulations, mandates that the dc power return be connected to the safety ground. This connection must be able to withstand the short-circuit current of the dc regulator output. (In certain cases it may be desirable to insert a frequency-dependent impedance in series with this connection to "isolate at frequency" an element of the system. That could be done when creating a single-point ground system — directly referenced to the chassis — or a controlled hybrid-ground system.)

In the VAX 8800 CPU, the dc output could, under fault conditions, produce approximately 400 amperes. Thus the interconnection must handle this high fault current. This interconnection was accomplished by bolting the junction node of the combined dc-power and logic return to the chassis for the entire length of the horizontal bus bar. This portion of the chassis was chosen as the connection point because it was not used as a conductor for any other high-frequency currents.

In summary, the grounding approach we used for the 8800 featured the following design points:

- The logic and dc return and the chassis ground are connected together at the horizontal power-return bus.
- The power-system outputs and the chassis ground are isolated from ground at RF frequencies by high impedances using lossy ferrite inductors. DC currents and line-frequency (50/60 Hz) fault currents may thus flow unimpeded.
- Particular care was taken to minimize the flow of logic-return currents through the system chassis, thus isolating the peripheral boxes (CI750, BA11AW, etc.) from the system chassis ground. Insulated chassis slides, shunted by lossy ferrite inductors, accomplished that isolation. Although there are still common-mode currents with the ferrite inductors, they reduce unwanted common-mode noise voltages that can couple into circuits through parasitic inductances. That is a far worse problem, as we demonstrated to our own chagrin.
- The I/O panel bulkhead and the logic and power returns for the VAXBI bus and memory backplanes are tightly bonded to the single-point ground at the CPU power-return bus.
- The elimination of circulating noise and logic currents through the chassis will maximize the effectiveness of the shielded cabinet as an attenuator of radiated energy.

The implementation of this approach is shown in Figure 6.

I/O and Expansion of Grounding

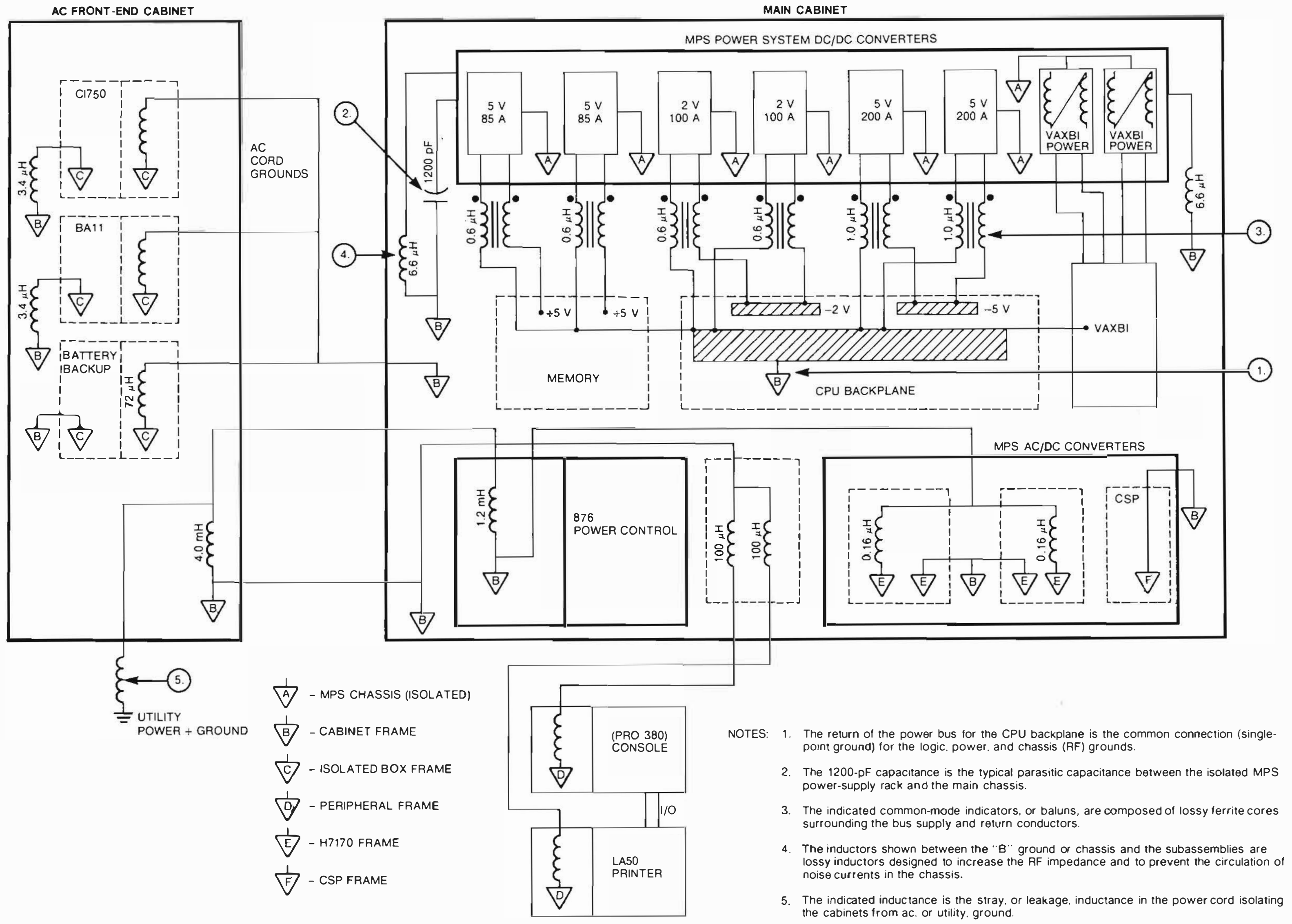
Once the main processor's grounding had been defined, we had to deal with grounds between the external elements, such as the I/O subsystem. The VAX 8800 system can accommodate a large array of I/O devices by utilizing the VAXBI architecture. The H9652 EC-ED cab has provisions for two expansion boxes, the CI750 and the BA11AW. These boxes are self contained and have integral power supplies, logic backplanes, and interconnects. In keeping with our grounding architecture, we isolated these boxes from the chassis ground by using low-Q inductances. The signal/logic ground was then established by means of cables to the VAXBI-to-CPU backplane. This scheme ensures that the chassis is not used as a signal/logic return.

System to System Grounding

Grouping systems together or networking them has a large impact on system noise and the subsequent grounding techniques to eliminate it. In terms of the signal-to-noise ratio and from the aspect of grounding, a networked system can be divided into two cases: the dense network, and the dispersed network.

Dense Network

A dense network is a group of computers or systems with associated support hardware that is located within one area, either an office or a computer room. This area is likely to contain systems from different vendors as well as phone-switching networks, experimental equipment, or industrial controllers and monitors. All these devices share a common ground that could be a grid or simply a branch ground as part of their safety ground. This connection also provides a signal reference between interconnecting devices in the area through the chassis and



NOTES:

1. The return of the power bus for the CPU backplane is the common connection (single-point ground) for the logic, power, and chassis (RF) grounds.
2. The 1200-pF capacitance is the typical parasitic capacitance between the isolated MPS power-supply rack and the main chassis.
3. The indicated common-mode indicators, or baluns, are composed of lossy ferrite cores surrounding the bus supply and return conductors.
4. The inductors shown between the "B" ground or chassis and the subassemblies are lossy inductors designed to increase the RF impedance and to prevent the circulation of noise currents in the chassis.
5. The indicated inductance is the stray, or leakage, inductance in the power cord isolating the cabinets from ac, or utility, ground.

Figure 6 System Ground Schematic

power line ground in a complex way. All these devices can generate high-frequency currents that flow into the ground. These currents must flow through the complex impedance of the grid where, consequently, RF voltages can develop. Under those conditions the ground would act as a noise injection point rather than a stable reference.

Dispersed Network

The dispersed network is an interconnection of computers or systems spread over a wide area, perhaps residing on different floors of a building or in different buildings altogether. Communication on this scale cannot depend on a mutual RF ground because it cannot be reasonably established. In this case, communication must be accomplished by means of either transformer-coupled circuits, optical links, or differential driver/receiver logic.

Both types of networks illustrate the fact that system networking cannot, and in some cases should not, be accomplished by attempting to create an absolute ground reference to the network.

System to Peripheral Grounding

As a system expands with the addition of peripheral devices, such as disk drives, printers, and LANs, the ground system must be viewed as a large hybrid arrangement. Interconnecting these devices must be predicated on the ground-current characteristics (signature) and the I/O connections of these devices to the system.

This signature is particularly important when connecting devices that were designed to be used as small, standalone applications. Their designs may have involved decreased line-filtering capabilities and minimally sized chokes for ground isolation or perhaps none at all. It is imperative that such factors be considered when connecting peripheral devices to a large system.

Summary

We now offer some conclusions based on our recent experiences with the VAX 8800 and other new systems. These conclusions take the form of recommendations for minimizing noise-related problems in any computer system.

Ground Noise Current Signature

It is important to identify the spectrum of ground-conducted noise for each subsystem element. This noise depends on parasitic elements in the circuits and electromechanical structure. Therefore, this information is best obtained empirically by measurements on the actual hardware. The noise current amplitudes and fundamental frequencies should be measured on cable shields, chassis grounds, I/O logic returns, and power inputs.

Segregation of System Ground Networks

A ground system schematic should be developed for each particular subsystem. The interconnection of ground types will be based on the intended system application. As a general rule, the ground types should be segregated to account for the finite amplitudes and often unpredictable paths of the noise currents. This segregation of grounds (e.g., power, chassis, and safety grounds) can be accomplished by carefully choosing the frequency-dependent impedances. These impedances are lossy ferrite inductors placed in series with the appropriate ground connection.

Appropriate Signal and Power Interconnect

The optimal signal interconnections are designed as controlled-impedance transmission lines with each signal and its return path closely coupled and having equal impedance to the chassis ground. Depending on the noise sensitivity, data rate, and interconnect length, the implementation can range from coaxial cables with overall shields to ground-plane ribbon cables to ribbon cables with alternate ground/signal pairs. Even the crudest, slowest signal line that relies on chassis ground for a signal return is doomed to failure if it is sensitive to noise.

High-performance data lines should certainly be designed with low-impedance differential line drivers and receivers, either directly coupled or transformer coupled. Single-ended line drivers and receivers may be acceptable within a subsystem in which the noise between grounds is low and controlled. Communication through unbuffered TTL outputs and inputs are never acceptable when leaving a subsystem backplane.

The initial cost of and board space needed for proper line drivers and receivers are more than justified in today's distributed computing environment. Their use increases reliability and decreases start-up problems. The power interconnects should be designed with minimum inductance and the lowest high-frequency characteristic impedance that is reasonable. The circulating path of supply and return power currents should be kept as low as possible. This design allows better power-system transient performance and ensures the existence of minimal radiated magnetic fields.

Notes

1. A short circuit between the high-voltage primary and the low-voltage secondary could produce lethal voltages referenced to the chassis ground at accessible points within the computer. With this shield, however, the short will produce a high fault current to the chassis. That current will open various protective devices, such as fuses and circuit breakers, that render the system safe in the event of a fault.

Appendix

Determining Skin Depth

To calculate the impedance of a given conductor, the depth of current penetration — or skin depth — in a conductor must be calculated first. To do that, a designer must perform the following steps:

1. Determine the type of metal of which the conductor is made (i.e., copper, zinc, etc.).
2. Look up in a reference table the magnetic susceptibility of the material. (The *CRC Handbook of Chemistry and Physics* contains tables of this nature.) Two types of listings of susceptibility are commonly used. The first type gives values of specific susceptibility that must be converted by multiplying the value by $4 \times \pi \times$ density of material, called *P*. For copper, this value would be $-0.086 \times 10^{-6} \times 4 \times \pi \times 8.89$, which equals -0.960×10^{-5}.

   The second type uses susceptibility in one gram formula weight. This value must be converted by multiplying it by $4 \times \pi \times$ density of material or molecular weight, which for copper would be $-5.46 \times 10^{-6} \times 4 \times \pi \times 8.89/63.54$, which equals -0.960×10^{-5}.
3. The resulting figure must now be converted to relative permeability by adding 1.0 to the susceptibility factor. For copper, this value would be $1.0 - 0.960 \times 10^{-5}$, which equals 0.9999904.
4. The relative permeability must be converted to permeability by multiplying the value from step 3 above by the permeability of air ($4 \times \pi \times 10^{-7}$). For copper, this value would be $0.9999904 \times 1.25663 \times 10^{-6}$, which equals 1.25662×10^{-6}.
5. The next piece of information needed is the conductivity of the material used. This value must be in the form of siemens per meter, although most listings will be in ohms per centimeter. To convert, multiply the table entry by 1×10^{-2} and then take the reciprocal. For annealed copper, this value is $1/1.7241 \times 10^{-6} \times 1 \times 10^{-2}$, which equals 5.8001×10^{7}.
6. The skin depth can then be determined by the relationship $1/(\pi \times$ frequency of concern $\times$ conductivity $\times$ permeability$)^{1/2}$. The result can be manipulated to the form of $1/(\pi \times$ conductivity $\times$ permeability$)^{1/2}/($frequency of concern$)^{1/2}$. For copper, this value is $1/(\pi \times 5.8001 \times 10^{7} \times 1.25662 \times 10^{-6})^{1/2}$, which equals $0.06608/($frequency of concern$)^{1/2}$. For example, if the frequency of concern were 1 KHz, then the skin depth would be 2.089×10^{-3} meters, or 2.089 millimeters, deep.

   If the frequency of concern were 50 KHz, then the skin depth would be 295 micrometers.

Cheryl A. Wiecek

The Simulation of Processor Performance for the VAX 8800 Family

An effort was initiated in the fall of 1981 to simulate the performance of the processor design for the VAX 8800 family of computer systems. That simulation stayed current with the changing design and continues to be used today for studies associated with developing VAX processors. This paper discusses why this simulation was done, how it was structured, and what was simulated. Since the results generated are quite extensive and detailed, only the conclusions from these studies are presented here. What was learned from the model and how it affected the processor design are particularly emphasized.

Many levels of simulation are done within processor development projects well before any actual hardware is built. Structural models at the circuit and gate levels are used in tasks such as verifying timing and developing diagnostic tests. Behavioral models at the function level are useful for verifying processor instruction microcode. Another useful class of models simulates performance at the microcycle level. Such models look at a processor's design as a collection of hardware resources that must be managed. These models are most useful for gathering design trade-off information and verifying the design performance estimates. By emphasizing the key hardware resources and how they interact, performance simulators can

- Focus on how those resources are being used
- Indicate how well they support the required activities
- Provide a high-level view of the interactions in the processor system

This paper describes the performance simulator used on the project that developed the VAX 8800 family of computer systems. This modeling project began in the fall of 1981, and the simulator continues to be used today to study alternatives for new VAX processor designs. The following two sections discuss how the simulator was designed and what was simulated. The third section highlights the results and discusses what was learned from them.

Methodology

The overall structure of the performance model mirrors the structure used previously for the performance simulation of a PDP-11 processor design.[1] The model contains three parts, all developed as separate entities:

- The instruction stream that is acted on by the processor resources
- The microcode that directs instruction execution
- The simulation of the processor resources and timing

These three parts are then combined to generate simulation results. The tasks performed to develop each part are discussed in the following section.

Workload Model

The most appropriate model for the workload fed to the simulator is the streams of VAX instructions from typical programs being executed. Information about each executed instruction is required to obtain performance data at the microcycle level about the processor and its resources. The software used to extract these execution streams had already been developed from a previous project. That software is essentially a debugger that uses the VAX T-bit to generate a software trap after the execution of each instruction in the traced program.[2] That tracing permits the collection of the next instruction's operation code, the addressing modes and registers of the operand specifiers, the read and write references, and the operand values.

The task of choosing which programs to trace was bounded by a number of requirements and constraints. One requirement was to provide some initial performance estimates for the VAX 8800 family processor. Those estimates emphasized integer, logical, and floating-point operations in CPU-intensive programs. Another requirement was to select programs that exercised the processor resources that we wanted to model, especially the cache subsystem, where capturing best-case, typical, and worst-case scenarios was important.

All the constraints involved the programs from which instructions were traced. A reasonable length for these programs was about one-half million VAX macroinstructions, thus permitting the simulator to process them in a reasonable time. We avoided programs that required extensive microcode characterization for instructions that were either less frequently executed or too complex, such as those in the packed decimal group. Moreover, the trace software was limited to processing executing programs that ran in nonprivileged user mode. Thus we had to avoid programs, such as editors, having extensive operating-system service calls, which could only be partially traced.

We chose six programs to drive the model. These included four benchmarks and two popular utilities for creating executable images on VAX systems. The number of iterations in the four benchmarks was shortened proportionally, keeping the mix of instructions constant to retain their representativeness. Three benchmarks were written in FORTRAN: Towers of Hanoi, a prime-number generator, and single-precision Whetstone; one, called Puzzle, was written in PASCAL. The other two programs were a FORTRAN compile and a VAX/VMS link, both written in BLISS. For all their constraints, these programs exercised the model well. The accuracy of the performance estimates was confirmed later by measurements on a prototype machine.

Microcode Model

How microcoded instruction control is characterized has a significant impact on both the speed and results of a processor performance simulator. For example, creating a model at a very detailed level permits a finer analysis of the results, but takes a long time to develop and run. Therefore, we had to decide what the trade-off should be between time and detail. We also wanted to stay current with the latest developments in the processor microcode, which we knew would change significantly during the project. With all that in mind, we decided to use the latest version of the actual microcode sources as the input to a unique process, partially automated, that extracted the information needed by the simulator. This strategy allowed us to ignore details that were not required by the simulator, as well as to keep up with microcode revisions as they were released. A useful by-product of this approach was the ability to produce microPC histograms with the simulator. This information helped to explain how the microcode was being used.

One step in modeling the microcode is to determine the control fields that are key to the processor's performance. Only a small number of the defined fields are actually needed. Many microwords are effectively no-operation instructions for the simulated processor pipeline. Table 1 contains the microword key for the performance simulator. Each microword has three fields: SRC, ALU, and DST. In any microword, each field has a command subfield and up to three operand subfields. (The address operands generated by the trace software are actually extracted as both the traced program and the simulator are being run. The other operands and commands are extracted from the microcode prior to simulation execution.)

Before any actual microcode had been developed, simulated microwords were written manually from microcode flows provided by the group developing the firmware. Once the actual microcode was available, a significant portion of the performance simulation microcode was generated automatically by mapping real fields to the small number of fields that the simulator required. This automatic mapping of processor microcode to that used in the simulator was complicated by several issues.

One problem was that the microbranching logic required additional information at simulation runtime to decide which branch path to take. To solve that problem, the firmware group flagged microbranches by inserting comments in their microcode. Those comments were then caught by the microcode translation software,

Table 1 Microword Key to the Performance Simulator

Field	Command Description	Operands
Any	No operation performed.	None
SRC	Stall if the memory data registers (MDRs) specified by ASRC and BSRC are not yet valid for input to the arithmetic logic unit (ALU).	ASRC, BSRC
ALU	Send a cache arbitration signal and stall the pipeline if it is not the winner.	None
DST	Send the cache a read request for *x* Bytes starting at Address, and set MDR number to valid when the data is available.	MDR number, Bytes, Address
DST	Send the cache a write request with *x* Bytes of data starting at Address. The value of Signal determines whether hardware or microcode control sends the write buffer data to memory.	Signal, Bytes, Address
DST	Conditionally flush the IB and provide the cache with a new Address for prefetching IB data.	Address
DST	Send the cache notification of a new address for prefetching IB data once the decoder handles the IB-address page cross.	None
DST	Send the cache a read/write probe request.	None

which marked them for processing at runtime. Another problem was that some VAX macroinstructions had not been coded yet, and others were more complicated than required for simulation. (Many of the VAX floating-point instructions were in this category.) In those cases sequences of handwritten microcode were used.

Processor Simulation Model

The structure of the processor simulation model was driven by the need to provide timely answers to questions asked by the designers. The results had to be generated, verified, and distributed as quickly as possible to be most useful in design trade-off decisions. The requirements we considered most important were the following.

- The simulator must have a modular structure that facilitates replacing, reconfiguring, and reusing routines while minimizing the runtime overhead.
- A general-purpose control mechanism is needed to manage communication and synchronization between a number of independent tasks running in parallel.
- Extensive and flexible I/O features are needed to generate cycle-by-cycle traces and reports with simulated performance statistics.
- The ratio of simulated time to real time must not be a bottleneck to obtaining results.

We chose a structure that favored changing and reusing parts of the simulator, but which ran slower, over one that ran faster, but was hard to change. We did this knowing that the simulator would be used to try many design ideas that would eventually be discarded. The simulator also had many parameters built in so that different configurations and timings could be tried. The structure we chose could be used to evaluate many design alternatives. Since this was the first VAX processor to be modeled this way, we had to design and build all the software for the simulator; none of it could be borrowed from other projects. Therefore, we knew that producing results quickly would be difficult.

The structure chosen required that the simulated processor be partitioned into a number of independent components, each modeled by a deterministic state-machine. That machine defined the actions to be done when each state was entered, and the conditions to be evaluated for deciding the next state transition. This approach had several advantages. The hardware designers could relate easily to state-machine models of their particular designs, even though the states in the simulator sometimes marked performance-related events, not real hardware states. This structure also made it possible to replicate components and reconfigure the original single-processor version of the simulator into a dual-processor version.

A monitor is needed to control the communication, synchronization, execution, and status of these independent state-machine components. For communication between components, only certain types of send and receive operations are used. This restriction allows the component interfaces to be simple and well defined. There are three types of send operations:

1. A targeted send directs source information to a single destination within the current cycle.

2. A broadcasted send directs source information to zero or more destinations within the current cycle.

3. An arbitrated send directs source information to a single destination, stalling execution of the sending component until the information is delivered.

There are two types of receive operations:

1. A targeted receive results in the delivery of source information from a send operation.

2. A collection receive is limited to probing source information from a send operation; this information is used by the model to make decisions.

The monitor keeps two queues for the components: one for component send requests, the other for component receive requests. The monitor also synchronizes send and receive requests on behalf of the components and reports errors when undelivered send or receive entries remain in the queues.

Synchronization between components is achieved using the send, receive, and timing services built into the monitor. The send and receive operations allow the specification of a phase number so that components can send and receive information only at certain intervals within the basic microcycle clock recognized by the monitor. The monitor blocks components from executing while they wait for send or receive requests to be serviced. States within a component can be designated as time sensitive. When the next state to be executed within a component is so designated, that component is blocked from executing until the monitor increments the clock.

Execution proceeds on the basis of one machine cycle. State-machine components are chosen to execute, one at a time, starting at the state at which each was last left. Component execution continues until the required send, receive, or timing service returns control to the monitor. When all components have reached states in which no more activity is possible for the cycle, the monitor will increment the master clock and the execution of components can resume. End-of-simulation and detected-error conditions cause the monitor to generate a report of results by calling each component to execute its report code.

The complete model for the VAX 8800 family processor ran on a VAX-11/780 system and executed about six VAX macroinstructions per CPU second. That translates to a ratio of simulated time to real time of about 90,000 to 1. The control monitor was written in PL/I; the processor state-machine components were written using VAX assembler macros. Once the ADA language had been added to the list of VAX-supported languages, we translated the entire processor performance simulation model into that language. This new simulator is being used for follow-on processor performance studies. The ADA language was chosen because its multitasking features provide excellent support for the control monitor functions that we defined.

Verification of the Simulation Model

An important and often overlooked aspect of developing a performance simulation model is the effort required to verify that the model reflects the actual design. In the early stages of a project, the details of the proposed design are usually communicated by word-of-mouth. Continuous changes to that original design enlarge greatly the margin for error within a performance simulator. Since wrong performance data is counterproductive, a great deal of our effort went into verifying that the simulation operation and results accurately reflected the current state of the design.

Once the performance simulator produced results, the designers reviewed cycle-by-cycle traces of simulator activity to confirm that the simulator's operation matched the processor design. In addition, we developed a set of short tests that exercised certain key functions. These tests were rerun for each new version of the simulator, and the test results were exhaustively compared to those from the previous version. This procedure was effective in revealing unanticipated interactions and errors due to changes made in both the simulator and the design. As the design progressed, we were able to compare our simulation results with those from a behavioral model used for debugging microcode.

Eventually, we could compare our results with those from a working prototype system. Because the model tracked the design's evolution closely, these comparisons showed the performance model to be an accurate representation of the design.

Performance Model for the VAX 8800 Family Processor

This section describes the processor hardware resources that were modeled. For each modeled component, there is a short summary describing its function, the information communicated with other components, and the parameters that can be specified at runtime to control simulation configuration and timing. Although some information about the VAX 8800 family processor design is included, reference 3 should be consulted for more detail.

Figure 1 is an overview of the processor performance simulator used for the VAX 8800 family. The various components are represented by circles, the communication paths by arrows. As described earlier, each component is an independent state-machine that communicates with other components using defined send and receive operations.

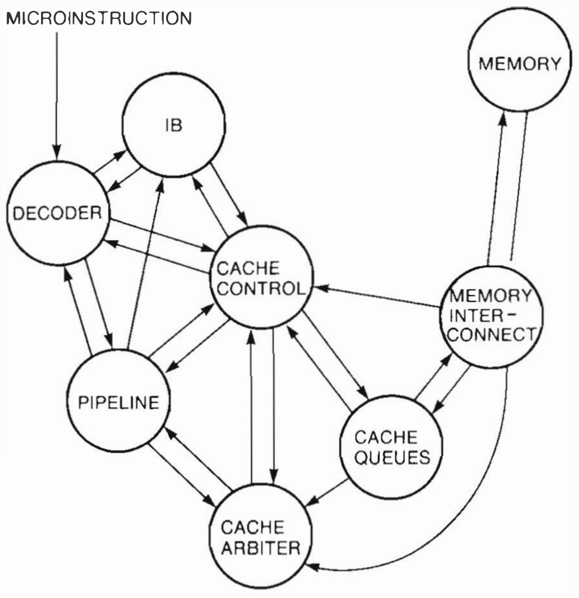


Figure 1 Performance Model for the VAX 8800 Family

Decoder

The decoder state-machine sends the pipeline a microinstruction during every unstalled cycle and detects the end-of-simulation condition. To do those actions, the decoder requests bytes from the instruction buffer (IB), using information provided in the instruction trace. When the IB indicates that the requested bytes are available, the appropriate microcode flow is chosen to start execution. If the IB cannot deliver the requested bytes, then no-operation microinstructions are fed to the decoder. The decoder must also communicate with the cache control. For example, the decoder must resolve any IB-address page crosses detected by the IB prefetch hardware in the cache. Also kept by the decoder is a parameter that controls the number of VAX instructions executed between cache flushes due to context switching.

Pipeline

The pipeline state-machine simulates how microinstructions provided by the decoder are to be executed. During any one cycle, parts of three consecutively queued microinstructions are processed:

- The DST field of the oldest microinstruction
- The ALU field of the next microinstruction
- The SRC field of the microinstruction most recently queued

For every cycle that the pipeline is not stalled, the oldest microinstruction is retired after the command in its DST field has completed. The actions performed by the pipeline are described in Table 1. The pipeline can send flush requests to the IB, and processor read and write requests to the cache (after arbitrating and winning it). The pipeline also manages the validation of the memory data registers (MDRs). Pipeline stalls that result from those actions are made known to the decoder. The only pipeline parameter the user must enter is the cycle time in nanoseconds, used for calculating performance data at the end of simulation.

Instruction Buffer

The IB state-machine simulates a first-in, first-out (FIFO) cache for VAX instruction stream data. The IB accepts requests for bytes from the decoder and notifies it whether or not the bytes are available. The IB model does not actually store any stream data; however, it does manage the count of valid bytes within IB longwords as that data is shifted in and out. The cache-control component prefetches data for the IB and also notifies the IB of prefetched data whenever no other activity is scheduled for the cache during a cycle. When full, the IB notifies the cache control of that condition. In turn, the IB is notified by the pipeline model when it needs to be flushed due to a change in the instruction stream sequence.

The configuration of the IB is controlled by two parameters: the number of blocks, and the number of bytes per block. For the VAX 8800 family processor, the IB has four blocks, each four bytes long.

Cache Arbiter, Control, and Queues

From the viewpoint of performance, the cache subsystem in the VAX 8800 family processor contains an important set of resources. This cache design was modeled in the simulator by three state-machine components: the cache arbiter, the cache control, and the cache memory-request queues. From the viewpoint of performance simulation, these functions were the most independent ones that could be segregated.

The cache arbiter state-machine collects requests from the three components that require cache service. The first, the pipeline model, sends read/write arbitration signals for the processor. The second, the cache-control model, sends read arbitration signals for a stalled-processor condition. The third, the memory interconnect model, sends memory arbitration signals. During every cycle, the arbiter sends to the cache control the arbitration winner that will have the cache during the next cycle. There is a fixed priority for choosing an arbitration winner. Memory has the highest priority, followed by processor reads and writes of various types; cache IB prefetching (the default) has the lowest priority. The cache-control and memory-request queues models also provide status information used in deciding an arbitration winner. Certain types of stalls result in no winner. The arbiter model requires no parameters to be specified by a user at runtime.

The cache-control state-machine is the center of the performance simulation model in the sense that it communicates with all but one of the other state-machine components. The hardware resources managed include the combined instruction-stream-and-data cache, and a longword delayed-write buffer used to hold write-hit data until it can be written into the cache. Like the IB, the cache control model keeps control and status information only for the cache and the write buffer. During every cycle, the cache control acts on the request chosen during the last cycle by the arbiter. That request can be a refill from memory, a read lookup and the appropriate cache hit or miss activity, or a write to the delayed-write buffer and memory. For a cache-write request, the data in the delayed-write buffer is written to the cache when the next write request is processed, and then only if the address of the buffered write actually hit in the cache. If there are no memory or processor requests, data is prefetched for the IB automatically, by default.

A number of parameters can be specified at runtime within the cache control, most of them specifying the configuration of the cache. Such configuration parameters include

- Switching the cache on or off
- The cache size in bytes
- The set size
- The block size in bytes
- The block fill size in bytes
- The block replacement algorithm (random, least recently used, or FIFO)
- The memory updating algorithm (write back or write through)
- Allocation for write misses

Control does not exist for all possible cache options in the processor model for the VAX 8800 family, but the cache routines do support them. The implemented cache configuration is 64KB, direct mapped with 64-byte blocks and a 32-byte block fill (done as two separate 16-byte refill sequences). It features write-through memory updating and no allocation for write misses. For study purposes, another parameter was included that allows either one- or two-cycle read hits to the cache. The VAX 8800 family processor design implements one-cycle cache read hits.

The cache memory-request queues state-machine manages the IB read-miss queue, the processor read-miss queue, and the write-buffer queue. The IB read-miss queue has two elements, thus allowing two outstanding misses for IB data. A third outstanding miss will replace the second one, thus avoiding a pipeline stall. The processor read-miss queue has one element; therefore, two outstanding read misses will stall the pipeline. However, processor read hits are allowed to continue with one outstanding read miss. The write-buffer queue consists of two octaword (16-byte) elements. Consecutive writes within the same octaword are buffered until an event forces data in the write buffer to be sent to memory. That event can be encountering either a write that is not in the same octaword or a microcode control command. The cache control sends read-miss and write requests to the appropriate queue. If a queue is full, a signal tells the cache control that no more requests can be accepted.

From the cache queues, requests to memory are generated and sent to the memory interconnect after the arbitration for that interconnect has been won. These requests are prioritized to facilitate choosing which of three possible requests will be sent to the memory interconnect at any point in time. To maintain the ranking, a two-bit counter will increment only on the appearance of a write following a read. The request chosen is the one with the lowest rank count. If two requests have the same ranking, priority will be given first to the write, then to the processor read, and finally to the IB read. The cache queues component has one parameter that can be specified at runtime: the number of cycles that a request ready to be sent to the memory interconnect must remain queued. The final processor implementation required only one cycle, although this timing was not known when the model was built.

Memory Interconnect

The memory interconnect state-machine handles requests between the cache queues and memory. Transactions requiring one or more cycles on the bus include cache-refill data, in octaword packets, from memory; processor-write requests of up to an octaword in size; and processor data- or instruction-read requests for 32 bytes (returned from memory as two octaword packets). Until transmitted, each transaction "owns" the bus. A one-cycle settle time is required between transactions as well. Arbitration for the bus occurs during every cycle to choose a winner for the next cycle. Priority is given first to the current transaction holding the bus, then to the one-cycle settle time, then to memory, and finally to any pending write or read from the cache. A cache request to memory is queued during the cycle after the request was transmitted on the bus. The timing of subsequent cache requests for memory is controlled by the sum of two parameters specified at runtime. These parameters are

- The number of cycles between the time a cache request transmits on the interconnect and the time the cache receives an acknowledgment from the bus
- The number of cycles between the time the cache receives the bus acknowledgment and the time the next cache request can transmit on the bus

The VAX 8800 family processor implementation has a value of two for each parameter, although this timing had not been determined when the model was created. Several other parameters were included in the memory interconnect state-machine for study purposes. The one-cycle settle time can be enabled or disabled, and the interconnect can acknowledge configurations with either one or two processors. We also included the capability to slow the memory subsystem, relative to the processor/cache request timing, by either two or three times.

Memory

We had considered modeling in detail the designs for both the memory controller and the array module. The effort required was so substantial, however, that we first modeled only the best- and worst-case scenarios. The ensuing results indicated that extra detail in the model would not yield correspondingly enlightening information; therefore, the memory state-machine models only best- and worst-case memory performance. The choice of best- or worst-case is a parameter specified by the user at runtime.

The best-case memory model assumes memory is never busy and can take requests from the memory interconnect whenever they are generated. Thus instead of the eight memory-array modules the processor is limited to, this model effectively simulates an infinite number of modules with no contention for specific ones. The only parameter the user must specify is the number of cycles between the time the read request reaches memory and the time memory arbitrates for the memory interconnect to return requested read data to the cache. The implementation has a value of approximately 14 cycles, which reflects the memory read latency. Write requests for memory are simply delivered; no further action has to be taken.

The worst-case memory model assumes only one array module is available to handle read and write requests. Requests for memory are queued in a buffer for processing by the array module. When all queue elements have requests, a memory-busy signal will inhibit the memory interconnect from sending additional requests until a queue element is available. A number of parameters can be specified by the user at runtime to control the timing of requests within the memory controller and the array module. One parameter is the length of the memory-request queue, a value from one to eight. The processor design used a value of three for this queue length. The other parameters are the numbers of cycles required for various operations, as described below. The actual value specified for the processor design is contained between the parentheses following each parameter's description. These parameters are

- The time a request must be queued before processing in the array module (2 cycles)
- The time required by the array module to process a read (12 cycles)
- The time required by the array module to process a write (9 cycles)
- The time required by the array module to process read data for a masked write (2 cycles)
- The time required for a refresh of the array module (12 cycles)
- The time between array refresh signals (300 cycles)

Processor Resources Not Modeled

In addition to some of the microcode and parts of the memory subsystem, several other parts of the design are not simulated. The translation buffer that contains virtual-to-physical address mappings is not modeled. (The design has a 1024-entry, direct-mapped translation buffer, half of it for system-space addresses, the other half for process-space addresses.)[3] The logic and microcode that handle alignment traps are not modeled. Any unaligned addresses associated with processor read and write requests for the cache are automatically aligned by the simulator. Finally, no I/O traffic is generated on the memory interconnect to compete with processor and memory traffic. These omissions could impact the simulated performance of some processor designs for some workloads. However, their exclusion from this model did not impact the performance estimates generated for the processor with the set of workload programs used.

Evolution of the Model

Before presenting studies done with the processor performance simulator, we should examine how the model evolved. Our most significant achievement was to continue developing the model even as project goals changed and as the design materialized over time. This continual adjustment resulted in a model that reflected the latest design and could be used in new design studies.

The first version of the simulator was not very detailed. It included the pipeline, the instruction buffer, the cache arbiter, a cache shell, and some hand-coded microcode for evaluating operand specifiers and for a limited number of VAX instructions. No lookup was done in the cache shell. A parameter specified the hit and miss percentages desired, and random number generation was used to decide the lookup results. Runs were made with both two and four IB longwords, and 90 and 100 percent hit rates in the cache; the workload was the Towers of Hanoi benchmark. Two important results were indicated: first, the performance was in line with the stated goals; second, it was desirable to have more than two IB longwords.

At that point, a more aggressive set of design goals was set by engineering management. Therefore, the next version of the simulator modeled more of the detailed implementation that was evolving. This detail included the decoder, the cache-control and memory-request queues, and the memory interconnect. We developed microcode translation software and used the first base-level microcode released to control the model. Some custom coding was done to accommodate single-precision floating point instructions that were needed. Both hardware and microcode bugs were uncovered during the design and verification of this simulator version, thus increasing its value to the designers.

Performance Simulation Results and Studies

Using the simulator just described, we carried out a number of studies to verify the processor's performance and to examine design alternatives. Since the detailed results are very extensive, this concluding section outlines the kinds of performance information gathered and highlights a number of studies that were done.

Performance Information Gathered

Information provided by a performance simulator falls into four areas:

1. Measuring the performance of a program on an existing processor and then tracing that same program to drive a processor simulator are used to produce a relative performance estimate for the proposed processor. (Of course, this comparison is reasonable only if both processors are implementations of the same architecture.) The information needed to make the comparison includes the following: the total number of instructions executed, the execution time required, and the cycle time on the measured system, as well as the total number of instructions simulated, the total cycles required, and the proposed cycle time on the simulated system. The VAX-11/780 processor was used as the comparison machine for generating performance estimates relative to the VAX 8800 family processor design.
2. Simulating the use of resources within processor system components produces information about how efficient each component is in processing requests and how well the components interact. Knowing what requests are received and what percent of the time component resources are stalled or busy (and why) provides insight into the overall system performance. We found that presenting this detailed information in terms of averages-per-instruction was an effective way of summarizing the activities. This information helped the designers in making hardware design decisions at a low level.
3. Varying the parameter values in a simulator and comparing the results produces useful information to evaluate high-level design and configuration decisions. Since the VAX 8800 family processor design was modeled, a number of studies have been done to evaluate schemes that could be used in new processor designs.
4. Analyzing the instruction stream data from the trace that drives the simulator produces information about how the architecture's instruction set is used. This type of information helps designers decide which optimizations are most beneficial, especially in the microcode flows. Gathering this information generally does not require processor-specific functions in the simulator. Therefore, the simulator does not produce that information. For our purpose, the information was gathered from another package of analysis software.[4] Only individual VAX instruction times that were specific to the VAX 8800 family processor came from the simulator.

Highlights from Simulation Studies

Initially we used the Towers of Hanoi, the prime-number generator, and the single-precision Whetstone benchmark to drive the model. From it we derived results indicating that the performance of the VAX 8800 family processor was between 4.5 and 5.6 times that of a VAX-11/780 processor. The designers made one change based on the resource utilization statistics the simulator generated. Cache read hits had required two cycles, rather than the usual one cycle, when the read address also matched a valid delayed-write buffer address. This number was changed to one cycle when the simulator showed the frequency of this event was higher than anticipated.

Once the basic processor design had been successfully modeled, work focused on broadening the microcode coverage and simulating various alternatives. Better microcode coverage allowed more programs to be traced and run through the simulator. We wanted to use more diverse programs, like the FORTRAN compile and the VAX/VMS link, to exercise the design using the simulator. Alternatives such as cache flushing to simulate context switching, the worst-case memory model, and the dual-processor version were also added. To study the model's behavior, we ran many simulations, varying the basic processor configuration and comparing results to detect the effects. Even today, this work continues as new design ideas surface.

The following list shows the VAX 8800 family processor simulation parameters and configurations that were most sensitive from a performance point of view:

- Context switching, simulated by invalidating all cache entries every *n* VAX instructions, showed a performance degradation from 8 percent when done every 10,000 instructions, to 23 percent when done every 2,000 instructions. We chose an interval of 5,000 instructions for the simulator, which is a conservative estimate. (The degradation was 13 percent for 5,000 instructions.)
- A timing requirement of two cycles for read hits in the cache, rather than one cycle as implemented in the VAX 8800 family processor design, degraded the simulated performance by 9 percent.
- The latency time for memory reads decreased performance by about 0.75 percent for each additional cycle of latency.
- The worst-case model for memory, using only one array module, required 14 percent more cycles than the best-case model. (This result contributed to our decision to use only the best and worst cases.)
- A slow memory interconnect and controller relative to the processor degrades the performance gains when a faster processor is used. Doubling the processor speed by cutting the cycle time in half increased performance by only 1.5 times over that of the slower processor with the same memory. Tripling the speed increased performance by only 1.7 times.
- Enhancements made in the FORTRAN compiler for generating code had a great impact on the instruction stream traced, as well as on the performance estimates derived using the FORTRAN benchmarks. This improvement was particularly noticeable for the FORTRAN compiler released with VMS Version 4.

Summary

The development of the VAX 8800 processor performance simulator continued throughout the entire project. The simulator helped to verify the attainment of performance goals and provided performance trade-off information to the designers. The model's results fostered discussions about interfaces, helped the designers to find problems, and uncovered unanticipated interactions. The simulator continues to contribute to current processor design efforts through its use in studying the performance impact of alternatives.

In addition, we learned a number of important lessons that will be useful in designing future simulators. First, it is important to develop the basic processor simulation functions as early as possible in a design project. Having a general-purpose cache model that can be called and controlled from different processor implementation models is one of the most important functions.

Second, defining and developing a monitor to control the various parts of a simulator, apart from implementing the particular design, has significant implications for designers of performance simulators. Having separate control functions allows the implementor to concentrate on understanding the design to be modeled, as well as to take advantage of features provided by the control monitor to debug the model. Separating control from the simulated design, however, does not result in a simulator with the most optimized runtime performance.

Acknowledgments

I had the support of many people in developing the performance simulator for the VAX 8800 family processor. The processor hardware and firmware teams explained the design, reviewed the results, and encouraged the effort. Simon Steely and Mark Firstenberg helped to design and implement the original simulation tools. Peter Craig developed the microcode characterization process and software. Eric Rasmussen created the dual-processor version of the simulator and the ADA performance simulation model.

References

1. C. Wiecek and S. Steely, "Performance Simulation as a Tool in Central Processing Unit Design," *Performance Evaluation Review,* vol. 11, no. 1 (August 1979): 41–47.

2. T. Leonard, ed. *VAX Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1986).

3. S. Mishra, "The VAX 8800 Microarchitecture," *Digital Technical Journal* (February 1987, this issue): 20–33.

4. C. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *ACM Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems* (March 1982): 177–184.

Stuart J. Farnham
Michael S. Harvey
Kathleen D. Morse

VMS Multiprocessing on the VAX 8800 System

Some features of the VAX 8800 architecture are particularly relevant to multiprocessor operation. Special hardware, not included in the VAX architecture, allows the VMS operating system to use both CPUs in an asymmetric, tightly controlled fashion. The processors operate in a master-slave relationship with one CPU handling all I/O. The hardware handles interprocessor interrupts, cache coherency, and shared memory. VMS uses the interprocessor interrupt in managing operations between the master and slave CPUs. The VMS system also uses interlocked instructions, exception handlers, and traps to handle multiprocessing. These instructions allow events to be scheduled and executed efficiently on both processors.

Every computer system is a combination of hardware and software architectures, the operating system being a direct result of their merger. The same operating system can be implemented on different hardware systems with the same architecture, but a user can access only those features that each set of hardware can support. The most effective merger is the one allowing users of the resulting operating system to make maximum use of all the features designed into both the hardware and software architectures.[1] The VAX 8800 multiprocessor is an example of the result of such an effective merger.

The VAX Architecture and Multiprocessing

Many of the VAX 8800 hardware features important to VMS multiprocessing are defined by the VAX architecture for single-processor and multiprocessor systems alike.[2] These features include the processor modes, I/O and software interrupts, exception handling, asynchronous system traps (ASTs), and interlocked instructions. This section briefly describes these features, which are discussed in more detail later.

Processor Modes

The VAX architecture defines four modes in which a processor may execute. In order of decreasing levels of privilege, these modes are kernel, executive, supervisor, and user. Most of the critical resource management code in the VMS system is executed in kernel mode; in fact, some instructions can be executed only while in that mode. Two examples of such instructions are LDPCTX and MTPR (move to processor register). LDPCTX loads the context (stacks, page tables, and so on) of a process into a CPU so that the process can execute. MTPR is used, among other things, to enable, disable, or trigger certain interrupts during resource management.

Interrupt and Exception Handling

The VAX architecture supports the immediate servicing of important events by means of a mechanism that can transfer control away from the currently executing process. Events that are primarily relevant to and normally invoke software in the context of the currently executing process are called exceptions. Events that are relevant to other processes, or to the system as a whole, are called interrupts, which are serviced in a system-wide context.[2] The VMS operating system provides a handler routine for each exception and interrupt defined by the VAX architecture.

Upon system startup, the VMS operating system initializes a system control block (SCB), which defines the locations of the various event handlers, as shown in Figure 1. The SCB contains

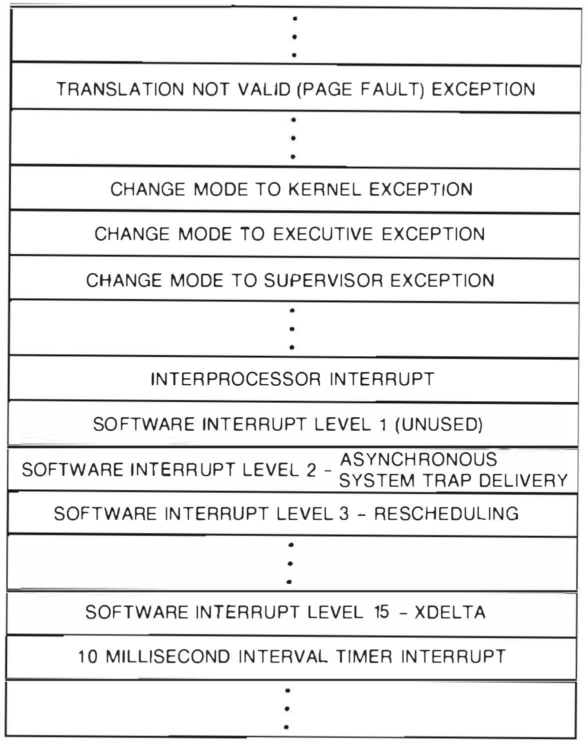

TRANSLATION NOT VALID (PAGE FAULT) EXCEPTION
CHANGE MODE TO KERNEL EXCEPTION
CHANGE MODE TO EXECUTIVE EXCEPTION
CHANGE MODE TO SUPERVISOR EXCEPTION
INTERPROCESSOR INTERRUPT
SOFTWARE INTERRUPT LEVEL 1 (UNUSED)
SOFTWARE INTERRUPT LEVEL 2 - ASYNCHRONOUS SYSTEM TRAP DELIVERY
SOFTWARE INTERRUPT LEVEL 3 - RESCHEDULING
SOFTWARE INTERRUPT LEVEL 15 - XDELTA
10 MILLISECOND INTERVAL TIMER INTERRUPT

Figure 1 System Control Block

an assigned longword that holds the address of the handler for each interrupt and exception serviced by the operating system.

Interrupts and exceptions have varying degrees of urgency. Each event has a specific interrupt priority level (IPL) that designates the relative priority of that event. The VAX architecture includes 31 IPLs, divided into 15 software levels (numbered, in hexadecimal, 01 to 0F), and 16 hardware levels (10 to 1F). User applications and system services run at the process level, which may be thought of as IPL 0. Interrupt levels with higher numbers have higher priorities. That is to say, a request at an IPL higher than the processor's current IPL will interrupt immediately; requests at the same or lower levels will be deferred.[2] The interprocessor interrupt and the 10-millisecond (ms) interval-timer interrupt are examples of hardware interrupts. The rescheduling interrupt and the AST-delivery interrupt are examples of software interrupts.

Software executing in kernel mode posts a software interrupt by setting the appropriate bit in the software interrupt request register (SIRR). A bit exists in the SIRR for each software interrupt level. An interrupt can take place only when the IPL level of the CPU has been lowered below that of the pending interrupt. For example, the handler for the interprocessor interrupt (executing at IPL 20) can post a reschedule event (a software interrupt at IPL 3) by setting the appropriate bit in the SIRR. When the CPU's IPL drops below IPL 3, the IPL 3 interrupt handler is invoked, which is the VMS code that initiates process rescheduling.

This technique allows high IPL code threads to schedule lower IPL functions in a way that allows all potentially interrupted code threads at intermediate IPLs to complete first. Should a higher IPL code thread merely lower the IPL by force to execute the lower IPL function, any intermediate IPL code threads that had been interrupted would complete out of order, thus breaking the software synchronization.

AST Delivery Mechanism

In any mode, the VAX/VMS system can interrupt a code thread executing at IPL 0, begin a new code thread (also at IPL 0), and then continue the previously interrupted code thread. This mechanism is called "delivering" an AST. The hardware notifies the operating system that an AST is deliverable to the currently executing process by means of an interrupt at IPL 2. (Note that this is the only instance of the VAX hardware posting a software interrupt). Any process-context code thread that must execute without interruption by an AST has to be executed at IPL 2 or higher. If a deliverable AST is queued to the current process and the IPL of the CPU drops below 2, then an IPL 2 interrupt will be generated. To execute that interrupt, the IPL 2 interrupt handler first verifies that the AST can be delivered and then delivers it to the process, after which the new code thread associated with the particular AST is executed.

An AST code thread is associated by a process with events that are expected to complete asynchronously to the main thread of the process. An example of such an event is an I/O request that, once issued, is handled by the system in parallel with the main thread of the process. Upon I/O completion, the associated AST is delivered, which causes the main thread of the process to be interrupted in favor of the AST's code thread.

When an AST is specified for an asynchronous event, it is assigned a particular processor mode.

When the AST is queued to a process, its delivery is deferred while that process is executing in a more privileged mode than that of the queued AST. For example, when an AST in supervisor mode is queued to a process executing in kernel mode, the AST will not be delivered until the context changes from kernel mode to at least supervisor mode.

Interlocked Instructions

The VAX architecture includes a few instructions that allow synchronous access to locations in memory. Only those instructions will guarantee consistent results if multiple processors want simultaneous access to the same memory location.

For bit manipulations, these interlocked instructions are

- BBCCI — Branch on bit clear and clear interlocked
- BBSSI — Branch on bit set and set interlocked

For arithmetic manipulations, there is ADAWI — Add aligned word interlocked.

For queue manipulation, the instructions are

- INSQHI — Insert at head of queue interlocked
- INSQTI — Insert at tail of queue interlocked
- REMQHI — Remove from head of queue interlocked
- REMQTI — Remove from tail of queue interlocked

These instructions are used extensively in the operating system to provide multiprocessor synchronization. They are also available to user processes to synchronize access to shared application data.

The VAX 8800 System

The specific implementation features of the VAX 8800 multiprocessing system are described in this section. Remember that the 8800 is only one of many implementations of the VAX architecture. Several important hardware features provided by the 8800 are not specified in the VAX architecture but are required for VMS multiprocessing. These hardware features are

- Primary processor access to all peripherals
- Interprocessor interrupts
- Shared main memory
- Cache coherency

VAX 8800 Implementation

The VAX 8800 system consists of two VAX 8800 processors that share main memory by means of a fast memory-system interconnect called the NMI bus.[3] The processor hardware is completely symmetric; that is, either processor can fulfill the role of primary processor for any booted instance of the operating system. Figure 2 is a block diagram of the VAX 8800 system.

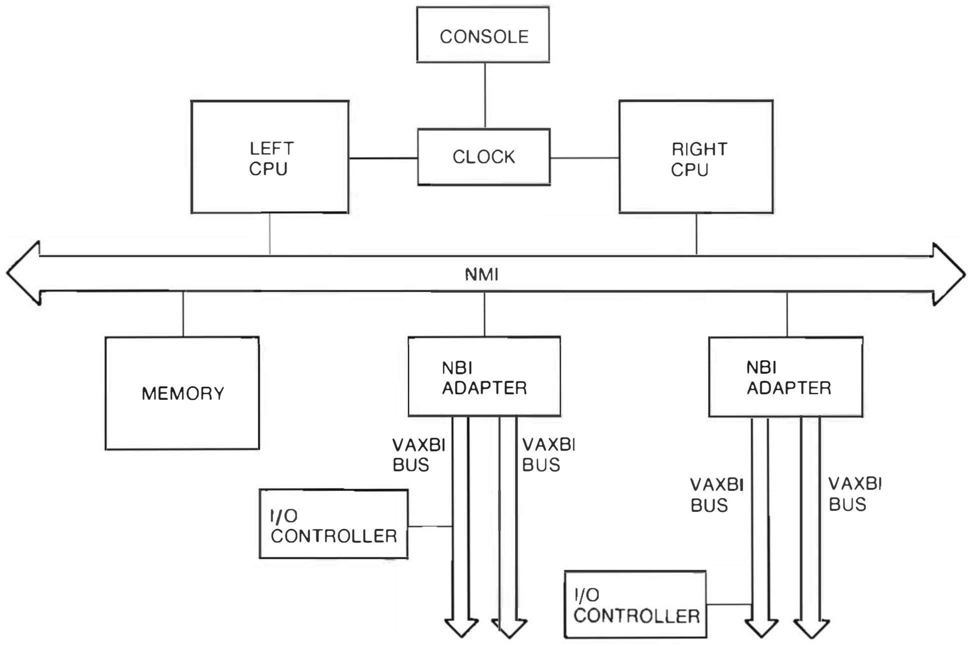


Figure 2 Block Diagram of VAX 8800 System

There is one console subsystem in the 8800, which is shared by the two CPUs. The console command language, implemented in software in the console subsystem, is a superset of the console functionality specified by the VAX architecture.[2] Both CPUs can be controlled from the single console terminal. After the system is booted, the console terminal can be used like any other terminal connected to the system.

All I/O devices are connected to the system through VAXBI buses. The 8800 can accommodate up to four VAXBI buses, each of which can accommodate up to 16 nodes, generally I/O controllers. The buses are connected to the NMI by means of the NMI-to-VAXBI adapters, called the NBIs. Each NBI consists of either two or three parts: an NBIA, which is the interface to the NMI; and one or two NBIBs, which are interfaces to the VAXBI buses. An NBIB is one of the 16 nodes on its respective VAXBI bus.

Under VMS multiprocessing, all peripherals are controlled by the first processor to be booted, designated the primary processor. The other processor, the secondary, is prevented from accessing any peripheral devices (disks, terminals, and so on) because the code communicating with those devices runs in kernel mode, an access mode that VMS utilizes only on the primary. Thus, all I/O peripherals will be accessed only by the primary processor. Typically, the left CPU in the VAX 8800 system is chosen as the primary processor. However, console commands are available to designate either CPU as the primary one. A change in that designation takes effect after the next INIT command is received by the console.

The VAX 8800 hardware provides the capability for one processor to interrupt the other. This interruption is accomplished by writing a value of 1 to an internal processor register on the interrupting CPU by means of the privileged MTPR instruction (from kernel mode only). The VMS system uses this mechanism to synchronize the CPUs as different system events occur.

The main memory contains one copy of the VMS software, which depends upon the memory subsystem and interlocked instructions for cache coherency and the consistency of memory contents. The VAX 8800 memory subsystem automatically handles all cache updates; no software logic is needed to maintain consistency between the cache contents in each processor. The 8800 does implement a write buffer to optimize transfers across the NMI to the memory subsystem. Therefore, the interlocked instructions must be issued to flush the necessary write data all the way out to memory. If one processor modifies shared data, the other needs to see the change in a synchronized and timely fashion.

Multiprocessor Implementation Improvements

The VAX 8800 system includes features that are improvements over previous multiprocessing VAX hardware implementations, such as the VAX-11/782 system. Larger amounts of physical memory can be used, all of which is available to the VMS system or the system diagnostics. Moreover, the 8800 cache provides better performance, and the system has a smaller footprint and a better price/performance ratio. Perhaps the most significant fact from a system-management viewpoint is that only one console subsystem with one terminal is needed to control the entire multiprocessor. This single control point has ramifications for setting up the system and running it as a multiprocessor.

The console subsystem has access to the memory configuration of the 8800. With previous multiprocessors, the system manager had to configure memory by manually determining the appropriate data, then entering it into customized command procedures on specially built floppy disks in the console.[4]

The console subsystem of the 8800 also eliminates the need for operator intervention to boot or restart the secondary processor. The VMS system is initially booted on the primary processor and subsequently directs the console subsystem to boot the secondary. Similarly, the console subsystem restarts the VMS system on the primary processor after a power failure. The operating system then directs the console to restart the secondary at the appropriate point in the power-recovery sequence. At no time must the operator be involved in bringing the secondary on line.[4]

The VMS Operating System

The multiprocessing aspects of the VAX architecture and the VAX 8800 implementation provide the underlying hardware support for a totally integrated multiprocessing computer system. This section discusses aspects of the VMS software that are specifically related to multiprocessing as implemented for the 8800. (See reference 5 for additional multiprocessing information and recommended programming techniques.)

Classification

In multiprocessing terminology, VMS multiprocessing is classified as "asymmetric" and "tightly coupled." An asymmetric system is one in which one CPU, called the primary, has critical system-wide responsibilities, including the management of all the CPU resources. The other CPU, called the secondary, has more restricted responsibilities that exclude the management of critical system resources (including itself). This type of multiprocessing system is also referred to as a "master-slave" arrangement. The other classification, tightly coupled, means that both processors operate in a closely synchronized fashion; if they fail, they fail together.

On a VMS multiprocessing system, both processors share the same copy of the operating system, although some code is executed only by one or the other CPU. Most of the kernel logic in the VMS operating system is executed only by the primary processor. That eliminates the need for the complex synchronization and locking mechanisms that would otherwise be required to protect the system's data structures from access by multiple CPUs.

History of VMS Multiprocessing

VMS multiprocessing was introduced during the development of VMS Version 3.0. At that time, the power of a single VAX-11/780 processor was insufficient to build the VMS executive in a reasonable amount of time. Several constraints were placed on the multiprocessing development effort. It had to involve minimal changes to VMS kernel mode routines, use existing hardware, and have minimal performance impact on single-processor VMS systems.[6]

The first constraint above had the greatest impact on the chosen design of VMS Version 3.0. To achieve fully symmetric multiprocessing, changes would be required throughout the whole operating system to extend IPL synchronization as already implemented by VMS for single-processor operation. Since those changes were too extensive to make, we chose an asymmetric design in which the synchronization of critical code was achieved by limiting that activity to the primary CPU. In this context, existing IPL-based techniques were sufficient to synchronize the code threads in kernel mode.

The second constraint led us to configure a system with two VAX-11/780 CPUs coupled by an MA780 shared memory. In this configuration, each CPU has a separate, independent console subsystem; neither has access to the other's console. Booting this multiprocessor requires special console command files and operator intervention for both CPUs. Similarly, the I/O devices configured on one CPU are inaccessible on the other. Since most of the I/O subsystem code executes in kernel mode, this constraint has the effect of limiting the I/O devices usable by the multiprocessor to those connected to the primary CPU.

The final constraint led to a design that allows multiprocessing code to be inserted dynamically into the running executive. No multiprocessing code is present in a single-processor configuration of VAX/VMS.

The multiprocessing capabilities in VMS Version 3.0 were extended to support the new VAX 8800 system. These extensions take advantage of new functions allowed by the new VAX design. For example, as mentioned earlier, the shared console subsystem allows the secondary processor to be booted from the primary under program control; no operator intervention is required.

Division of Work between Processors

As mentioned earlier, the VMS multiprocessing code is a master-slave implementation. The secondary CPU is required to do whatever work is assigned to it by the primary. The secondary CPU can execute application code only, while the primary CPU handles the I/O, paging, and all resource management, as well as the execution of application code. Since all system services that manage system resources are executed in kernel mode, only the primary CPU is allowed to execute those services. The secondary CPU can execute code that is in any other mode: user, supervisor, or executive. Thus, to be technically accurate in multiprocessing terminology, the VMS multiprocessing system is symmetric for code in the user, supervisor, and executive modes, but asymmetric for code in kernel mode.

The VMS boot code creates a SCB for each processor. As described earlier, the SCB contains vectors to routines that handle various interrupt and exception events. Many VMS interrupt and exception handlers are identical for both the primary and secondary processors. However, there are cases in which exceptions or interrupts must be handled differently, depending upon which processor receives the event. The interprocessor interrupt and the software interrupt used for rescheduling are both examples of system-wide events. Both are vectored through the SCB but require different handlers for each processor. (Figure 1 shows the various interrupt levels in the SCB.) The AST-delivery software interrupt and the quantum end, a scheduling event (described later), are examples of process-related events that also require different exception handlers in the SCB of each CPU. By separating the handlers into processor-specific SCBs, the more costly and difficult task of run-time separation within an otherwise commonly executed handler is avoided.

Typically, when an exception occurs on the secondary, that CPU's exception handler "reflects" that exception back to the primary. To do that, the exception handler stores both the address of the primary's exception handler and an appropriate processor status longword (PSL) on the stack of the current process. The secondary's exception handler then saves the context of the current process and passes the process back to the primary by requesting a rescheduling event. The process eventually executes on the primary, whose exception handler will immediately get control as if the exception had occurred there originally. Exception processing is therefore synchronized on a system-wide basis by virtue of running on the primary processor only.

The SCB for the primary CPU consists of multiple pages of interrupt and exception vectors. The format of the first page is defined by the VAX architecture. This page contains vectors for all implementation-independent exceptions and interrupts, and for a few implementation-dependent ones. Additional pages of vectors are provided for I/O interrupt handlers. Under VMS multiprocessing, the length of the SCB for the secondary CPU is one page. The pages that make up the I/O subsystem portion of the SCB are not needed on the secondary, which will not initiate I/O requests nor receive I/O interrupts.

Interprocessor Interrupts

The VAX 8800 hardware provides a key feature for optimizing the VMS multiprocessing software: the ability of one processor to interrupt the other. This interprocessor interrupt mechanism is used extensively on each CPU by the VMS operating system.

The primary processor interrupts the secondary for several reasons. First, the primary can request an invalidation of a translation buffer entry corresponding to a system-space address that is about to be invalidated on the primary. This event forces coherency between the translation buffers of both processors with respect to mapping changes in the shared system virtual address space. Second, the primary can interrupt because it has queued an AST, typically for I/O completion, for the process currently executing on the secondary. This event ultimately results in the process being rescheduled onto the primary, where the actual delivery of the AST to the process can be accomplished. Finally, the primary can initiate and synchronize a system-wide shutdown or a crash.

The secondary processor will interrupt if it wants the primary to take back the current process and find another process for the secondary to execute. The secondary will also interrupt if it detects a hardware error or if it wants to initiate a system-wide crash.

Secondary State Transitions

A state variable is maintained to record the current state of the secondary processor. The primary processor uses this state to determine whether or not to schedule work for the secondary. When the secondary is booted, the state variable is already set to INIT. After booting, the secondary changes the state variable to IDLE. During its next reschedule operation, the primary will notice the IDLE state and attempt to schedule a process for the secondary to execute. After finding a process for the secondary, the primary sets the state variable to BUSY. The secondary, which has been continually checking the state variable for this transition, then loads the process's context from memory and sets the state to EXECUTE.

The secondary will execute its current process until the process either receives its quantum of CPU time or is blocked by some request that must be synchronized in a system-wide context. (That request must be executed in kernel mode on the primary.) At this point, the secondary saves the process's context in memory and sets the state to DROP. Using the VAX 8800 interprocessor interrupt mechanism, the secondary then interrupts the primary and requests another process to execute. The primary takes the saved process back from the secondary, setting that CPU's state to IDLE. Thus, the state transition has made an entire circuit.

Figure 3 shows the state transition diagram for the secondary CPU. The primary's paths are marked P and the secondary's paths are marked S to indicate which processor controls each transition from one state to another. The only state not explained above is the STOP state, used only when the secondary is shut down.

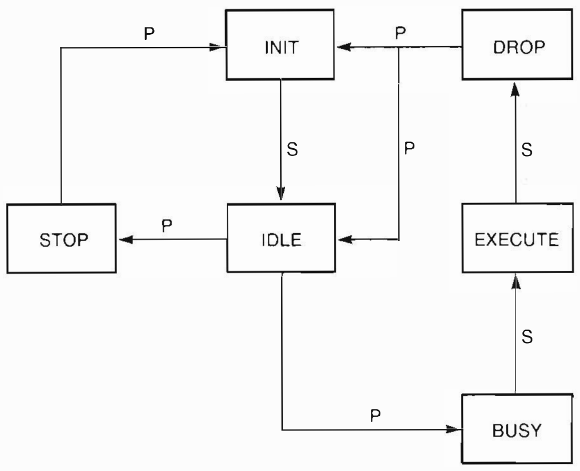


Figure 3 Secondary CPU State Transitions

Process Scheduling under the VMS Operating System

Some aspects of process scheduling were discussed in the previous section. This section describes in greater detail how process scheduling is implemented in the VMS system and which of its aspects are different in a multiprocessing environment.[6]

Single-Processor Scheduling

The VMS scheduling algorithm implemented on a single processor is round-robin and preemptive, with the highest priority process being executed first. There are 31 levels of process priority (which are not the same as interrupt priority levels). Thirty-one is the highest priority, one the lowest; process priorities are subdivided into real-time (priorities 16 to 31) and "normal" (priorities 0 to 15) ranges. The real-time priorities are used by time-critical applications, such as high-speed data acquisition. When a process is created, it is assigned a base priority. Its priority during execution is guaranteed never to drop below that base priority unless either that process or another, privileged process requests it to.

Each process is allowed a quantum of CPU time (usually 200 ms, equivalent to 20 interrupts of the 10-ms interval timer; however, a system manager can change the default). Each time the interval timer interrupts, the interrupt handler checks to see if the current process has used up its quantum. If so, quantum-end processing is initiated.

For a process with a priority in the real-time range, quantum-end processing consists of awarding a new quantum to the process and allowing it to continue execution. A reschedule event will occur when a normal-priority process has used up its quantum. In the latter case, the current process is placed at the end of the scheduling queue maintained for that process's priority (there is one such queue for each process priority), and the process at the head of the queue is chosen to execute.

The priority of a normal-range process is raised after certain blocking events have cleared. For example, to provide good response time to interactive users, a process's priority will be temporarily boosted after the completion of terminal input. This arrangement results in a tendency for compute-bound processes to remain at their initial priorities (called the base priority). However, I/O-bound and interactive processes, which are blocked more frequently, usually attain priorities somewhat higher than their base ones. A process's priority is lowered one point when the process is scheduled to execute, unless it is already running at its base priority.

Multiprocessor Scheduling

The primary processor schedules all work on the system, for both itself and the secondary processor. The scheduling algorithm used for the primary processor is basically the same one used in a single-processor system (an important goal in this implementation). For the multiprocessor scheduling algorithm, however, certain modifications were made to extend the effectiveness of process scheduling to utilize the additional CPU resources that are available. The execution environment of the secondary processor is more constrained than that of the primary. Most notably, the kernel-mode code is restricted to the primary CPU. The multiprocessor scheduling algorithm attempts to keep that secondary CPU as fully utilized as possible with minimal scheduling overhead in the following ways:

- The primary processor always schedules a process to run on the secondary before scheduling a process for itself to execute.
- The primary processor will schedule a process to run on the secondary only if that process does not require immediate execution in kernel mode and does not have an AST (which requires kernel-mode execution) ready to be delivered. This scheduling helps prevent situations in which a process can flip-flop between processors, sometimes called scheduler thrashing.
- Scheduling is preemptive on the primary processor, but not on the secondary. Thus, if the secondary processor is executing one job when another job with higher priority becomes computable, the primary processor will not interrupt the secondary to give it the higher priority job. Therefore, processes executing on the secondary processor are more likely to run for their entire quantum than are processes executing on the primary.

  This approach guarantees only that the highest priority process will be executing, not the two highest priority processes. To guarantee the latter would require significantly more interprocessor interrupt traffic and is likely to increase thrashing on the entire system, and will especially affect the primary's ability to devote processing time to its own selected process.
- If all computable processes require execution in kernel mode, then the primary processor cannot schedule a process for the secondary and will execute a process itself. Should that happen, an AST-delivery interrupt will be generated automatically after the primary processor stops executing the process in kernel mode. The primary processor handles this interrupt by performing a rescheduling operation. As a result, the primary processor sends the process it was just executing, which is no longer in kernel mode, to the secondary processor in a timely fashion. The primary is then free to execute another process itself.

  When there is only one computable process, one of the CPUs will remain idle. In this case the primary processor executes the process itself even it may be perfectly eligible to execute on the secondary. Thus the overhead processing associated with the post-kernel mode AST and the subsequent rescheduling of the secondary can be avoided. This case also has the effect of preventing future thrashing if the process needs access to kernel-mode resources, at least until enough computable processes become available to keep both processors busy.
- The system services[7] that request event-flag waits ($WAITFR, $WFLAND, and $WFLOR) are among the most commonly executed kernel-mode services.[1] If a process running on the secondary processor requests an event-flag wait, the VMS operating system will attempt to avoid rescheduling the process onto the primary CPU. The system-service dispatcher on the secondary CPU first checks to see if the requested flags are already set. If so, the process is allowed to continue executing on the secondary without rescheduling.

  If the flags are not set, an interprocessor interrupt requesting that the process be placed into an event-flag wait state (either LEF or CEF) will be sent to the primary CPU. When that processor services the interrupt, it again checks to see if the wait request has been satisfied (the flags have been set). If so, the process is allowed to continue executing on the secondary. If the flags are still not set, the process is taken out of execution and placed into the appropriate wait state. The secondary processor then becomes available for scheduling.

Although a process may currently be eligible for scheduling onto the secondary, the VMS operating system cannot predict whether or not that process will require kernel-mode services in the near future. If those services are needed, the process would have to be rescheduled onto the primary. For example, utilities that perform interactive tasks (such as editors or the mail system) require numerous I/O requests. Other types of programs incur many page faults. These processes are therefore poor candidates for execution on the secondary. Sometimes a system manager can predict that certain processes will have those characteristics, and he or she can take preventive measures to avoid processing on the secondary.

The following VMS multiprocessing scheduling features give the system manager manual control over the scheduling of processes onto the secondary CPU:

- A SYSGEN parameter exists to limit the maximum priority of processes allowed to execute on the secondary.[5] Recall that priority boosts are granted to processes after certain events, such as I/O completion. These I/O-intensive processes tend to stay at priorities above those of compute-intensive ones. Therefore, setting the SYSGEN parameter a point or two above the default base-process priority may effectively screen out many "unsuitable" processes from the secondary processor. The system manager can set the SYSGEN parameter to 0 (indicating no priority screening is to occur) or to any value from 1 to 31, which sets the priority limit to the specified value.
- A process can be made ineligible from executing on the secondary processor by means of the SET PROCESS/CPU=NOATTACHED command. This command prevents user processes that execute only interactive or I/O-bound utilities from running on the secondary. This fixed-process attribute remains in force until it has been changed with a SET PROCESS/CPU=ATTACHED command.[5]

Summary

The VAX 8800 system running the asymmetric VMS operating system provides the most computing power currently available in the VAX family to execute compute-intensive applications. The 8800 represents a merger of a new hardware implementation of the VAX architecture with preexisting multiprocessing capabilities in the VMS operating system. This software uses features of the VAX architecture and the hardware for which it was originally intended. With the advent of new multiprocessing hardware, the software design could be modified to take advantage of additional capabilities offered by the advanced hardware design in the VAX 8800 CPU.

Acknowledgments

The authors thank Jill Angel of Digital's Corporate User Publications Group for excellent assistance in organizing this material and for serving as writing coach, and Lawrence Kenah of the VAX/VMS Development Group for technical assistance in planning this paper. Thanks also to all others who reviewed this paper and made technical and editorial suggestions and improvements.

References

1. K. Morse and R. Kinicki, "A Performance Study of Multiprocessor Scheduling Algorithms on a VAX-11/782," *Conference Proceedings of the International Conference on the Management and Performance of Computer Systems* (1985): 280–289.
2. *VAX-11 Architecture Reference Manual* (Bedford: Digital Press, Order No. EY-3459E-DP, 1987).
3. J. Fu, J. Keller, and K. Haduch, "Aspects of the VAX 8800 C Box Design," *Digital Technical Journal* (February 1987, this issue): 41–51.
4. *VAX-11/782 User's Guide* (Maynard: Digital Equipment Corporation, Order No. AA-M543A-TE, 1982).
5. *Guide to Multiprocessing on VAX/VMS* (Maynard: Digital Equipment Corporation, Order No. AA-HP69A-TE, 1986).
6. L. Kenah and S. Bate, *VAX/VMS Internals and Data Structures* (Bedford: Digital Press, 1984).
7. *VAX/VMS System Services Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-Z501B-TE, 1986).

Gabriel P. Bischoff
Steven S. Greenberg

A Parallel Implementation of the Circuit Simulator SPICE on the VAX 8800 System

Multiprocessors are efficient only if the added computing power can be used to solve specific applications. To demonstrate the VAX 8800 multiprocessor's advantages, the authors converted the circuit simulator SPICE into the parallel program CAYENNE. Their methodology involved using VAX instructions and VMS system services to create and control a series of master and slave processes. Other VMS instructions were used to synchronize these processes and to manage the critical sections. Modifications for parallel processsing were made in SPICE's load, LU factorization, and local truncation error phases. The result was that CAYENNE, with two slave processes, ran 1.7 time faster than SPICE.

The realization that two processors might be better than one is not new. Indeed, parallel computing can be traced back to the nineteenth century.[1] The advent of very large scale integration opened a variety of new opportunities in the field of parallel processing for specific applications such as image processing and signal processing. Designing and efficiently using a multiprocessor for general-purpose, high-speed computing, however, is more complex.

The majority of today's application programs are written for single-processor machines. To convert these programs to run on multiprocessor machines and achieve close to the ideal speed up, linear with the number of processors, is not an easy task. Two approaches can be adopted to accomplish this conversion task. The first is to design specific compilers that automatically convert programs written for single processors into programs that run efficiently on multiprocessors. The second is to leave to the application programmer the task of writing code that makes efficient use of the multiple processors.

The first approach is the best from a user's point of view; however, good multiprocessor compilers have yet to be designed. The second approach leaves more flexibility to the programmer, who can modify some of the algorithms in the program to have more concurrency. Indeed, the two approaches should not be mutually exclusive: the compiler can detect parallelism at the instruction level whereas the programmer can define parallelism at the algorithmic level. Parallelism on the VAX 8800 system is achieved through the second approach.

We will describe in this paper the features of the VAX architecture and the VMS operating system that we used to implement our methodology for parallel processing. We will present a set of FORTRAN routines we wrote to relieve the application programmer from having to know the inner workings of the VAX architecture and the VMS operating system. We will then describe the modifications made to the circuit simulator SPICE[2] to develop a parallel processing implementation, called CAYENNE. Finally, we will give comparative timing results on two simulation examples.

VAX/VMS Primitives for Parallel Processing

The VAX 8800 system is a shared-memory multiprocessor; all communications between processors are performed through sections of shared memory rather than through message passing. When writing parallel code on a shared-memory multiprocessor, a programmer must be aware of two concepts: critical section and processor synchronization. A critical section is a section of shared memory that could be accessed by several processors at the same time if no precautions were taken to prevent that. Allowing simultaneous access to shared memory could result in incorrect data. Processor synchronization is the means by which processors proceed in an orderly fashion. It consists of mechanisms allowing processors to broadcast the beginning or the completion of a task or to wait until a signal is received.

Some VAX instructions and some VMS system routines support the management of critical sections and processor synchronization.[3,4] We use three VAX instructions to control access to critical sections:

- BBSSI — Branch on bit set and set interlocked
- BBCCI — Branch on bit clear and clear interlocked
- ADAWI — Add aligned word interlocked

The instructions BBSSI and BBCCI are the VAX implementation of the atomic-test and set instructions that allow the control of access to critical sections to one process at a time. The instruction ADAWI performs an interlocked integer addition and returns a condition status depending on whether the result is zero or nonzero.

We use three system routines of the VMS operating system to support processor synchronization:

- SETEF — Set event flag
- CLREF — Clear event flag
- WAITFR — Wait for event flag

These routines are services provided by the VMS operating system to synchronize processes. Indeed, the significant entity in the VMS multiprocessor environment is not the processor but the process. A processor is a physical processing unit, whereas a process is a software entity created by the VMS operating system. Multiprocessing is achieved by creating several processes that VMS will assign to available processors. Only the operating system, not the user, can assign a given process to a given processor. Event flags are bits maintained by VMS. Several different processes can have access to the same event flag, and signaling between processes can be achieved by setting or clearing an event flag. For example, the system service WAITFR places a process in a wait state pending the setting of an event flag.

Additional VMS system routines allow the creation of processes, the creation and mapping of sections of shared memory, and the initialization of event flags. These system routines are:

- CREPRC — Create process
- CRMPSC — Create and map section of shared memory
- MGLBSC — Map global section of shared memory
- ASCEFC — Associate common event flag cluster

More information on these routines can be found in the *VAX/VMS System Services Manual*.[5] We used the VAX instructions and the VMS system routines listed above to write a set of routines that embeds our methodology for parallel processing.

Parallel Processing Methodology

In the next section we outline the methodology we use to achieve parallelism and in the process define some important terminology. A program we wish to convert for parallel processing is divided into serial phases. Each phase is divided into tasks that are executed either serially or concurrently. A phase whose tasks are executed serially is called a single-stream phase, whereas a phase whose tasks are executed concurrently is called a multiple-stream phase. The single-stream phases are executed by a master process, whereas the multiple-stream phases are executed by slave processes. The slave processes are idle when the master process is active and vice versa. Figure 1 shows this relationship. Master and slave processes run the same executable file, thus leading to easier program maintenance. As mentioned earlier, processes are dynamically assigned to processors by the VMS operating system.

We designed a general set of FORTRAN routines for this environment. This set now has seven entries and implements the critical-section and process-synchronization concepts defined earlier. It also performs the necessary initialization and provides facilities for debugging a multiprocess execution. The remainder of this section describes the functions available in this set.

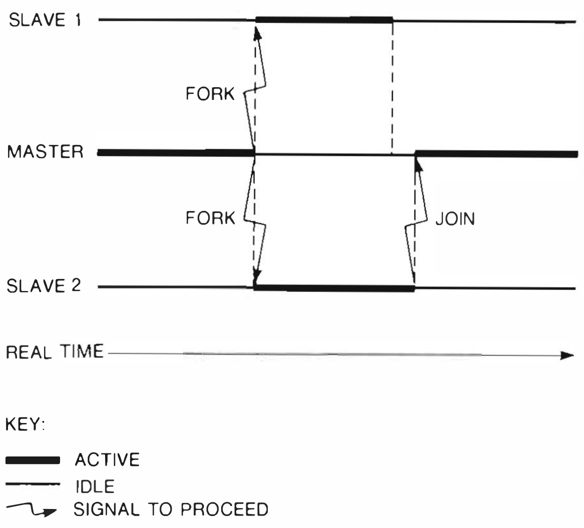


Figure 1 Synchronization of Processes

Initialization

Initialization is performed by a logical function called MASTER_PROCESS, which is set to TRUE if a master process runs the executable file and FALSE if a slave process runs it. The slave processes have special names that differentiate them from the master process. An argument list permits the specification of the number of slave processes to create and the input and output files to use for those slave processes. Through this argument list a unique process number is returned to each calling process.

A user can also specify the number of slave processes to create by using a command-line option when the program is run. For example, the program CAYENNE would be run with *N* slave processes if invoked with the command CAYENNE/SLAVES=N at the $ prompt. If the calling process is a master, MASTER_PROCESS will create the sections of shared memory, initialize the event flags used for synchronization, and create the required number of slave processes. If the calling process is a slave, the function will map the shared virtual-address space to the existing sections of shared memory. The sections of shared memory are FORTRAN common blocks defined as shared when the program is linked with an appropriate linker command. During this initialization phase, CREPRC creates slave processes, CRMPSC and MGLBSC create and map sections of shared memory respectively, and ASCEFC initializes the event flags.

Synchronization

Synchronization is performed by four of our seven subroutines: FORK, JOIN, JOIN_EXIT, and JOIN_FORK. These subroutines use the VMS system routines SETEF, CLREF, and WAITFR to perform the necessary interprocess signaling. Each subroutine accomplishes the following functions:

- FORK — This subroutine is called by the master process to signal the slave processes to proceed. The master process then waits in this subroutine for the slaves to signal back.
- JOIN — This subroutine is called by the slave processes to signal the master process to proceed. The slave processes then wait in this subroutine for the master to signal back.

  Only the last calling slave process signals the master process. The VAX instruction ADAWI is used to identify this last calling slave process.
- JOIN_EXIT — This subroutine is called by the slave processes to signal the master process to proceed. However, the slave processes then exit instead of waiting for a signal. That is the way the slave processes are stopped when they are no longer needed.
- JOIN_FORK — This subroutine is called by the slave processes to synchronize two multiple stream phases with no intervening single-stream phase. The use of this subroutine allows slave processes to be synchronized without having to signal the master process.

These synchronization routines put a process that needs to wait for a signal into a wait state. Processes in a wait state do not use any CPU time. Each call to one of these synchronization routines, however, requires many machine instructions to be executed. If the application programmer anticipates a very short waiting time, an alternative to the previous method of synchronization is synchronization through busy wait. In this scheme a process will loop, executing an instruction of the form DO WHILE (FLAG_IS_NOT_SET) ENDDO. The process will execute the previous instruction until the logical FLAG_IS_NOT_SET is set to FALSE.

The busy-wait form of synchronization needs to be used with care. It can lead to loss of overall system performance. Indeed, the process executing a busy-wait instruction will use CPU time that might be more productively used by another process. In addition, the logical FLAG_IS_NOT_SET, which is constantly checked for, is shared by all processes. Therefore, access to this logical must be carefully controlled. If several processes change this logical at the same time, its final value will be unknown. If no process updates FLAG_IS_NOT_SET, a process may execute the busy-wait instruction forever, thus leading to deadlock. Deadlock occurs when processes are waiting to receive a signal that will never be sent.

Critical Section

Critical sections in a parallel implementation should be minimized. They are the bottlenecks of the multiple-stream phases because they can be accessed by only one process at a time. If a critical section cannot be avoided, the time spent to access this section should be minimized. Exclusive access to critical sections can be achieved by using either the VAX interlocked instructions or the VMS system services.[3] The former method implements a busy-wait form of access synchronization, the latter uses event flags.

The two subroutines LOCK and UNLOCK are assembly language routines implementing a busy-wait form of access synchronization. We chose this method because it is faster in elapsed time, and the time spent by a process waiting is expected to be small when the access to critical sections has been minimized. These subroutines are used in the following manner to access a critical section:

```
CALL LOCK(SECTION_ENTRY)
    CALL ACCESS_CRITICAL_SECTION
CALL UNLOCK(SECTION_ENTRY)
```

SECTION_ENTRY is an integer associated with a given critical section. This integer is set to 1 when a process is using the critical section and to 0 when no process is using the critical section. The two calls LOCK and UNLOCK ensure that only one process at a time executes the code ACCESS_CRITICAL_SECTION. We use these routines only once in CAYENNE for dynamic task allocation.

Parallel Debugging

Debugging parallel code is somewhat more complex than debugging sequential code. We debug our parallel code using the following methodology. The functionality of our parallel code does not depend on the number of slave processes or on which specific process performs a particular task. Therefore, the whole code can be executed by the same process. For example, CAYENNE runs with only one process if the number of slave processes is specified to be zero. This allows most algorithmic modifications made in the code to be debugged with the VMS debugging facilities provided for sequential code.

After the first debugging phase, a code section could still have errors when run with multiple processes. Our routines allow two forms of debugging, requested either through a flag in the argument list of the logical function MASTER_PROCESS or through a command-line option. The first form of debugging permits the assignment of a different terminal to each process and the setting of a debugging session for each process on its assigned terminal. The second form of debugging is intended to be used with a workstation. A different workstation window is assigned to each process, and a debugging session is set up for each process in its assigned window. The number of processes that can be debugged concurrently is limited to either the number of terminals available or the number of workstation windows that can be opened.

Example

The following example, shown in Figure 2, illustrates some of the functionality of our set of routines. We want to compute the sum SUM of all integers from 1 to $N*S$. We assume that a master process with the help of N slave processes does the task. Each slave process is assigned a unique number PROCESS_NUMBER between 1 and N by the logical function MASTER_PROCESS. The section of shared memory consists of an array PARTIAL_SUM of size N. The slave processes work in parallel. Each slave process adds S consecutive integers and stores its result in the shared memory location PARTIAL_SUM(PROCESS_NUMBER).

After the slave processes have completed their task, the master process adds their partial sums, stored in the shared array PARTIAL_SUM, to produce the final result SUM. The code corresponding to this procedure follows. (Remember that master and slave process run the exact same executable file.)

```
PROGRAM parallel

   LOGICAL master_process
   INTEGER process_number
   INTEGER number_of_slaves,default_number_of_slaves
   INTEGER debug_flag
   PARAMETER (default_number_of_slaves=5,debug_flag=0)
   COMMON /shared/ number_of_slaves
   COMMON /local/ process_number

   IF (master_process(process_number,number_of_slaves,
  *     default_number_of_slaves,'input','output',debug_flag)) THEN
      CALL master_code
   ELSE
      CALL slave_code
   ENDIF

END

SUBROUTINE master_code

   INTEGER number_of_slaves,maximum_number_of_slaves,i
   PARAMETER (maximum_number_of_slaves=10)
   INTEGER partial_sum(maximum_number_of_slaves),sum
   COMMON /shared/ number_of_slaves,partial_sum

   CALL fork
   sum = 0
   DO i = 1,number_of_slaves
      sum=sum+partial_sum(i)
   ENDDO

END

SUBROUTINE slave_code

   INTEGER process_number,number_of_slaves,start,s,i
   INTEGER partial_sum(1)
   PARAMETER (s=200)
   COMMON /local/ process_number
   COMMON /shared/ number_of_slaves,partial_sum

   partial_sum(process_number) = 0
   start = (process_number-1) * s
   DO i = start+1,start+s
      partial_sum(process_number) = partial_sum(process_number) + i
   ENDDO
   CALL join_exit

END
```

Figure 2 PROGRAM Parallel

In the next section we describe how we created parallel processing in several phases of the circuit simulator SPICE to produce the program CAYENNE.

Modifications Made in SPICE

Before addressing each parallel phase of CAYENNE, we give a brief overview of the circuit simulator SPICE.

Overview of SPICE

SPICE performs several types of circuit analysis: steady-state analysis, transient analysis, and small-signal analysis. The most commonly used analysis for digital circuits is the transient analysis, which becomes increasingly time consuming as the size of the simulated circuit increases. Figure 3 gives a global description of the algorithms used by SPICE for a transient analysis.

The circuit equations form a system of ordinary differential equations. This system is solved numerically at successive time points t_i, $i = 1$, N. It is reduced at a given time point t_i into a system of nonlinear equations by using a discretization method. A discretization method approximates the time derivative of a variable at a given time point as a function of the value of the variable at that time point and at previous time points. This method introduces a discretization error that must be controlled and maintained below a specified threshold. This error is called the local truncation error. The resulting system of nonlinear equations is reduced to a system of linear equations by performing a first-order Taylor expansion of the nonlinear elements of the circuit. This linearization introduces another error called the linearization error. The resulting system of linear equations is then solved exactly, using an LU factorization of the system matrix.

After the solution of the system has been obtained, the linearization error can be estimated. If this error is too big, a new linearization is performed around the previously computed solution, and the new linear system is solved again. Successive linearizations are performed until convergence is obtained, that is, until the linearization error is below a specified threshold. When convergence is reached the solution of the nonlinear system is obtained, and the local truncation error is then checked. If this error is too big, the solution at time point t_i is rejected and the system of differential equations is solved at a new time point t_j so that $t_i - 1 < t_j < t_i$. If the error is below a specified threshold, the solution is accepted, and the system is solved at a new time point $t_i + 1$ so that $t_i < t_i + 1$. This procedure is repeated until the entire transient analysis is computed. During a transient simulation the circuit simulator SPICE spends up to 90 percent of its CPU time in three phases of the previous algorithm. These phases are as follows:

- Load Phase — This phase consists of loading the matrix and the right-hand side of the system of linear equations obtained as described above. Device-model equations and linearization errors are also computed in this phase.
- LU Factorization Phase — This phase consists of factoring the matrix of the system of linear equations into the product of a lower triangular matrix and an upper triangular matrix. This factorization is used to solve the system of linear equations.
- Local Truncation Error Phase — This phase consists of computing the local truncation error committed at each time step.

The modifications for parallel processing made in these three phases are described next.

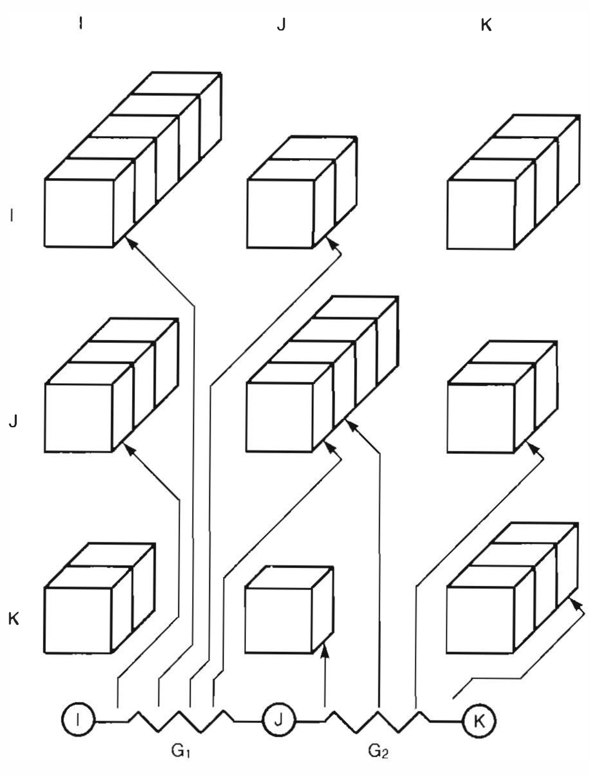
```
time = 0
DO WHILE ( time < finish time )
   discretize differential equations
   DO WHILE ( not converged )
      linearize algebraic equations
      solve linear equations
      check convergence
   ENDDO
   IF ( local truncation error too big ) THEN
      reduce time
   ELSE
      save results at this time
      advance time
   ENDIF
ENDDO
```

Figure 3 Transient Analysis Algorithm for SPICE

Load Phase

In the load phase each circuit element computes and loads all its contributions to the matrix and the right-hand side of the linear system obtained from the circuit equations. Several distinct elements may contribute to the same matrix or right-hand side entry. This means that the matrix and right-hand side are critical sections in the load phase, and access to them needs to be controlled. One approach to synchronize accesses to the matrix is to use a single lock on the whole matrix.[6] In this case only one processor can write into the matrix at a given time, leading to contention for shared resources and decreased efficiency.

In our approach locking the entire matrix is avoided by creating an additional data structure to store each individual element contribution. This structure can be viewed as a three-dimensional matrix whose third dimension is used to store each individual element contribution to a given circuit-matrix entry. Figure 4 depicts such a matrix. There is no unused memory in this structure because it has a variable depth in its third dimension. Nevertheless, using this structure will increase the memory requirements of the simulator. In the design of CAYENNE it was necessary on many occasions to trade memory for speed. Our test examples show that CAYENNE requires an average of 20 percent more data memory than SPICE version 2G5 requires. The contributions for each matrix entry are subsequently summed and loaded in parallel into the circuit matrix. The matrix load is therefore performed in two successive multiple-stream phases.



G_1 - CONDUCTANCE OF FIRST RESISTOR
G_2 - CONDUCTANCE OF SECOND RESISTOR

Figure 4 Three Dimensional Matrix

It is crucial that tasks are evenly distributed among slave processes so that no slave process stays idle while others are computing. A dynamic task allocation was chosen for the first multiple-stream phase of the matrix load. That allocation was preferred to a static task allocation because the time needed to load each element cannot be estimated accurately. Indeed, computation of device models may be bypassed during simulation. The model equations of a device are not computed at a given iteration of the analysis if the voltages applied to this device did not change significantly compared to their values at the previous iteration. This strategy saves CPU time.

Dynamic task allocation is achieved through an array of tasks whose number exceeds the number of slave processes. A task consists of a list of circuit elements to be loaded. Tasks are defined so that each requires approximately the same amount of work. The amount of work needed to load a circuit element is estimated roughly by neglecting bypass and evaluating the CPU time needed to load the element. Dynamic task allocation is expected to minimize any imbalance that may occur during simulation through device model computation bypass.

The task allocation for the second multiple-stream phase of the matrix load is done statically since the work needed to perform this phase can be divided into tasks requiring the same amount of CPU time. The only interlocked access to shared memory during the matrix load is the one on the array index, which defines the next task when dynamic task allocation is used. This index is successively read and incremented by all slave processes.

LU Factorization Phase

The time spent by a direct-method circuit simulator in the load phase is linear in the number of elements, whereas the time spent solving the linear system of equations is superlinear in the size of the matrix.[7] For large circuits the matrix solution part will therefore become more important and will dominate over the load phase.

In SPICE the matrix-solution phase is done using sparse matrix LU factorization. Although full matrices can be factorized efficiently in parallel,[8] the parallel factorization of sparse matrices is more difficult. The LU factorization algorithm has a sequential dependency, and the amount of concurrent work that can be done at each step in a sparse matrix is small.

It is possible to design algorithms that detect the maximum parallelism at each step of the LU factorization. Such algorithms have been used for vectorized circuit simulation.[9] In our environment synchronization is done through software and the fine-grain parallelism used for vectorization may not be efficient. Based on these considerations, we have proposed and implemented an algorithm in which particular care has been taken to minimize the overhead incurred with parallel processing. The details of our algorithm can be found in reference 10.

Local Truncation Error Phase

The parallel computation of the time step does not present major difficulties since the computation of the local truncation error for each energy storage element is independent. Each slave process is assigned a set of energy storage elements and computes the time step required by this set. The master process then computes the minimum time step among the time steps returned by the slave processes. The energy storage elements are statically assigned among slave processes so that the work among them is balanced.

Results

The parallel algorithms described in this paper have been implemented to produce the program CAYENNE. We now present two examples to compare the timing performances of SPICE and CAYENNE.

The first example is the simulation of a MOS arithmetic logic unit (ALU) on a VAX 8800 system. The circuit has 200 nodes and 1350 elements. Twelve hundred Newton Raphson iterations are required for the transient simulation. The efficiency of our parallel implementation is measured in this example. If a multiple-stream phase runs sequentially in an elapsed time T_s and in parallel with N slave processes in an elapsed time T_p, we define the efficiency, E, of the parallel execution by

$$E = (T_s - T_p)/(T_s - T_s/N)$$

E represents the ratio of the actual savings in elapsed time to the potential savings in elapsed time. Table 1 gives timings and efficiencies for the ALU example. As a comparison, SPICE simulates the same circuit in an elapsed time of 834 seconds.

Table 1 Timing Performances and Efficiencies

Phase	CAYENNE 0 Slaves (Seconds)	CAYENNE 2 Slaves (Seconds)	Efficiency (Percent)
Load	694	97	86
LU	22	14	70
LTE	67	35	96
Total Simulation	867	529	—

The second example is the simulation of a MOS control store. The circuit has 160 nodes and 530 elements, and the transient simulation requires 1404 Newton Raphson iterations. SPICE spends 91 percent of the simulation time in the three phases we modified for parallel processing. CAYENNE executing with two slave processes achieves 90-percent efficiency in these phases and simulates the circuit 1.7 times faster than SPICE. For this simulation, CAYENNE on a VAX 8800 runs 9 times faster than SPICE on a VAX-11/780 CPU. Table 2 shows these comparisons.

The efficiencies of a parallel execution of CAYENNE depend on the size of the circuit. Indeed, there is a fixed overhead incurred by

Table 2 Comparison of SPICE and CAYENNE Elapsed Run Times

Case	Elapsed Seconds	Ratio
SPICE on VAX-11/780	3990	9.1
SPICE on VAX 8800	750	1.7
CAYENNE on VAX 8800	440	1.0

calling the synchronization routines JOIN, FORK or JOIN_FORK. The bigger the task performed by the slave processes before a call to a synchronization routine, the smaller the relative cost of synchronization. The simulations of our examples were also run on a lightly loaded system. Loss of efficiency occurs when processors have to be shared with nonrelated processes, and busy-wait synchronizations may waste significant resources. A workload consisting of several independent simulations of equal importance is already decomposed, and CAYENNE should be run in single-process mode. If the turnaround of a single, large simulation needs to be minimized, however, CAYENNE should be run with two slave processes on a dedicated or lightly loaded 8800.

Summary

We have described a general methodology for parallel processing on the VAX 8800 system and a user-friendly set of routines that embed our methodology. We have also presented the successful conversion of the circuit simulator SPICE into the parallel program CAYENNE. New schemes to minimize the overhead of parallel processing and to balance the load among processes contribute to the overall efficiency of our implementation.

Acknowledgments

We would like to acknowledge Bob Kusik for initiating this project, Craig Yankes for introducing us to parallel processing within the VAX/VMS system and for providing us with an initial library of routines from which our methodology evolved, and John Faricelli, Nadim Khalil, Karem Sakallah, and John Sopka for many fruitful discussions.

References

1. R. Hockney and C. Jesshope, "Parallel Computers," (Bristol: Adam Hilger, Ltd., 1981).

2. L. Nagel, "SPICE2. A Computer Program to Simulate Semiconductor Circuits," Memo no. ERL-M520, University of California, Berkeley (May 1975).

3. *Guide to Multiprocessing on VAX/VMS* (Maynard: Digital Equipment Corporation, Order No. AA-HP69A-TE, 1986).

4. S. Farnham, M. Harvey, and K. Morse, "VMS Multiprocessing on the VAX 8800 System," *Digital Technical Journal* (February 1987, this issue): 111–119.

5. *VAX/VMS System Services Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-Z501B-TE, 1986).

6. G. Jacob, A. Newton, and D. Pederson, "Direct Method Circuit Simulation Using Multiprocessors," *Proceedings of the International Symposium on Circuits and Systems* (May 1986): 170–173.

7. A. Newton, "The Simulation of Large Scale Integrated Circuits," *IEEE Transactions on Circuits and Systems,* vol. CAS-26 (September 1979): 741–749.

8. R. Thomas, "Using the Butterfly to Solve Simultaneous Linear Equations," Laboratory Memorandum, Bolt, Beranek, and Newman, Inc. (March 1985).

9. F. Yamamoto and S. Takahashi, "Vectorized LU Decomposition Algorithms for Large Scale Circuit Simulation," *IEEE Transactions on Computer Aided Design,* vol. CAD-4, no. 3 (July 1985): 232–239.

10. G. Bischoff and S. Greenberg, "CAYENNE: A Parallel Implementation of the Circuit Simulator SPICE," *Proceedings of the IEEE International Conference on Computer Aided Design* (November 1986): 182–185.

Dennis T. Bak

The Impact of VAX 8800 Design Methodology on CAD Development

Contributing to the success of the VAX 8800 project was an integrated CAD environment supporting the hardware design effort. A CAD group dedicated to this single project was chartered to supply a smoothly operating CAD process from initial design conception to final production. The CAD environment evolved through a blending of existing tools available in Digital with new tools developed outside the company. Gaps in the environment were filled through extensive modification of existing tools and new development efforts. The driving force behind the CAD process was a design methodology, radical for its time but second nature now.

Past CAD Development Efforts

Prior to the mid-1970s, logic development efforts within Digital Equipment Corporation were largely done without the extensive use of CAD tools. Hand-drawn schematic diagrams were the primary means of expressing logic designs.

A major advance in design automation took place in the mid-1970s when the Stanford University Design System, or SUDS, began to be used within Digital. SUDS allowed the entry of schematics into and the extraction of net lists from a graphics database. Although it was a major step forward in the automation of design processes, SUDS required significant user training and experience to become an effective tool.

Building a SUDS database capable of being used by a computer opened a new avenue for the evolving CAD groups to automate their design processes. These groups soon developed a large body of programs to support net-list extraction, design analysis, placement and routing, and eventually manufacturing parts-lists generation. Simulation tools were developed to help verify the operations of a design before any actual hardware was available. The increased complexity of design drove CAD developers to provide more powerful CAD tools. In turn, logic designers soon grew increasingly dependent on CAD tools as their capabilities increased.

The design methodologies and the CAD tool suite evolved to support large-CPU designs, such as the VAX 8600 family. SUDS eased the burden of entering and coping with design changes; however, the actual contents of its schematics differed little from those of the earlier hand-drawn ones. In large part the schematics entered by designers into SUDS correlated directly with the physical entity being built, showing all components and their pins.

At the inception of the VAX 8800 project in the early 1980s, a vast collection of CAD tools, written by many internal groups, had sprung up. Most of these tools required large ASCII data files and significant manual intervention by CAD experts. Although many aids were provided to develop design processes, they lacked the cohesiveness and simplicity needed to put a process directly into the hands of the designers.

At about this time, a number of significant advances were made in CAD technology. Engineering workstations were announced at prices that made it practical to put them directly into the hands of designers. Moreover, new design methodologies, such as structured computer-aided logic design, or SCALD, were also developed.[1]

These methodologies could significantly improve the quality of design while decreasing the time to develop complex systems. Therefore, Digital made a commitment to use those methodologies on the VAX 8800 project to produce not only the product but a more productive way of developing it.

Design Methodology

The development of CAD tools for the VAX 8800 project was a considerable challenge to the CAD designers. The complexity of the VAX 8800 design, with its particular gate array implementation, demanded that the design quality be high before anything was committed to hardware. In fact, the project managers made a radical (for its time) commitment to simulate the entire design and verify its timing before *any* hardware was built. Therefore, the CAD process had to be designed to meet not only that goal but also to facilitate the rapid production of hardware once the design had proven acceptable. This section of the paper describes the methodology we followed to make the best use of our CAD tools. The next section describes those tools and how they were used.

The tool suite that evolved, pictured in Figure 1, supported both logical and physical design processes with checks and balances to ensure that the design topologies remained the same. Schematic diagrams, captured at an engineering workstation, were processed into a logical net list that was used by the simulation and verification tools. Once a logical design reached a certain level of maturity, it was mapped into a physical design. At that point a physical analysis, to determine delays and signal integrity, was performed. Placement and routing tools were then run to further refine the design. The part of the physical design database that represented the logical topology was then passed back to the logical side of the design process. There, a comparison was made to ensure that the physical and logical designs were congruent. The results of simulations based on the physical design were also passed to the logical process for comparison with the simulations based on the logical design. These mechanisms provided the primary checks to ensure that the logical design matched the physical one.

We decided that the best way to assure success was to develop a complete paper specification of the machine to be built. Once the overall goals for the machine had been established,

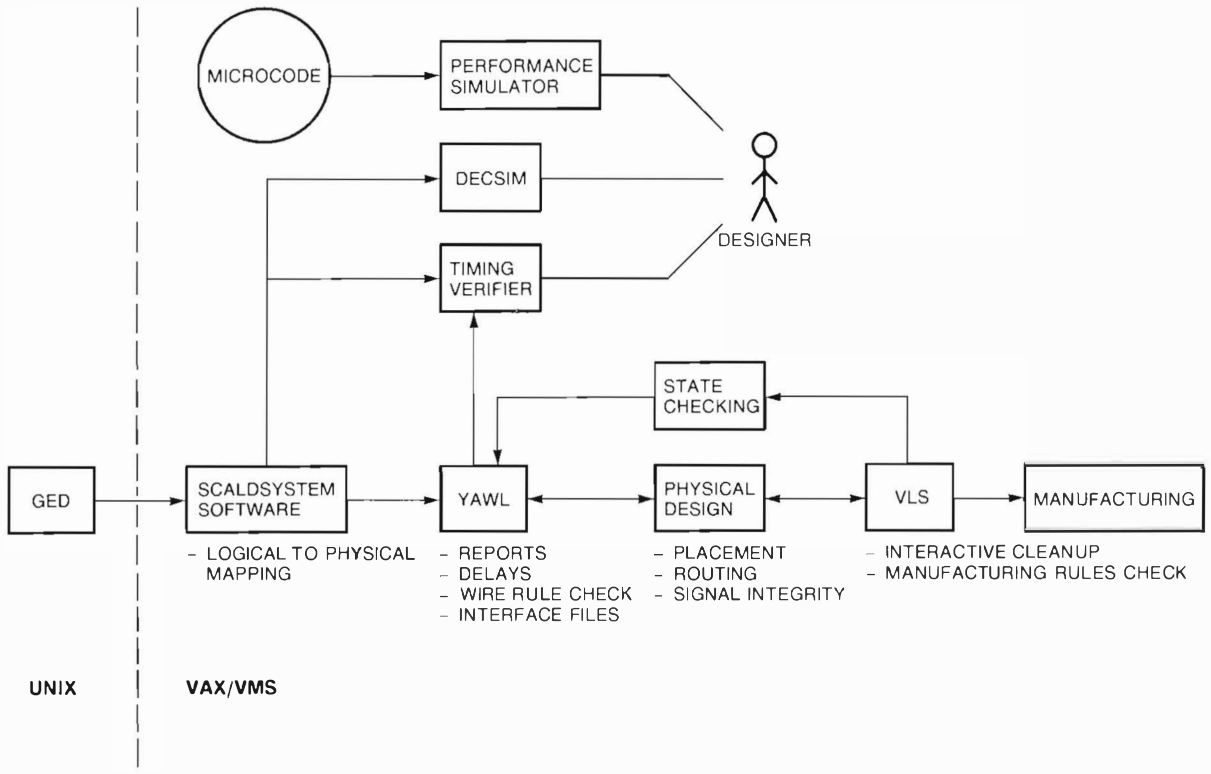


Figure 1 CAD Tool Suite

the designers developed the specifications for each major logic section. This high-level logical design was then partitioned into functions required within modules and gate arrays. These primary interfaces were specified before any detailed logic was developed. As it turned out, that partitioning remained relatively intact throughout the project.

The next step was to develop probe designs and abstract models for the most complex parts of the machine. These designs and models tested whether or not particular logic functions could be developed and timing constraints met. In some cases the probe designs were carried through to the actual fabrications of gate arrays or modules. This continuity allowed us to test the limitations of the selected ECL technology as well as the logic design.

The probe designs proved useful in many ways to both the designers and the CAD developers. The designers were able to verify that their logic implementations would work. The CAD developers were able to use the designs as test cases to develop and debug processes. These test cases proved to be critical to the project's success, especially when the finished design was given to the manufacturing organization. The process was so smooth, in fact, that designs flowed through it with few problems.

The Influence of SCALD

At the onset of the VAX 8800 project, we investigated the tools available within Digital for building a process to support the evolving design methodology. This study lead the CAD team to explore several systems being developed by other companies. One system being developed by Valid Logic, Inc., the SCALDSystem CAD system, was procured by Digital. This system put the power of dedicated engineering workstations directly into the hands of logic designers. Of equal importance was the fact that the SCALDSystem CAD tools were being developed by the same people who conceived the SCALD approach to hardware design.

Logical schematics, requiring almost no information about the physical design, were entered into the SCALDSystem database. These schematics were entered in a hierarchical manner through an easy-to-learn graphical system. Such an arrangement encouraged the designers to avoid the creation of paper schematics by transferring their concepts directly to the workstation screens.

The decomposition of the design was from the top down, but the actual entry of design data occurred simultaneously at many levels. A "design tree" evolved in which cells forming gate arrays were merged onto modules that plugged into the backplane to form a system. The logical design was entered via the SCALDSystem tools onto schematics. The physical implementation of that logical design was left to the physical design tools.

Simulation and Timing Verification

Simulation on the VAX 8800 project was approached from two different viewpoints. The first aimed to determine whether or not the performance goals of the proposed microarchitecture were within the necessary range, as specified by the project's needs.[2] This simulation started early in the project before any detailed logic design had been completed. Once those performance goals had been verified, the second level of simulation focused on the logic design as it evolved.

The designers could verify that each piece of the design functioned as specified while that piece was being developed. As the design tree evolved, the number of logic levels given to the simulation tools increased until the entire logic design had been entered. At this point the designers actually had the equivalent of a software breadboard of the entire VAX 8800 processor. Microcoded instructions were "running" on this software breadboard long before any hardware was available.

The ability to run instruction streams on the breadboard gave the project several advantages. Logic designers could debug their logic concurrent with the microcode developers verifying their microcode. Moreover, the diagnostics engineers could write as well as debug significant numbers of microdiagnostics much earlier than was usual in a design project. The early completion of those diagnostics allowed the first available hardware to be checked thoroughly.

Making the design logically correct through simulation did not ensure that the machine would work at the desired cycle time. In the

ECL technology used in the VAX 8800, signal timing was critical. Therefore, a timing verifier, part of the SCALDSystem tools, was used to ascertain whether or not the timing goals were being met.

It was within the timing verifier that the influence of the physical implementation on the logical design was first felt. The logic designers had to ensure that the placement of gates and routing of signals was optimal for all critical elements. Delay information was then extracted from the physical design and fed back to the timing verifier.

Physical Design

As the logical design evolved, we developed a CAD process to convert it rapidly into a physical design. A set of automatic placement and routing tools, together with delay-estimation and signal-integrity tools, was used to give feedback to the designers. The important question here was whether or not they could build physical representations of their logic designs. These tools also passed data to the timing verifier, which analyzed the effect of the physical design on circuit timings.

Since all the logic had to be verified before any hardware was fabricated, all processes had to be designed to handle a large number of designs in parallel. The relevant Digital manufacturing facilities and outside vendors were acquainted with the physical design through the test cases rather than through an actual prototype. Thus the facilities and vendors could configure and debug their own manufacturing processes before any completed physical designs were sent to them.

To ensure a smooth transition into the fabrication phase, manufacturing engineers were assigned to work directly with the designers early in the design process. Thus these engineers became familiar with the VAX 8800 technology and the machine as it evolved. This was an important step because our manufacturing organization was to build all the hardware, including the prototypes. This early acquaintance with the design allowed them to develop manufacturing processes to support the rapid change to full volume shipments soon after the VAX 8800 system was announced.[3]

Computational Resources

One of the largest VAXcluster systems ever built was assembled to support the VAX 8800 project. This cluster consisted of 14 VAX-11/780 and VAX-11/785 systems with over 20 gigabytes of mass storage. Even this large amount of storage was inadequate at times to support the demands of the databases. Forecasting the computational requirements of this project proved difficult. The VAXcluster system provided the computational power and flexibility to grow as the demands increased.

The availability of sufficient computational resources was critical to the success of our project. The design methodology of extensive simulation was effective only with reasonable program run times. Once the design was verified, large numbers of physical designs were released for fabrication within a short period, which consumed significant computational and storage resources.

The Tool Suite

Design Data Management

A design data management (DDM) system was developed to organize the many files that contained the actual design data. At the heart of that system was the concept of a "design object." This object was some functional piece of the design, usually conforming to the physical partitioning. For example, each gate array and module in the system was defined as a design object. For each object we developed a hierarchy of subdirectories within the VMS file system. This separation of data files into subdirectories allowed various tools within the CAD process to know where to find input files and to write output files.

The design database was continually churning with new information. To give a stable picture as the overall design evolved, a "snapshot" of a design object could be taken at any time, thus generating a revision of the design object. New subdirectory file trees were then created for each revision. Using this scheme a designer could create a "frozen" revision of a design. He could then use that revision for simulations or other activities while changes were being made to another revision of the design.

The relationships between design objects were defined within a revision-matrix file kept with each file tree. This file defined the system-level hierarchy of the machine: which design objects were subordinate to a given object. Using this file a designer working on a module design could select frozen revisions of the gate array designs on that module and be assured of not having them changed as he worked on it.

Another facility provided by the DDM system was a user interface to the design environment. This interface consisted of a simple command language for transversing the design trees and for running specific tools. Since these tools required a large number of input variables, we established a system of default parameters to minimize user input. For cases in which those defaults proved inadequate, users or CAD developers could change parameters to meet the design's needs.

Schematic Capture

Using the ValidGED editor, logic schematics were entered directly into the workstations by the designers. The extracted wire lists were then transferred from the SCALDSystem UNIX-based workstation through a communications port to the VAXcluster system. The workstations were also interconnected in a networking environment, thus providing communication between them. To ease the burden on designers to learn multiple operating systems, only graphical data entry was permitted on the workstations. All the other CAD tools were run in the more native VAXcluster environment.

Since the majority of a designer's time was spent interacting with CAD tools on the VAXcluster system, there was no need for each designer to have a dedicated workstation for schematic capture. The ratio of designers to workstations of about two to one proved adequate. The easily learned GED editor supported a rapid increase in the number of nondesigners — managers, secretaries, and documentation writers — in the user community. All were drawn to the system by the ease of graphical data creation. Eventually, this documentation activity accounted for the majority of workstation usage.

Simulation and Timing Verification

Another proprietary tool, called the DECSIM system, was the primary simulator used on the project. This system supported mixed-level simulations, both structural and behavioral. The logical design was transferred hierarchically to the DECSIM system. This system allowed the designers to deal with complex designs by viewing the simulation in the same hierarchical form as the schematics. For complex devices, such as multiplier chips and RAM devices, behavioral models were developed. These more efficient models increased the overall performance of the simulations. In the case of RAM devices, abstracting to a behavioral model also allowed the microcoded instructions to be loaded efficiently.

Complementing the functional simulation facilities of DECSIM system was the timing verifier (TV) in the SCALDSystem tools. TV analyzed circuit timings to ensure that the design would work under worst-case conditions at the desired clock rate.

Wire delays are a major factor to be taken into account by timing verification. The placement of the physical gates was critical to minimize the wire lengths and hence the delays. Since the placement was not available in the initial design phases, statistical delays based on loading were used. As placement information became plentiful, the latest refined delays were sent to the timing verifier. When the physical design had been completed, delays based on routed lengths were used. If the required timing was not met at any point in the process, the offending circuits were redesigned or the layout was changed to correct the problem.

Wirelisting and State Maintenance

The logic gates entered on schematics by the designers were, in general, assigned to physical components by the CAD process. This mapping occurred initially within the SCALDSystem postprocessor software using a random gate-to-component assignment. This random packaging was then fed into a system called YAWL (for Yet Another WireLister). YAWL acted as a general-purpose wirelister, generating interfaces to many tools and accepting feedback from the physical design tools.

As the physical design process refined the gate assignment, YAWL ensured that the logical design topology did not change. By accepting feedback data from the placement and routing tools and the physical design system, YAWL caught any illegal changes that would have altered the logic functions.

Eventually, the complexity of maintaining the state became so large that YAWL alone could not cope with it. Therefore, several other programs were placed in the feedback loop from the physical design tools to detect changes made in the process of manually cleaning up the physical design. These programs were needed since, even at that late stage, a designer could still add logic to the design. The CAD process therefore had to handle these additions as well as to detect illegal transformations to the logic. The resolution of these changes took a lot of resources, both in terms of time and computer power.

In addition to being the state maintainer, YAWL acted as a primary source of the design data needed for the remainder of the CAD process. YAWL created many reports to inform designers of problems between their logical and physical designs. Most of the interface files in the CAD process were either read, written, or both, from YAWL, which played a key role in the overall process.

Placement and Routing

Two processes were developed for the placement and routing of gate-array and module designs. The gate array process was highly automated, requiring a minimum of interaction by the designers. The process was organized to make several runs from which a designer could select the one that best optimized his logic design.

The bounded problem of placement and routing within a gate array was easy to solve in comparison to the module designs. Here the constraints placed by designers, the limitations of tools, and the complexities of design required extensive human intervention.

Analysis tools were used extensively to assist in determining the quality of design at the two design levels: gate arrays and modules. These tools analyzed such factors as thermal dissipation, signal integrity, and crosstalk. The constraints defined in these tools and in the extensive design-rule checkers were met, thus ensuring a high-quality design.

Most of the tools used for the physical design were developed within Digital. Those developed outside the VAX 8800 CAD group were modified, sometimes extensively, to meet the needs of the project.

Physical Design and Manufacturing Interface

A proprietary physical design system, called the VAX layout system (VLS), was used for the final physical design tasks. VLS took the physical design, as given by the placement and routing tools, and added the data required to manufacture the design. A layout designer, through the VLS interactive graphics system, could manually complete the routing that could not be handled by the automatic tools. Some additional parts that were necessary for fabrication, such as handles for modules, were also added at this time. The net result was a complete design, specified so that it could be used to manufacture the product.

The design data was then collected to form a release package. To keep track of the formal release of design data, a system called POST was developed by the CAD group. POST provided an on-line database, which any member of the project team could query to determine the release status of a design.

Problems Imposed by the Design Methodology

Up to this point, we have described the basics of the design methodology used to develop the VAX 8800 system and some highlights of the CAD tools supporting that methodology. As mentioned earlier, the CAD process was placed directly into the hands of the designers. Thus a tight coupling was established between the process of design and the design process. This coupling posed several major problems, as now described, for the CAD group.

Training

With direct control of a process or tool given to the designers, they all now needed extensive training. On previous projects, one highly knowledgeable individual could run a tool; now, there were 30 or so novice users all learning to use that same tool. Extensive support for those users, in terms of both trainers and documentation, had to be provided.

In most cases the designers quickly learned how to utilize the tools. In a few cases — the placement of modules in particular — placement experts were needed owing to the specialized nature of the task. In summary, the extent of the support required by users was greater than anticipated.

State Maintenance

The task of state maintenance proved to be extremely complex owing to the freedom given to designers to make changes at almost any point in the design process. To ensure that the logical and physical designs matched, it was necessary to do a complete isomorphic comparison of the physical topology against the logical topology of the design.

Logical Prints

The schematics generated by the designers at their workstations represented the logical design, not the physical one. Certain features available in the SCALDSystem tools, such as vectorized signals and gates, allowed it to produce a concise representation of the logic. This came, however, at the expense of not putting physical data back onto the print set. For reasons of state maintenance, we were also unable to restructure a print set once mapped to a physical implementation. Both these factors contributed to a print set that appeared quite different from those generated by previous projects.

Logical print sets, while initially envisioned as being beneficial, later caused problems in documenting the designs. This was particularly true for module-level designs for which training was needed so that groups outside the project team could interpret the new symbology.

Cross References

Using logical print sets alone, a technician could not probe a pin of the physical boards. Since an abstract mapping took place in the CAD process, it was necessary to develop an extensive set of cross references showing the mapping of the logical to the physical design. These cross references proved to be cumbersome and, when printed, consumed vast amounts of paper.

Libraries

CAD tools run on libraries, and each major tool has its own format for library data. These libraries must be consistent across the entire process. Despite all the safeguards built into the process, we found that inconsistencies still crept back into the database. Discovering and eliminating those inconsistencies, many of which were found late in the project, consumed a lot of time.

Summary

Both the design methodology and the CAD process supporting the VAX 8800 project were quite successful. The first prototype hardware delivered to us worked as expected. We found only a small number of hardware problems during the prototype debug phase of the project. Most of those problems were in areas that had not had extensive simulation or timing verification.

Some general conclusions reached from the VAX 8800 project can help future CAD designers to improve their tools.

- A close coupling from the start, both physically and organizationally, between all groups associated with the project leads to the development of a smooth process flow.
- The design methodology has a direct and far-reaching impact on the CAD process. The capabilities of CAD tools directly affect the design methodology.
- Extensive simulation and timing verification before fabrication can help to achieve a high-quality product.
- The impact of radical changes (e.g., in the data content of schematics) must be appreciated and then taken into account by all project members.

In future projects we will focus on reducing the process-loop times and enhancing the capabilities of the simulation and timing verification tools. It will be easier to function in future design environments, and more tools will be placed directly into the hands of the designers. The design methodology will be modified to make the resolution of the design state easier and therefore faster.

References

1. Structured Computer Aided Logic Design was developed at Lawrence Livermore Laboratories and applied there to the design of the S1 computer.

2. C. Wiecek, "The Simulation of Processor Performance for the VAX 8800 Family," *Digital Technical Journal* (February 1987, this issue): 100–110.

3. A. Matthews, "On-line Manufacturing Data Access on the VAX 8800 Project," *Digital Technical Journal* (February 1987, this issue): 136–141.

Andrew J. Matthews

On-line Manufacturing Data Access on the VAX 8800 Project

Previously, the transition from design to manufacture involved transferring significant amounts of data on paper. To minimize product start-up time, the VAX 8800 project used an on-line system that eliminated much of the paper. The key task was transforming the data from existing CAD tools with different formats into manufacturing data. Two generic types of VMS files, DATA and DRAWING, contained data for each Part Number and Revision Number. VMS's subdirectory and access-control capabilities provided total revision control. Manufacturing engineers pulled files at will using DATA files to drive their processes and viewing DRAWING files from VAXstation II workstations.

A key objective for the VAX 8800 project was to go from the completed design to full-volume manufacture in the shortest possible time. In the past, delays have often occurred in the transition from Design Engineering to Manufacturing. Therefore, to achieve our goal, we had to eliminate or minimize those delays.

We knew of a number of ways to speed up this transition phase. Since there is normally a tremendous flow of data on paper between Engineering and Manufacturing, one way was to eliminate the paper itself. A second way was to accelerate the controlled revision process when changes were required. And a third way was to accelerate the query-and-response process that was necessary to solve specification problems. One can see right away that these activities involve many people and consume significant resources. Therefore, a formal project was established to determine how best to implement the three ways to minimize delays.

The project team determined that although the data flowing between Engineering and Manufacturing was vital, the paper itself was not. Thus the team's goal was to find out how to establish a paperless, but not drawingless, scheme to pass that information between the two organizations. The team also set some constraints on this scheme. First, existing data techniques should be used whenever possible rather than developing new ones. Second, Manufacturing should be free to obtain data as required rather than have Engineering "push" it to them. Third, any intermediate data processing functions and groups, which all have priorities and queues of their own, should be bypassed. Finally, the data had to be organized in the way Manufacturing needed it, that is, by Part Number and Revision, among others. Therefore, some translation process had to take place between the data sources in Engineering and the data repositories used by Manufacturing.

The data sources in Design Engineering are many and varied. Digital uses a large set of CAD tools in its design processes.[1] These tools use a variety of methods to gather, store, and manipulate data. The databases associated with these tools are the sources for all the specifications conveyed to Manufacturing as plans and drawings. Manufacturing also has its own set of CAM tools used in various processes.

The primary CAD and CAM process tools did not communicate since they were all based on different data formats and revision procedures. The primary goal of the project was to take the design data created by the CAD tools and, with as little paper as possible, turn it into manufacturing data that could be used by the various manufacturing groups. The direct way that goal could be accomplished was to create an integrated source of data as VMS files that would be available on line to engineers in Manufacturing. This capability of data transfer was called manufacturing data access, or MDA.

As typically happens in a rapidly evolving technological environment, the standard data-transfer processes already in place had rapidly become outdated. The result was that the standard process was handling only part of the data, and informal systems evolved to deliver the remainder. MDA had to identify all these data processes, regardless of their sources. Then, it had to provide all the data needed to build and test the product through a consistent on-line process. That task was accomplished by "reverse engineering" the existing processes. All the process managers responsible for the product in Manufacturing were interviewed to find out what data they were receiving by both formal and informal means. They were asked, in particular, what additional data they needed. The result was a lengthy list of data files, most of which existed or could be easily generated.

One key limitation to this type of data-generation process was the availability of an appropriate engineering database. For example, a visual-inspection process might need the color of a component, but this data may not be in any engineering database. Therefore, some manufacturing data processes would have to continue using other sources, typically libraries of additional information, as well as the engineering database.

The objective of MDA was to provide on line all the data needed for new product start-up. The problem, as noted earlier, was that this data was derived from many different files used by the CAD tools. These separate software tools, having come from many sources at different times, generally operate on independent VMS files and do not yet utilize complex, integrated database capabilities. Therefore, another primary goal of the MDA project was to bring appropriate data management to these existing processes, but at the same time not to require significant changes within them.

Given this VMS file environment, the team made an early decision that the VMS system could provide the framework for comprehensive data management and organization capabilities if full advantage were taken of the possibilities inherent in the system. That is, files and directories, subdirectory schemes, and access control lists had to be used effectively. The advantages of using VMS features for these existing files rather than implementing a specialized data-management scheme were numerous. This procedure meant that these capabilities would be immediately accessible to all of Digital's VAX users, could be readily linked to existing read and write processes for CAD/CAM files, and would require no unique training, software, or hardware.

The remainder of this paper describes the approach that MDA takes to achieve an integrated source of manufacturing data. As a first-generation paperless process, MDA was used on the VAX 8800 project with great success. We anticipate that MDA could evolve at a later date into a second-generation paperless process. In this process, users in Manufacturing would be able to selectively compose and generate any desired drawing from the databases. For the first design of MDA, however, that was too sophisticated a solution to be applied to a broad manufacturing community still in transition from paper processes.

MDA Capabilities

We designated the files containing the data that drives the computer-aided processes in Manufacturing as DATA files. Every drawing sheet in the full drawing package is electronically released as a plot file. These on-line files, called DRAWING files, are effectively the master drawings, and any locally generated paper prints are temporary working copies. DRAWING files are intended only for human interpretation (viewing or plotting); they do not have to be interpreted as structured data by other functional-process software. DATA files are used for that purpose.

Both DATA and DRAWING files are made available through a single unified process available anywhere on Digital's world-wide internal DECnet network. Data security is provided in the software by an access control list of specifically authorized users in Manufacturing. A list method rather than password control was chosen since the VMS system has all the capabilities to implement list control (identifying remote users). Control over access to the on-line product database remains with the data managers.

The files are organized around the Part Number and Revision Number of the physical object. A complete DATA and DRAWING file set is provided for each revision, thus leading to a degree of redundancy between files. We originally considered solving this redundant-data problem in the traditional CAD/CAM way by defining separate universal interface files and designing integrated databases from which any needed file could be extracted. To achieve the primary goal of minimizing all delays in product data transfers, however, we concluded that providing the process specific, but redundant, files needed directly in Manufacturing was worth the price.

This technique eliminated all hand-off delays and allowed the already proven processes to operate efficiently. Of course, the risk was that data in the redundant files could in some way diverge. Therefore, Engineering assumed the responsibility of verifying that the data was consistent between them. Engineering uses special software to verify that all files in a set, some of which come from different CAD tools, represent the identical design object and revision state.

The DATA files utilized are those the start-up team identified as being directly needed for each manufacturing process. Our ideal target for DATA files was the specific data set needed by a "work cell" of the manufacturing plant; this typically includes both a computer resource and specific people that together receive and adapt the generic data to the immediate needs of their particular plant and process. To minimize the process start-up time, eliminate queues, and assign responsibilities clearly, MDA avoided using intermediate data formats. These formats historically required preprocessing by some third party before they could be used in the plant. We expected the plants to adapt the DATA files to the specific needs of their own processes. For sophisticated data consumers with complex manufacturing needs, the source-data design files are also included with the on-line data.

The practical realities of the many CAD/CAM processes in use first required a smoothly operating file-management process. A large number of files are required to support the build-and-test processes for one designed object. A typical Digital part (e.g., a complex CPU logic module) is today completely specified by 50 to 70 DATA files and 30 to 50 DRAWING files. With that many files involved, a key to success for this type of file management is total data acquisition. Thus the process was made mandatory (not voluntary); that is, it could not depend on someone's remembering to do something. The only way to accomplish complete data acquisition was to integrate the data-management process with the CAD tools that generated the source files.

The principal MDA implementation concept was to use the extensive VMS subdirectories that "belonged" to each object and revision and then collect all the appropriate files into the appropriate directories. This technique makes possible a user data-access process based directly on the VMS system in which a user can answer several questions about the object or revision for which data is needed. MDA then provides him with a directory containing the files relevant to the requested object or revision. This directory represents the bounded set of data. Within that set each DATA and DRAWING file is "named" so that it is completely identified even if moved later to other manufacturing locations. The file-naming scheme is also not cryptic so that manufacturing users can specify and recognize the particular files they need.

An underlying objective of the MDA program was to provide an environment in which a released data file was perceived as being as stable as an approved and released paper drawing. Whenever a set of DATA and DRAWING files for a given revision of an object are released, that set of data becomes "read-only" and is placed under strict control. The engineering group will not modify any file within the set belonging to that revision, and subsequent revisions of that object do not overwrite prior revisions.

MDA allows users to pull data selectively as it is needed rather than pushing it automatically to predetermined receivers. The strategy here is to deliver not data, but automatically generated notification messages on Digital's electronic VAXmail system. The generation of mail is tied to the design-management functions of the hardware designers and the coordinators for engineering change orders (ECOs). The mail messages are sent to designated representatives in any of the manufacturing plants around the world to inform them to pull whatever data they require from the on-line system. Data users in Manufacturing are notified by automatic messages whenever new data is issued or when the status of existing data changes. This method takes advantage of the existing VMS Mail facilities for identifying remote users. A user access-control list has been implemented, and all user transactions are logged. These techniques confirm that new data has been received by users and provide an audit trail of who accessed particular data in case an error is discovered later.

Much of the data provided for the product is intended for the specific assembly and test processes implemented by the start-up team. Provision of this data is made possible by the close coupling of the Engineering Design and start-up team efforts and the sophistication of the data-driven fabrication and test processes. In other words, the designs of high-technology products are now aimed at specific manufacturing processes for assembly and test. Except for simple dimensional data, much of this product data can no longer be "post processed" (by software means only) onto a different manufacturing process. A major process alteration might require reconvening the start-up team and adapting the design and data for the new process.

Revision Management

Each revision of a part means that that physical design object has changed in some way. In the MDA process a complete set of DATA and DRAWING files is provided for every revision; there is no implied or referenced data. All active revisions still being built remain on line, and subsequent revisions do not overwrite earlier revisions. If the same DRAWING file applies to different revisions, it will be provided with each of those revisions. We were concerned initially that this simplified approach would generate a large number of redundant files, particularly DRAWING files. However, an analysis of the completed sets showed that, with the CAD design processes in use, only 10 to 20 percent of the files were unchanged from one physical revision to the next. Our conclusion now is that having some redundant files is a cheap price for the benefit and simplicity of having full data sets. Thus no data set has to reference data from another set, and old revisions can be readily archived.

The MDA process currently has one significant limitation. Unlike the existing procedures for paper drawings, there is no standard control process for putting a formal revision on a DATA file. On the other hand, it is not clear that a control process is sufficiently valuable in a product environment that is totally data driven. Traditionally, when necessary, a paper drawing can be changed separate from the physical revision of the object itself. That cannot currently be done for DATA files since there are no standard procedures that are equivalently recognized for naming them or for controlling revisions. If the DATA files really define the physical product, then an erroneous data file defines the wrong physical product. In that case, it can be argued, the right way to signify the change is to update the revision of the object itself. At the present time, if an incorrect DATA file is included in the released data set, the only unequivocal way to correct that problem is to advance the physical revision and generate a new set of data.

Within the MDA process, the status of any file is specifically marked. (The mere existence of the file within the process does not imply any particular status.) Typical categories of status are verified, issued, released, and obsolete. A status is implemented by using the file-ownership capabilities within the VMS system. As its name implies, MDA provides on-line access to all needed data and drawings for any and all revisions. However, the formal status (preliminary, released, etc.) of each part and revision available on line is controlled and specified by other existing standard procedures. That status is confirmed by MDA but cannot be determined solely from the status information that MDA provides on line with the data.

The MDA process is not directly coupled to the control procedures in Manufacturing, but is linked directly with status-setting activities in Engineering. For example, the issued status is set by a procedure run by the product's ECO coordinator when he issues an ECO package to his counterpart in the manufacturing plant. Therefore, the data users in Manufacturing are advised to use the displayed status only as confirmation of a change; they will continue to be notified first through the existing ECO control procedures.

Thus, MDA has on-line data available for a manufacturing activity when Manufacturing is notified, by means external to the MDA process, that they should be building a particular revision. Also, MDA provides no on-line information about such things as the interactions and relationships between revisions, which revisions of the modules go together, and which revisions go with which backplane revisions. Therefore, although MDA is a comprehensive data-management and access process, it is not also a true configuration-control and revision-management process.

Directories and File Names

Within the MDA process, the DATA and DRAWING files are managed by grouping them in VMS subdirectories for the object that these files specify. The subdirectories are tied to a common-root directory to facilitate the management of the overall physical data on the host (e.g., moving various directory structures between disk drives). The directory files themselves are owned by the data-management process. They may not be read directly over the network; the access process provided must be used. In pictorial form, the directory structure is described in Figure 1.

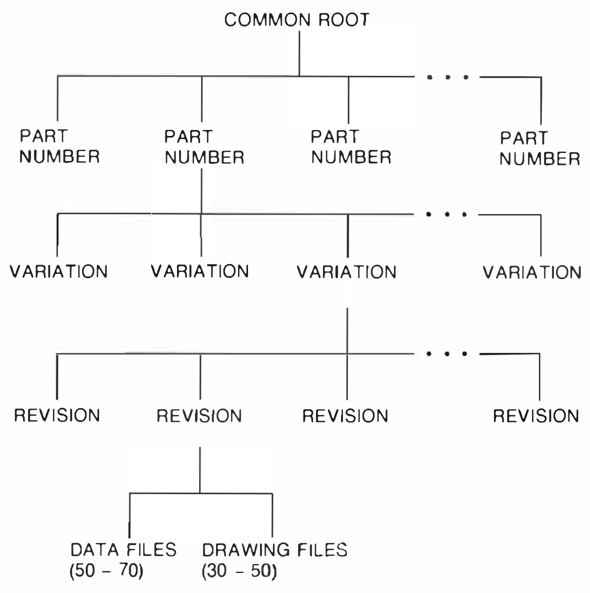


Figure 1 VMS Directory Structure

The name of each DRAWING file is tied directly to the Digital drawing number plotted by that file. For multisheet drawings, a plot file is made for every sheet in the complete drawing package, so there is a one-to-one correspondence between DRAWING files and drawing sheets. The files are named to match exactly the title block of the drawing sheet. A typical DRAWING file name is depicted in Figure 2.

For DATA files, a different strategy for file names was necessary since, unlike the DRAWING files, a one-to-one linkage does not exist. A DATA file relates to the physical object it defines; therefore, the file name defines the exact part to which that file applies as well as

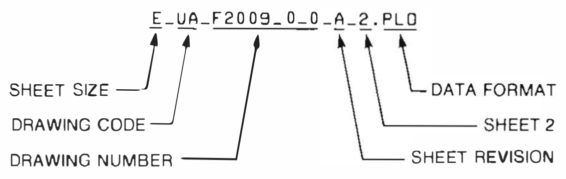


Figure 2 Typical DRAWING File Name

the file's specific content and format. File names must also continue to completely identify the files after they have been extracted from the MDA management process and moved to Manufacturing. Therefore, part of the file name is actually redundant with the MDA directory name. These file names can become extremely long, and although reading them is not a problem, typing them is. Thus the file names are automatically generated, and users can select them from menus. The name of a typical DATA file is structured as in Figure 3.

Since there were many DATA and DRAWING files, the file-naming scheme also permits the creation of a typical VMS "wild card" directory listing for specific types of DATA or DRAWING files. For DATA files, the specific type of process activity supported by that file is included as a unique field in the file name. For DRAWING files, the drawing code is included in the file name, which also implies the likely uses. These fields within file names are then used in Manufacturing to obtain file listings specific to an activity; wild-card directory listing is by far the most common style of use.

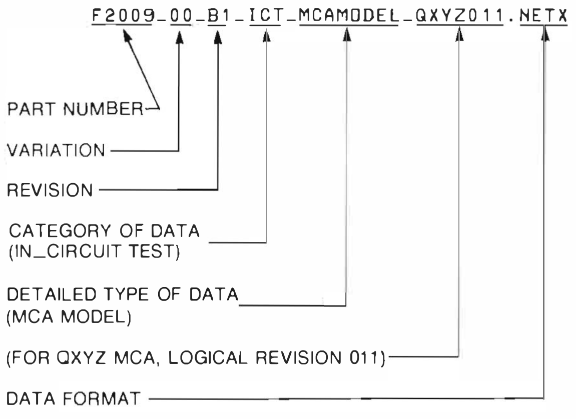

F2009_00_B1_ICT_MCAMODEL_QXYZ011.NETX
PART NUMBER
VARIATION
REVISION
CATEGORY OF DATA (IN_CIRCUIT TEST)
DETAILED TYPE OF DATA (MCA MODEL)
(FOR QXYZ MCA, LOGICAL REVISION 011)
DATA FORMAT

Figure 3 Typical DATA File Name

On-line Data Access

Since all DATA and DRAWING files for each revision of a Part Number are accessible on line, it is a simple process for authorized users to access them. A user first logs on to a captive (limited function) account on a specific host CPU from any system on the Digital's DECnet network. Since this process is controlled by a list of authorized users, no password is necessary. The user never sees the VMS prompt level but is immediately presented with a menu of MDA functions. He is then asked a short series of questions about either the Part Number or Revision Number and is provided with a directory of applicable files.

All user transactions with the data-access process are automatically logged. This logging provides several important capabilities:

- An accurate summary of the actual on-line data usage (which has showed that our initial assumptions were quite incorrect as to who would use what data, and how much access traffic there would be)
- A degree of additional security by tracking all data accesses
- A means to notify all users who have utilized any file in which an error has been found

Electronic Drawing Access, Plotting, and Management

At the present time, most DRAWING files are in the VMS data format of FILE__NAME.PLO since .PLO is the data format that can be released electronically to Digital's on-line drawing-microfilm service. A variety of software packages using this data format are available in each manufacturing plant. We expect to make a transition to a new industry standard when it comes into general use.

Providing each separate drawing sheet as a separate file was the first step toward a paperless process. The second step was to give Manufacturing the ability to view a drawing on a VAXstation workstation, manage drawings, annotate them, send those annotations back to the engineer, and make plots. These basic functions permit Manufacturing to do on line what they would have done previously with paper drawing sheets. Engineering provided some necessary software tools for these functions to expedite the transition to a paperless process in Manufacturing.

The workstation used is the VAXstation II system. The software provides the following capabilities:

- Access drawings directly from the on-line data process
- Create windows for the drawing, and zoom around it
- Annotate a copy of the drawing for use with specific processes
- Return a copy with questions for the responsible engineer
- Submit plot requests automatically for the whole drawing or any selected window to either a large electrostatic plotter or an LN03 Plus printer, both accessible on a local Ethernet link

The process of making snap-shot window plots of specific areas of interest on the LN03 Plus printer has proven to be a very effective capability, and shows some of the possibilities of replacing large sheet paper plots within the Manufacturing functions.

Summary

The MDA process has been operational since the first prototypes of the VAX 8800 system were built. MDA presently maintains approximately three gigabytes of VAX 8800 product data on line, including both prototype and production revisions. More than one hundred users from ten different locations in both Manufacturing and Field Service have logged an average of two hundred transactions per week. Although MDA contains significant amounts of control and verification software, there has been little formal user training. The simplicity of the MDA process allows the on-line Help information to be an effective source of primary documentation.

References

1. D. Bak, "The Impact of VAX 8800 Design Methodology on CAD Development," *Digital Technical Journal* (February 1987, this issue): 129–135.

digital™

VAX

ISBN 1-55558-001-7