□ *VAX 8600 Processor*

Digital Technical Journal

of Digital Equipment Corporation

Editorial Staff

Editor – Richard W. Beane

Production Staff

Production Editor – M. Terri Autieri

Designer – Charlotte A. Bell

Art Director – Gillian S. Cowdery

Typesetting Programmer – James K. Scarsdale

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

Comments on the content of any paper are welcomed. Use the Reader Response card or write to the editor at Mail Stop HLO2-3/K11 at the published-by address.
Comments can also be sent on the ENET to RDVAX::BEANE or on the ARPANET to BEANE%RDVAX.DEC@DECWRL.

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

ISBN 0-932376-83-5

Documentation Number EY-3435E-DP

The manuscript for this book was created using generic coding and, via a translation program, was automatically typeset on DIGITAL's DECset Integrated Publishing System. Book production was done by Educational Services Development and Publishing in Bedford, MA.

Cover Design

Aspects of the VAX 8600 design are featured in this issue. The component technology in the 8600 is the macrocell array, with ECL semiconductors. Our cover shows a module with its various electronic components, notably the macrocell arrays and their multilevel heat sinks.

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

Contents

Editor's Introduction

Richard W. Beane
Editor

The *Digital Technical Journal* bridges a gap in the information published about Digital's products by providing an explanation of their technological foundations. In the past, such explanations appeared in papers written by Digital's engineers for various periodicals. Unfortunately, anyone wanting concise technical details had to search through the gamut of this literature.

This journal was created to present that information in one publication. The papers are written by the engineers who developed the products, in terms of the technologies that went into their designs. Our audience is composed of engineers within Digital, as well as engineering educators and customers.

This issue, our inaugural one, features the VAX 8600 processor. Its design, with a pipelined architecture and emitter-coupled logic, offers many innovations besides increased speed. New packaging, cooling, and reliability techniques, and new automated simulation tools were used to develop this product. Some papers explain the final results of the development process; others discuss the process itself. All give the reader a sense of the unique ways in which Digital develops its products.

The first paper, by Tryggve Fossum, Jim McElroy, and Bill English, is an overview of the 8600's salient features. The distribution of processing into the various "boxes," the specific performance improvements, and the reliability and data integrity features are amply discussed. This paper establishes a framework to assist the reader in fitting the more detailed papers into an overall context.

The pipeline paper, by Mario Troiani, Steve Ching, Nii Quaynor, John Bloem, and Fernando Colon Osorio, explains the VAX 8600 pipeline in terms of a general model. This starting point is important in understanding the unique contribution of the pipeline's key element, the instruction prefetch unit. The paper explains how this unit fetches instructions, achieves control, and maintains data integrity.

The 8600 features fast, efficient floating point operations. The paper by Tryggve Fossum, Bill Grundmann, and Ginny Blaha discusses the instruction flow in the floating point accelerator and the role of emitter-coupled logic in its design. The authors describe how algorithms are processed and how microcode controls those operations.

The next three topics are closely related because decisions in packaging, signaling, and cooling must be made with their interacting effects in mind. The paper on packaging, by Jim McElroy, discusses the evolution of the process that identified the best way to package the modules and components. John Hackenberg's paper on signal integrity describes the software tools that enabled the design team to distribute power while controlling noise and avoiding cooling problems. The solutions to those problems, including the use of thermal design rules and special measuring techniques, are discussed by Brian Kalita and Bill English.

The final paper, by Bill Bruckert and Ron Josephson, explains why reliability consists of the avoidance, tolerance, and minimization of faults, and the improvement of MTTR. The authors discuss the techniques used to reduce failures, to identify those that do occur, and to make repairs easier.

These papers represent a cross section of the activities in a large design project, and they relate the results of design decisions as well as the process for making them.

Dick Beane

Biographies

Virginia C. Blaha Ginny Blaha is a senior engineer, currently working on the datapath design for the execution unit in a new high-performance CPU. On the VAX 8600 project, Ginny designed the multiplier module and several gate arrays as a member of the team that developed the floating point accelerator. She joined Digital in 1981 after receiving a bachelors degree in electrical engineering from Princeton University.

John E. Bloem Educated at Northeastern University (B.S.E.E., 1969 and M.S.C.S., 1973), John Bloem joined Digital in 1969. He first designed custom module systems, then the interfaces for PDP-11 and PDP-8 systems. As a senior engineer, he helped to develop the PDP-11 Commercial Instruction Set. John was the project leader for E Box development on the VAX 8600 project, and he also helped to design and test the I Box. He is presently an engineering manager planning a new high-end computer system.

William F. Bruckert In 1969, Bill Bruckert joined Digital after receiving a B.S.E.E. degree from the University of Massachusetts. Later, he received a M.S.E.E./C.E. degree from the same university in 1981. Starting as a world-wide product support engineer, Bill later worked on a number of PDP-10 system designs as a senior engineer. As a consulting engineer, he developed the cache, memory, and direct memory access designs for the VAX 8600 processor. He is currently investigating cache designs for future memory systems.

S. Stephen Ching Steve Ching is a consulting engineer now developing the design for a new high-end CPU. On the VAX 8600 team, he worked on prototype debugging, code optimization, and simulation development. After joining Digital in 1977, he worked on developing test generation tools in the LSI area and on a simulation system used in VAX system development. Steve earned a B.S.E.E. degree (1972) with honors from California State University and a Ph.D. (1976) in electrical engineering from the University of Missouri.

Fernando C. Colon Osorio Fernando Colon Osorio graduated from the University of Puerto Rico (B.S.E.E., 1970) and the University of Massachusetts (M.S., Ph.D., 1976). Joining Digital in 1976, he worked on several high-end PDP-11 systems and managed the Local Area Network Group in Corporate Research. On the VAX 8600 project, Fernando managed the RT-level simulation and prototype design verification. He is presently the manager of system research and advanced development in Digital's High Performance Systems Group. He was Associate Editor of the *IEEE Transactions on Computers* and is the coauthor of *Engineering Intelligent Systems.*

William English Bill English has been writing technical articles and documentation for over 25 years at Digital and other computer firms in New England. He is currently helping engineers from the High Performance Systems and Clusters Group to write and publish articles on the VAX 8600 project. Bill received the A.B. degree in physics from Harvard University in 1953 and the M.S. degree in mathematics from the University of Massachusetts in 1959. He is a member of Phi Kappa Phi and Sigma Xi National Honor Societies.

Tryggve Fossum Tryggve Fossum received a B.S. degree from the University of Oslo in 1968. Later, he earned his Ph.D. from the University of Illinois in 1972. Tryggve joined Digital in that year and worked on the design of high-end computers, notably the VAX-11/780 system. As a project leader on the VAX 8600 team, he guided the design of the floating point accelerator. He was also responsible for microcode development, memory management, and other aspects of the system's operation. He is currently a project leader working on the design of a high-performance system.

William R. Grundmann As a principal engineer, Bill Grundmann is presently leading a team designing the execution box for a new CPU. He was the logic designer on the team that designed the floating point accelerator for the VAX 8600 processor. He designed the adder module in the FPA as well as several MCAs in the datapath. Bill's other projects at Digital include work on the memory systems in the VAX-11/750 and PDP-11/44 systems. He joined Digital in 1977 after receiving a B.S.E.E./C.S. degree from the University of Connecticut, where he was a member of Eta Kappa Nu.

John H. Hackenberg In 1968, John Hackenberg came to Digital as a technician on the KI-10 project, leaving after two years to serve in the armed forces. He returned in 1971 and worked on the designs for various high-end systems, including the KL-10. John earned a B.S.E.T. degree from the University of Lowell in 1979. As a consulting engineer on the VAX 8600 project, he worked in the area of signal integrity. He is now developing a high-performance gate array in the High Performance Research and Engineering Group.

Ronald E. Josephson Ron Josephson is currently the engineering supervisor for power systems. As a project leader on the VAX 8600 project, he guided the development of the power systems for the processor design. In eight years at Digital, he has also worked on the design of power supplies, in particular the one in the LA-34 terminal and the H-7170 module for Digital's Modular Power Supply. Before joining Digital, Ron worked on the Aegis, Hawk, SP-49, and Patriot programs at Raytheon Corporation. He also teaches electronics at Quinsigamond Community College.

E. Brian Kalita Brian Kalita now works for Encore Computer Corporation in Marlboro, Massachusetts. On the VAX 8600 project, he was an engineering supervisor responsible for completing the mechanical design of the processor. Brian also performed thermal engineering tasks on several high-performance system programs. Before joining Digital, he was an applications engineer at Torin Corporation and a manufacturing engineer at the Torrington Company. Brian has earned B.S.M.E. (1973) and M.S. (1980) in management degrees from Rensselaer Polytechnic Institute.

James B. McElroy Jim McElroy is the manager of advanced development for Digital's High Performance Systems Technology Group. On the VAX 8600 project, he was the manager responsible for power and packaging design. Previously, he managed the Large Systems Power and Packaging organization. Before joining Digital in 1976, Jim worked at RCA for nine years doing packaging and interconnect design for military computer systems. He earned a B.S.M.E. degree from Northeastern University in 1968 and a M.S.M.E. degree from the same university in 1972.

Nii N. Quaynor Earning his B.E. degree from Dartmouth College in 1973 and his Ph.D. from S.U.N.Y. at Stony Brook in 1977, both in computer science, Nii Quaynor joined Digital in 1978. He first worked in corporate research on multimicro systems. In 1982, Nii joined the VAX 8600 project as a consulting software engineer and created models for large-scale CAD applications using a register transfer language. Later, he worked on the verification of the VAX 8600 design. He is now designing models for VAXcluster systems.

Mario Troiani Mario Troiani is a principal engineer working on advanced development for a high-performance processor. On the VAX 8600 project, he helped define the microarchitectural model, generated prototype debug models, and worked on the design validation strategy. On other projects, Mario designed the first T-11-based computer and helped to build a single-chip workstation. Joining Digital in 1977, he designed the test systems for the Module Repair Centers. He received a Dottore in Ingegneria Elettronica (1975, Summa Cum Laude) from the Universita di Trieste and a M.S.E.C.E. degree (1977) from the University of Massachusetts.

Foreword

Robert M. Glorioso
Vice President
High Performance Systems
and Clusters

How appropriate it is that this first issue of the *Digital Technical Journal,* a medium for communicating new technical ideas and results within Digital, should be dedicated to the VAX 8600 system. The 8600 represents the confluence of many new concepts and much good engineering in the areas of implementation architecture, interconnect, packaging, cooling, design methodology and tools, CPU and systems design verification, and complexity management.

The VAX 8600, or VENUS, program approached the problem of producing a high-performance VAX system in two ways. First, we reduced the cycle time by physical means. Second, by incorporating new design techniques, we reduced the average number of cycles required to implement instructions over a wide range of typical uses. The performance range of the 8600 makes it appropriate for customers with requirements close to those provided by mainframes. Therefore, we had to address mainframe reliability, maintainability, and lifetime cost-of-ownership issues from the beginning of the project. For this reason several new concepts had to be integrated into the design.

The key concept of the new physical technology incorporated in the 8600 is the use of ECL gate arrays called macrocell arrays, developed jointly by Digital in Marlboro and Motorola in Phoenix. In order to deal with the speed of ECL, we had to pay special attention to board, connector and backplane impedance and delay, as well as manufacturing problems. Incorporating ECL yielded a cycle time of 80 nanoseconds. Compared to 200 nanoseconds on the VAX-11/780 system, that represents a performance gain of 2.5, which is the minimum gain without architectural improvements.

The architectural challenge in this implementation was to increase the VAX 8600 performance by 1.5 to 2.5 times that of the 11/780 by executing more of the functions of each instruction during every cycle. Meeting this challenge required that the operations of instruction decoding and execution take place in parallel to a greater degree than in any previous VAX implementation. Thus the concept of pipelining became a necessity in the VAX 8600 implementation. Moreover, the higher speeds required different approaches to cache management, memory busing and management, and I/O. In particular, the concept of a "writeback" cache was introduced to reduce the number of times that individual accesses to slower main memory are needed. Furthermore, the memory and I/O buses were separated to allow higher memory bandwidth, which decreases the amount of needed memory, and to avoid I/O interference problems.

The resulting design, which has from 100–200 thousand gates (depending on how gates are counted), introduced new levels of complexity in both design and management that stretched us all into new domains of knowledge and maturity. For example, we discovered quite early in the program that our classic design approach of quickly designing on paper, building prototypes, and debugging them would NOT work. The design turnaround times for the chips alone would have gotten us to market much too late to be competitive. Thus we began the process of simulating, debugging and verifying the 8600 by using other computers instead of moving wires. That process required us to develop new tools for timing analyses, such as AUTODLY, and new methods for building data bases. Moreover, new techniques had to be devised for finding and fixing problems by using tools and libraries instead of real design bugs. And, of course, computer resources had to be identified, ordered, and installed.

Initially we had planned to use four KL-10 systems and a VAX-11/780 system as the computer resources needed for the whole program. We soon found that more machines were needed quickly if we were to succeed with simulation. In the course of the next two years we installed about one new system per month, ending with not only twelve KL-10s but also twelve 11/780s. Simulation was a tremendous challenge to the whole organization and required close cooperation from our partners in other groups, especially manufacturing and CSSE. The former helped us to get equipment and loaned us space, and the latter moved their own work around and loaned us systems and people to complete the simulation and verification. Moreover, networking at a much higher level was then needed and communications between the 11/780s and the KL-10 systems had to be improved. Our Site Resources and Engineering Group had to accommodate these changes, and their capabilities grew continually within the available constraints of time and space.

Finally, a word about the management of the VAX 8600 program. First, I believe we learned a great deal about the management methodologies required to produce a product as complex as the VAX 8600 system. Our fundamental philosophy was open communications at all levels of the project. We fostered the attitude that finding problems, discussing them, and asking for help were signs of intelligence and maturity, not ones of weakness or failure. To succeed, we knew this was the "right thing to do." We also developed a review process that encouraged project members and other groups to see our progress. This process included regular, open reviews for all project levels, weekly program reviews for all groups involved in the project (manufacturing, CSSE, VMS, semiconductors, purchasing, etc.), and monthly reviews for people throughout the company who were less directly involved.

During the course of the program these reviews allowed the development of close professional and personal relationships that clearly helped us to meet the VAX 8600 program performance, function, cost, quality, volume and schedule goals.

The following papers represent a cross section of the problems addressed, solutions found, and successes achieved in the course of developing the 8600. Many topics could have been included, but this group should provide the reader with some insight into the product design and management processes associated with this program.

At this time I would like to acknowledge formally all the people not only within the High Performance Systems and Clusters Group, but especially those outside this group who contributed so creatively and generously to this program. Naming each of them would surely consume the remaining pages of this journal since there were over 40 different facilities, and at least that many groups, involved in the VENUS program. The success of the VAX 8600 system is their success!

Tryggve Fossum
James B. McElroy
William English

An Overview of the VAX 8600 System

The VAX 8600 system handles 5 million Whetstones per second, which is over four times faster than the VAX-11/780 system. The 8600 uses pipelined instructions, a bigger cache memory, and a dedicated memory bus to achieve its speed. Inside, small processors-called boxes-perform tasks simultaneously. The I Box prefetches instructions while the E Box executes others; the F Box performs fast floating point operations, as do all VAX systems. Macrocell array technology, with fast gate speeds, and microcode control are used throughout. These aspects, plus a new cooling system and interconnect innovations, make the VAX 8600 system very reliable.

The main design objective of the VAX 8600 project was to gain a significant improvement in VAX computing performance with a minimal cost increase. Furthermore, the 8600 had to retain all the characteristics common to the 32-bit VAX Family. These characteristics included the following requirements: the new machine must run the VMS operating system, must interconnect to the present I/O bus structures, and must have the network links associated with the VAX computing environment. Improved performance is achieved through innovations in computer design and the introduction of large scale processing concepts into the VAX architecture. Innovations include the use of ECL macrocell arrays (MCAs) throughout the CPU and new electrical and mechanical packaging. Among the large scale processing concepts employed are a dedicated memory bus and pipelined operation in both instruction processing and memory references.

Designing a large scale computer is a process driven by Digital's performance goals for the machine. On some projects, little time remains to evaluate the relative costs of equivalent alternatives. All VAX systems, however, must meet price/performance design criteria, the most important of which is the customer's overall cost of ownership. Therefore, to meet those criteria, we used many techniques to enhance the system's reliability, availability and maintainability.

The VAX 8600 System

The VAX 8600 processor (Figure 1) consists of six relatively independent subprocessors: E Box, F Box, I Box, M Box, console, and I/O adapter. The E Box executes the VAX instruction set and generally directs the entire system. The I Box prefetches instructions and operands and decodes them for later execution by the E Box. This gives the machine a pipelined structure: several instructions can be present in the I Box and the E Box at the same time. The pipeline enables some frequently executed instructions to be completed in the E Box in a single machine cycle of 80 nanoseconds.

The M Box contains a 16-kilobyte data cache to increase the speed of memory access. It also contains a buffer that holds recently used translations of virtual memory addresses to physical ones. Using a translation buffer eliminates the need to look up these addresses for every memory access. The M Box interfaces the memory to all other parts of the system, and also interfaces the E, F, and I Boxes to the adapter bus for input and output. A "memory reference" by one of the other boxes happens in a "cache cycle," the objective of the design being to deal solely with the high-speed cache as often as

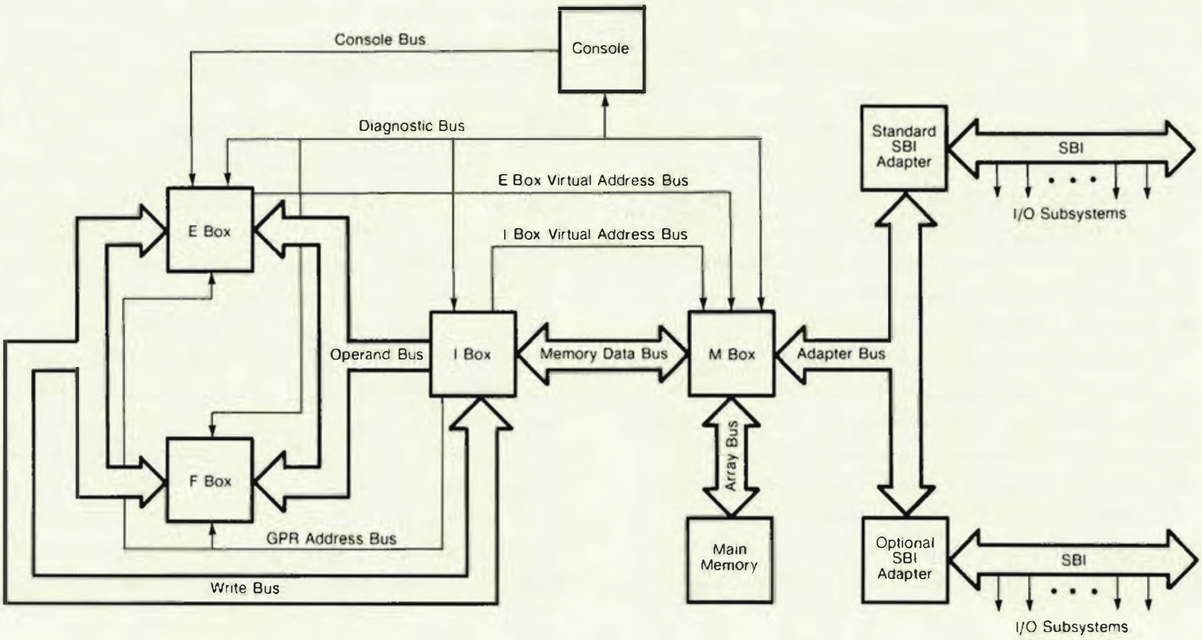


Figure 1 The VAX 8600 Operations Flow

possible. The M Box actually references storage only when needed data is not found in the cache or when room needs to be made in it for new data. As elsewhere in the machine, the M Box has a variety of reliability and maintainability features, including error correction on the data in the cache.

The F Box is a floating point processor or accelerator (FPA). When present in a system, the F Box intercepts floating point instructions as they are presented by the I Box. Special hardware for fast unpacking, aligning, adding, multiplying and dividing produces extra-high performance for scientific, computational number crunching.

The 16 general purpose registers (GPRs)–the I and F Boxes each have one set and the E Box, two–are basic to the accurate and fast manipulation of data. Therefore, altogether, four copies of the GPRs are kept to guarantee fast, flexible access and instruction retry.

The console is a microprocessor-based front end interface to the operator, the boot device, and the remote diagnostics. This unit is used to initialize the system on power-up, to test it, and to assist in isolating faults. The console also automatically handles various functions that are usually performed manually by an operator.

The I/O system is based on Digital's standard synchronous backplane interconnect (SBI), which is interfaced to the rest of the system via the M Box through an adapter on the adapter bus. The various device controllers and adapters to other interconnects are located on the SBI.

Although all boxes contain microcode, the main microcode is in the E Box. This allows the boxes to perform complex functions with a small amount of hardware, providing design flexibility and a good cost/performance ratio. All microcode storage is writable, which facilitates changes and additions whenever necessary. Initially, the RAMs are loaded from microcode files stored on a removable disk in the console subsystem. Microcoded diagnostic programs are also loaded in the control store when it is necessary to identify failing components.

A number of buses interconnect the various boxes. All data movement between the processor and both the memory array and the I/O subsystem occurs through the memory data bus connecting the M Box and the I Box. The I Box receives the instruction stream and the memory operands over this bus; the memory operands are then passed to the E Box and the F Box over the operand bus. Results from either of those boxes are sent via the write bus to the I Box, which in turn passes them to the M Box over the memory data bus. The write bus is also used to keep the four sets of GPRs identical to one another. Both the I Box and the E Box supply addresses (almost always virtual) to the M Box. All buses and registers handle 32-bit words.

The component technology used in the 8600 is the macrocell array, which provides a typical gate speed of one nanosecond and has high-density LSI ECL technology in a 68-pin package that is one inch square. MCA technology is an extension of the gate array concept. Instead of gates, however, each cell in the array contains a number of unconnected transistors and resistors. By creating interconnecting patterns, one can transform those components into small-scale/medium-scale integration (SSI/MSI) logic functions or "macros." These macros take the form of standard logic elements such as dual D-type flipflops, dual full adders, quad latches, and the like. Most of them are series-gated ECL structures for optimized performance.

E Box, Heart of the System

The E Box, the focal point of the entire system, executes the VAX instruction set, handles exceptions and interrupts, and controls the rest of the system. It is highly microcoded: most of its elements are directly controlled in each cycle by bits in the microword. Intensive microcoding makes possible the use of a datapath with a simple structure; the power of the datapath comes from the speed and ease with which it can be manipulated by the microcode.

As shown in Figure 2, the E Box contains a dual-ported scratchpad memory (Register Files A and B) comprising 256 32-bit registers. In the

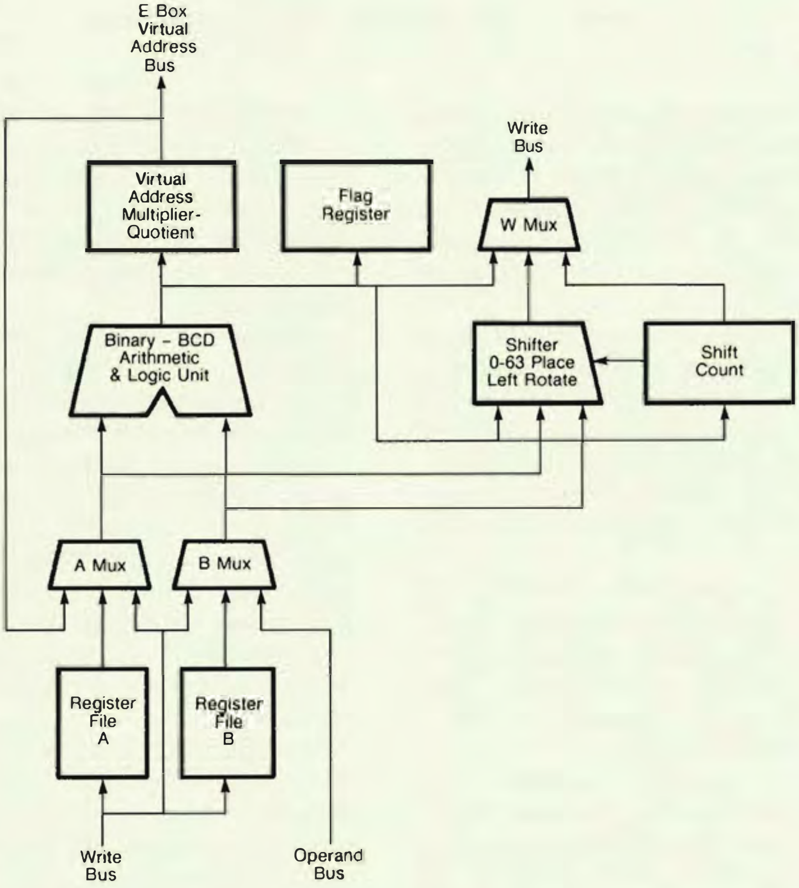

E Box Virtual Address Bus
Write Bus
Virtual Address Multiplier-Quotient
Flag Register
W Mux
Binary - BCD Arithmetic & Logic Unit
Shifter 0-63 Place Left Rotate
Shift Count
A Mux
B Mux
Register File A
Register File B
Write Bus
Operand Bus

Figure 2 Block Diagram of the E Box

scratchpad are basic machine registers, copies of the GPRs, about 150 constants and microcode temporaries, and some architecturally defined registers used by memory management and the operating system.

Arithmetic and logical operations are done by a 32-bit arithmetic and logic unit (ALU), which has all the usual functions for performing add, subtract, OR, exclusive OR, and similar operations. There are also some special ALU functions for speeding division, decimal arithmetic, and comparisons. The most significant performance factor related to the ALU, however, is the ability of the microcode to take any two values from the scratchpad, operate on them in the ALU, and store the result back in the scratchpad–all in a single cycle. With this capability, some whole instructions can be completed in just one cycle. And longer, repetitive instructions, such as those handling character strings, can be executed in short loops.

Paralleling the ALU is a barrel-shift network that accepts a 64-bit value, joins it end to end, and selects any desired 32 consecutive bits from the ring format. The value can be supplied by two scratchpad registers or one register concatenated with memory data. Control over the shifter can be exercised directly by a field in the microword, or through a shift control register. The register allows a new shift count related to some previously specified one. The shifter is used for unpacking and packing floating point data, translating different decimal data formats, arithmetic shifts and rotations, and various other bit manipulations. As in the case of the ALU, the shifter's power is enhanced by the ability of the microcode to take any two words in the scratchpad, shift them, and store the result back in the scratchpad, all within the same cycle.

I Box Handles the Details

The VAX architecture has a rich instruction set with a large number of opcodes and specifiers for fetching operands and storing results. While this variety is quite useful to the programmer and compiler writer, the task of decoding these opcodes and specifiers constitutes much of the total work in processing VAX instructions. Therefore, the 8600 has a separate subsystem dedicated to prefetching instructions, decoding them, fetching source operands, and storing results. That subsystem also receives condition codes from the E Box and makes all branch target fetches and decisions. Much of the time, this work is overlapped with the actual instruction execution in the E and F Boxes, thus achieving a high degree of simultaneous processing.

The I Box consists of two major parts: an instruction unit and an operand unit (Figure 3). The instruction unit contains an 8-byte FIFO instruction buffer, which receives instruction-stream data from memory, 4 bytes at a time. The unit evaluates these bytes to determine the addressing mode and to make instruction optimization decisions. Evaluation is done with the help of a decode RAM, which contains information specific to the individual opcodes and specifiers.

The instruction unit also supplies information about where to find the operands for an instruction. Using this information, the operand unit can generate the addresses for the operands and start the memory reads to fetch them. For this purpose, the unit has its own copy of the GPRs, since they are needed to calculate the addresses. Often the GPRs contain the operands, in which case either they are read directly or the numbers of the GPRs containing them are passed to the execution units (E and F Boxes). At other times, the operands are contained in the instruction stream itself, in which case they are extracted from the instruction buffer. Whenever possible, the instruction unit tries to process two specifiers in a single cycle by handling the second specifier as a GPR number. This *optimization* saves valuable cycles in frequently used instructions.

When the E Box is ready, the I Box supplies the operands to it along with a dispatch address identifying the start of the microcode appropriate to the execution of the instruction. When execution is complete, the operand unit will provide the address for storing the result in memory.

Therefore, the overall sequence of steps in performing an instruction is fetch instruction, decode instruction, generate address, fetch operand, execute, and store result. Any one of these steps for a given instruction may occur simultaneously with any other step for some other instruction. Of course, this is limited by the obvious restriction that no two operations can use the same resource (memory, register file, etc.) simultaneously. Thus, for example,

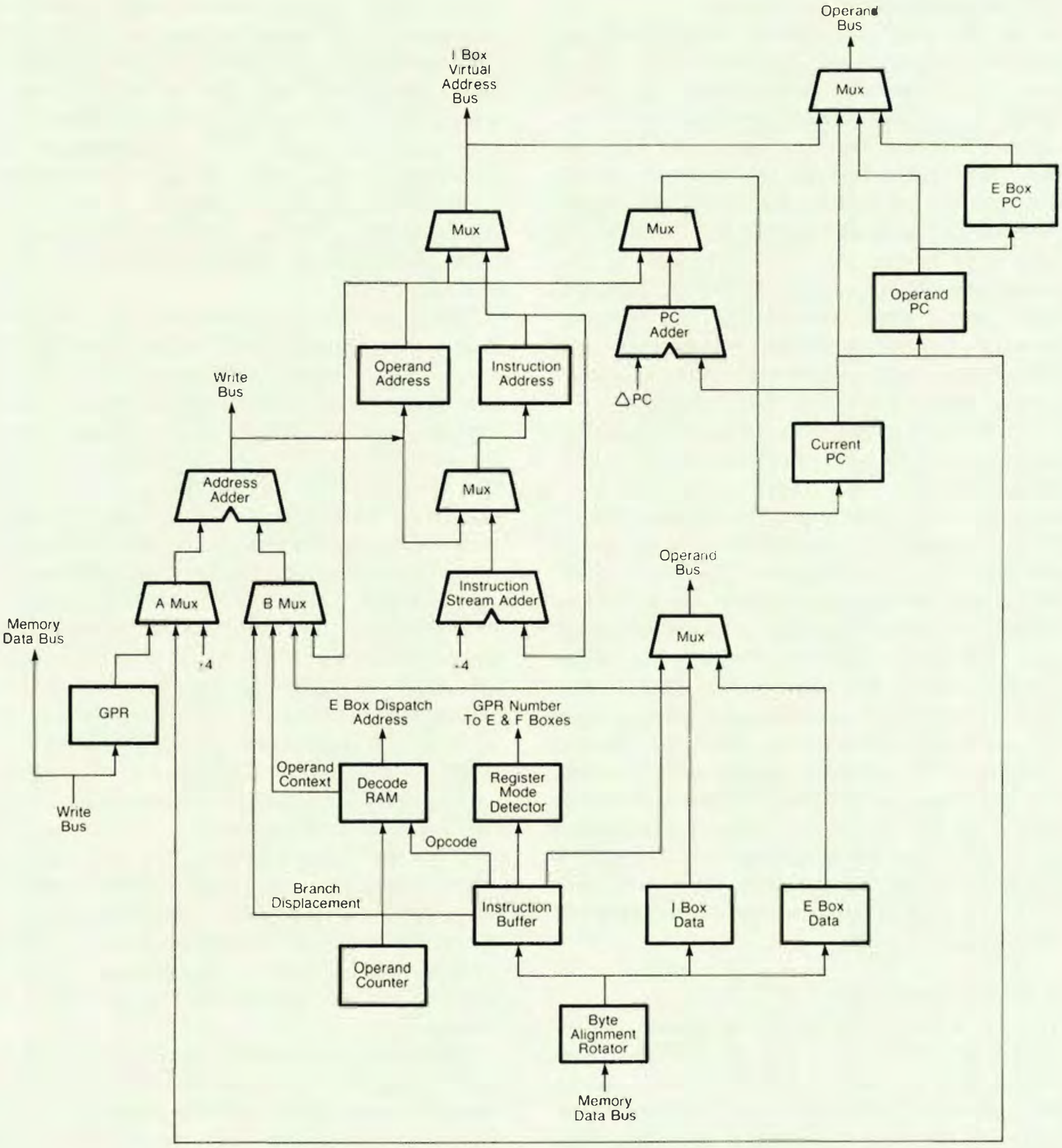


Figure 3 Block Diagram of the I Box

while the I Box is decoding instruction 4, it may also be calculating addresses for instruction 3 and fetching the operands for instruction 2. If the operands are in the GPRs, then the current cache cycle may be used for fetching more of the instruction stream (say parts of instructions 6 and 7, with 5 already in the buffer). Moreover, any of these steps may be happening while the E Box is executing instruction 1. This overlapped processing, called "pipelining," greatly improves performance and is detailed later in the Pipelined Instruction Processing section.

Of course, there are bound to be hazards whenever work is done in parallel. The pipeline cannot always operate at full speed due to conflicts produced by the various subsystems needing the same resources. Since several stages may be active simultaneously, the control of each stage is intimately tied to the past and present operations in the other stages, as well as to those in the E Box and the M Box. Each stage attempts to process the available input data as quickly as possible. Whenever input is unavailable or a result cannot be stored immediately, a stage is said to be "stalled." One objective of the I Box, and of a pipelined structure in general, is to minimize the time any stage spends in a stalled state as it can perform no useful work during that time. The execution unit will sometimes store a result in a register that is needed by the operand unit for the next instruction. A problem of this sort is resolved by using scoreboards and conflict detectors. In many cases, conflicts are avoided by passing the data as GPR tags, rather than passing the actual data. Fortunately, the VAX architecture normally precludes writing into the instruction stream, so the instruction buffer can prefetch freely across most instructions.

When appropriate, the I Box supplies all operands sign-extended and all floating point operands in memory format, independently of the source of the data. Therefore, the E Box and the F Box do not need to perform any special data manipulations before the data is used. In keeping with the principle of a high-speed, yet economical implementation, the VAX 8600 system uses the instruction buffer to fetch data for string and other multiple-operand instructions, thus using hardware that would otherwise sit idle. This procedure expedites large amounts of data through the processor without wasting cache cycles. This feature is especially important in commercial applications where data manipulation is more important than arithmetic speed.

Since the 8600 is designed to run with the VMS operating system, the processor must be prepared to deal with memory exceptions during instruction execution. This procedure is complicated by multiple instructions being in the pipeline at the same time. For sorting things out, the operand unit has multiple program registers that contain the starting addresses of all instructions in progress. A register log keeps track of GPR changes that must be undone should an instruction have to be repeated.

M Box and Memory

The memory system includes the storage array boards and the M Box. This box contains not only all of the control, transfer, and error logic for the storage array, but also a data cache for fast access to memory data (Figure 4). Each array board contains 4 megabytes of MOS storage, and the memory backplane can hold eight boards for a maximum of 32 megabytes. The basic storage unit is a block of four 39-bit words, each with 4 data bytes and a 7-bit error-correction code. Special logic is included for writing bytes, significantly decreasing the storage access requirements. The M Box interfaces to and handles communication among the three major parts of the system: the main memory, the processor, and the I/O system (via the adapter bus).

The cache is a high-speed memory with locations that act as temporary substitutes for a selection of the most frequently used storage locations. The cache is two-way associative, meaning that for each address, the data can be stored in either of two locations. The total cache size is 16KB in two 8KB parts; its locations are allocated in blocks of four words (16 bytes), addressed on a four-word boundary. In addition to the two data parts, there is a cache tag store containing the address bits for the blocks of data in the cache data store. For each block, the tag store also contains a valid bit and four written bits for the four words in the block. Associated with the data to ensure its integrity is an error code that enables the correction of single-bit errors and the detection of double errors.

The cache uses a writeback scheme for writing in memory. This means that a word is not written in storage when it is modified, but only

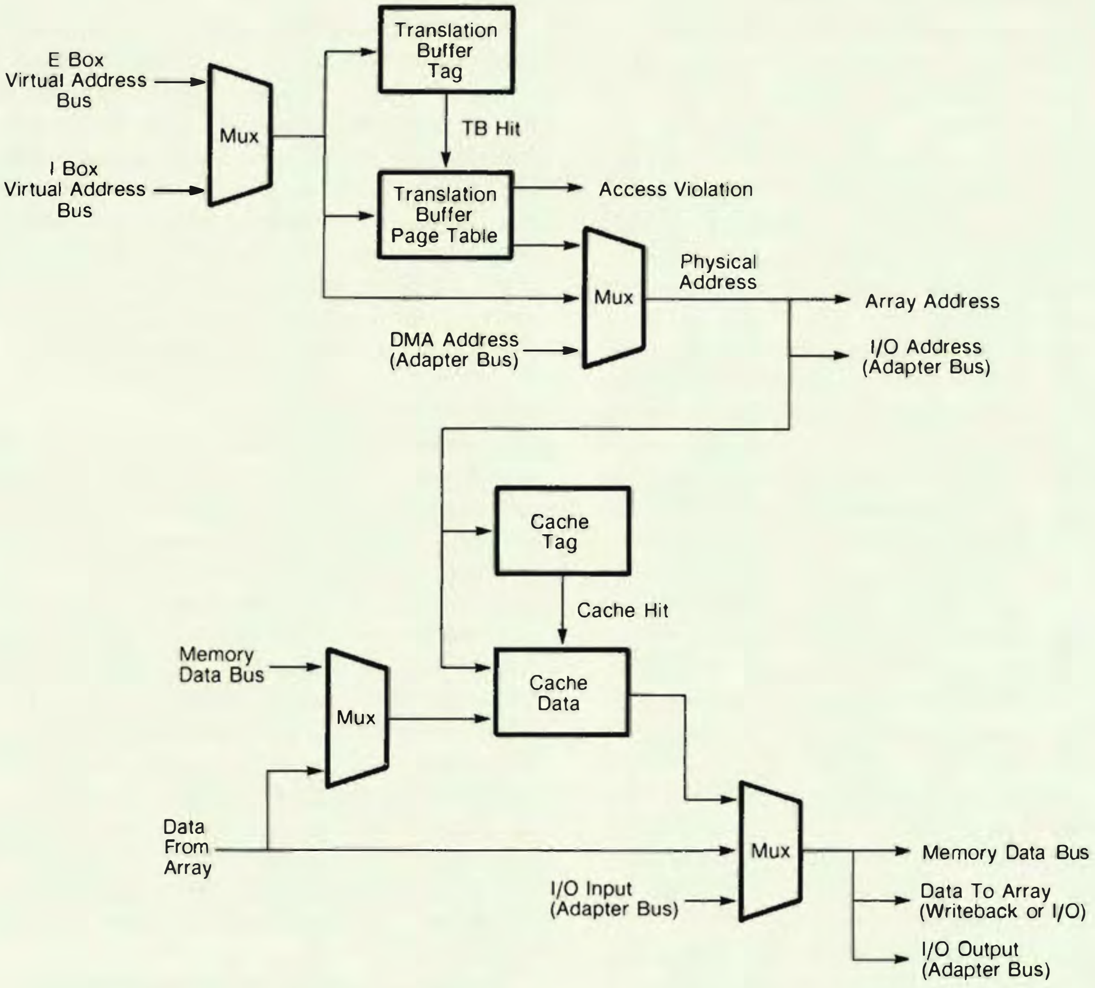


Figure 4 Block Diagram of the M Box

when its cache location is needed for other data. In the interim, data is placed only in the cache, so a single cache location may be used many times without requiring access to the memory array (whose corresponding location becomes invalid). The contents of the cache are finally written in the array only when that cache location is needed to represent a different storage location. The replacement policy is "least recently used." That is, of the two blocks available to store a given piece of data, the one less recently accessed receives the new data. When a memory word containing a corrected error is placed in the cache, the written bit is turned on to force eventual rewrite of the storage location, thus reducing the probability of a double error.

Addresses actually supplied to the cache or the memory array are always physical, and the direct memory access (DMA) references made by the I/O system always use physical addresses. There are three sources of memory references within the processor, each having its own port into memory: the instruction buffer, the operand unit, and the execution unit. Normally these references are virtual, meaning that the addresses have to be translated from virtual to physical before they can be used to access the cache. When a virtual reference is made, the M Box microcode uses the high-order part of the address to index into the translation buffer (TB), itself a cache containing the most recently used translations. The entry from this buffer is then prefixed to the remaining bits of the virtual address to form the desired physical address. The TB is one-way associative and has a capacity of 512 paging entries. Besides translation information, it contains access-protection data, which aids in creating a secure operating environment. Refilling entries in the buffer is done from page tables in memory.

Although the TB is located in the M Box, it is maintained by microcode running in the E Box. This provides an economical solution to the complicated task of keeping track of streams of references from the three ports. Each port can have two references in progress, since accessing the data cache and the tag store are overlapped with accessing the TB. The data, addresses, and control information for these operations are carefully queued, with handshakes to allow the subsystems to proceed as far as possible (but not any further) while waiting for references to finish. Any memory exceptions encountered while prefetching instructions or operands are held off until the data is actually needed by the execution unit. That unit then deals with the problem, using memory references that bypass the normal queue, thus leaving it intact for restarting later.

The result is a virtual memory system that is fast enough to allow a reference to complete during every cycle. With three subsystems making independent references, the high bandwidth of the bus, which allows that speed, can be well utilized.

F Box Performs Floating Point

For scientific and technical applications, the 8600 has a floating point accelerator (FPA), the F Box, that operates in parallel with the E Box. The FPA receives operands over the operand bus from the I Box and delivers results over the write bus for storage in GPRs and memory (Figure 5). It performs floating point calculations in all four VAX floating point formats, F, D, G and H (F numbers have 32 bits, D and G have 64, H has 128), and it also does integer multiplications. Usually the work involved in these calculations is split between the F Box and the E Box. The former does the arithmetic operations while the latter accesses memory for reading and writing operands, deals with exceptions, handles counters, and takes care of other chores.

The E Box has a fairly general-purpose datapath, capable of dealing with the myriad tasks involved in executing the VAX instruction set. On the other hand, the F Box consists of specialized hardware (almost exclusively gate arrays) for doing only those steps needed in floating point operations. Hence, these operations are executed in far fewer cycles. Furthermore, the F Box cycles twice as fast as the other subsystems; its datapath is 32 bits, and multiprecision operations are pipelined. The F Box also has its own copy of the GPRs, allowing the I Box to send both operands at the same time, one over the operand bus and one as an address for the GPR RAM.

Much of the original challenge in F Box design lay in making it compact so as to minimize interconnect delays. Of its two modules, one contains the logic for floating point addition, subtraction and division, while the other does floating point and integer multiplications. Both modules are microprogrammed, with each having its own microsequencer and control store. Moreover, the microcode is distributed among the various chips. This distribution enables a command to follow the data for several cycles and be repeatedly decoded as the floating point operation is executed. That allows normal operations to finish in a minimum number of cycles, while unusual conditions are detected and dealt with by the microcode.

The multiplier module uses column reduction and Booth encoding, together with a 3-input adder, to produce a 40-bit partial product every half-cycle. The adder combines the operations of unpacking and aligning in a single shift, making it possible to produce an F format sum in only two cycles. Thus, ADDF2 takes just two cycles (as opposed to four in the 11/780), MULF2 takes four cycles, and each add-multiply step in a POLYF polynomial evaluation takes only six cycles.

The VAX 8600 system continues a tradition of providing high-speed, accurate floating point performance. All operations are accurate to one half of the least significant bit. Any floating point exceptions cause the instruction to back up to its beginning. Then control is given to an exception handler, which scales the operands before resuming computation. By having all four formats available, intermediate calculations can be done in a format with greater range and precision, thus avoiding exceptions and returning a more accurate result in composite operations.

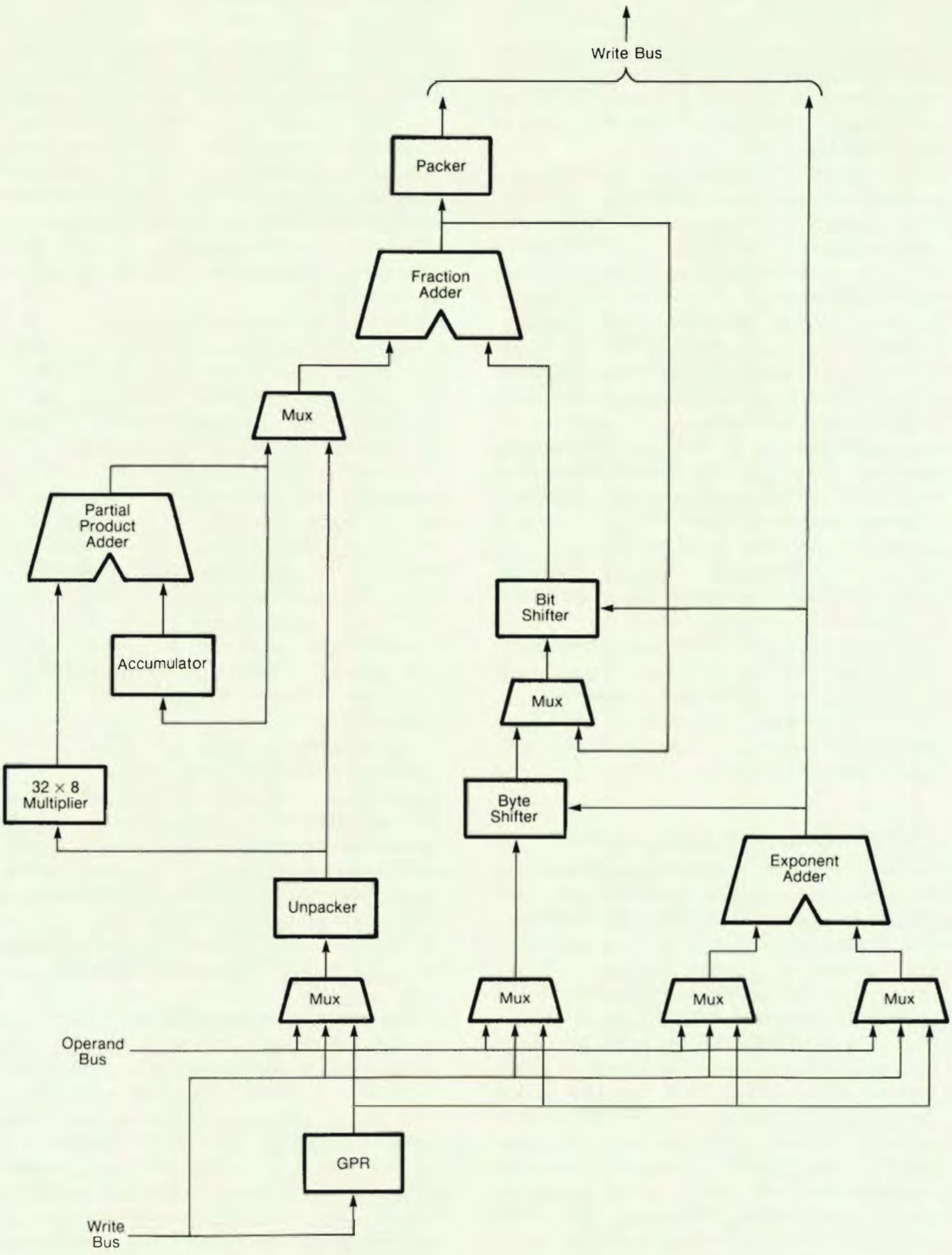


Figure 5 Block Diagram of the F Box

Besides the basic operations of add, subtract, multiply and divide, the 8600 provides special instructions for argument reduction and polynomial evaluation. These instructions carry extra precision and also facilitate the high-speed software implementation of transcendental and other sophisticated mathematical functions.

System Microcode

In addition to controlling the E Box datapath, the E Box microcode supervises the operation of the whole processor. Microcode initializes the system and tells the instruction buffer when to prefetch instructions or string data. Furthermore, it starts and stops operand processing in the operand unit, maintains the address translations in the TB, and orders the F Box to perform arithmetic operations. The microcode executes the full VAX instruction set, including recent additions such as G and H floating point, and interlocked queue instructions for multiprocessing. Since it is backward compatible, the microcode also executes the PDP-11 instruction set.

Considerable effort was expended on optimizing the microcode and the E Box datapath to execute the VAX instruction set. The result is a relatively narrow microword of 84 bits (including two for parity), which nonetheless allows most high-frequency instructions to complete in a single E Box cycle. Having immediate access to all 256 scratchpad locations makes it possible to store decimal strings and other data structures internally, saving crucial instruction cycles. Low-frequency operations are implemented principally in microcode rather than in hardware to save board space and reduce cost.

The E Box microcode is written in a straightforward language that is easy to write, understand and debug. Of the 8K control store locations, 7K are used for the system microcode. The remaining 1K are available to the customer for implementing special functions, and "hooks" are provided for fast and easy access to user microcode.

All subsystems have microcode; however, compared to the E Box, they all contain more specialized hardware and microcode to perform fewer, but more specialized tasks. Even so, microcode still provides an economical, flexible alternative to hardware as a means to implement control. Wherever practical, normal, high-frequency operations are done in hardware, whereas unusual operations are handled in microcode.

Much of the error reporting and recovery is also implemented in microcode. If an error related to the currently executing instruction occurs, the microcode is trapped. It then collects the error information, fixes the error condition, backs up the affected instruction for later restart, and enters the machine-check software.

Console

The console, connected to all four of the boxes by a serial diagnostic bus, is actually an extensive subsystem based on a PDP-11 computer. The console monitors environmental and power-supply conditions, serves as the VMS operating system terminal, supplies a time-of-year clock, and provides an assortment of diagnostic functions. Associated with the console are a local LA100 terminal for use by the operator, an RL02 removable disk for bootstrapping and diagnostic activities, and a remote diagnostic link. Bootstrapping is done automatically by the console, which serially passes microcode and initializing information to the various boxes over the diagnostic bus. The console and the E Box communicate via the console bus (C bus) to set up the I/O system and to implement console functions such as examine, deposit, start, and halt.

Input/Output System

The I/O system provides input/output over a synchronous backplane interconnect (SBI) interfaced to the M Box via the adapter bus. This system offers complete compatibility with the myriad peripheral equipment currently available for the VAX-11/780 Family of machines. Moreover, the 8600 can have two SBIs, and its separate memory bus relieves them of any involvement in processor-memory transfers. Therefore, a significant increase in both the computational capacity and the I/O throughput of an existing VAX system can be gained simply by replacing only its processor with an 8600 and leaving the entire peripheral system in place. A single SBI can handle 13.3 megabytes of data per second, all for input/output; two SBIs have a combined capacity of 17.1 megabytes. Some I/O device adapters connect directly to the SBI; others must connect through a UNIBUS or MASSBUS. The theoretical maximum capacity of the adapter bus is 33.3 megabytes using two ultra high-speed adapters with transfers in 16-byte blocks.

The latest I/O equipment is designed to be used with the computer interconnect (CI), which has a bandwidth of 70 megabits per second, and the Ethernet, which has a bandwidth of 10 megabits per second. The 8600 is the first VAX system to include the CI interface signals in its own backplane, providing as standard equipment the hardware necessary for its inclusion in a VAXcluster. The VAXcluster is a loosely coupled, multiprocessing environment of 16 nodes. Any node in the cluster can be either any member of the VAX Family, including another 8600, or an HSC-50 mass-storage controller. The HSC-50 controller provides intelligent, high-speed and shareable access to both disks and tapes for all the CPUs in the cluster; the maximum sustained data rate is 3.4 megabytes per second. Each HSC-50 controller handles six data channels, and each channel can access four datapaths for either disks or tapes.

The Ethernet can handle 1,024 stations with a maximum separation of 2,500 meters in a branching, unrooted tree. It is used in local area networks for communications between computers (such as DECnet service), unit-record equipment, workstations and the like.

Performance Improvements

The improved ability of the 8600 to execute a specific instruction, as compared with the 11/780, can be determined by comparing the following factors: the shortening of the cycle time, the decrease in the number of cycles required, and the decrease in memory access time. Since the 8600 overlaps instructions, simply comparing the speed of individual instructions does not give a true indication of the ability of the new VAX processor to perform an actual task. Because of the operational sequences chosen, even benchmarks often fail to give a complete picture of the improvement. This is true because the 8600 improves the speed of handling interrupt and exception functions even more than the speed of instruction operations. And, of course, other quantities such as memory size and disk capacity also affect the comparative performance.

In designing the VAX 8600 system, the basic performance objective was to increase the average instruction execution speed by a factor of four. This objective was not only met but exceeded. The most significant features contributing to this performance improvement are the following:

- The pipelined machine organization reduces by 40 percent the average number of machine cycles required per instruction. The I Box prefetches instructions and operands while the E Box is processing the current instruction. The address and data functions used to reference memory are also pipelined.
- The VAX 8600 cycle time is 40 percent of that of the 11/780 (80 versus 200 nanoseconds) and 60 percent of that of the 11/785 (80 versus 135 nanoseconds).
- Faster and larger RAMs in the E Box allow the microcode to accomplish more processing in a single cycle.
- The cache uses a writeback strategy that eliminates unnecessary writes to memory.
- The two-way associative cache is twice the size of the cache in the VAX-11/780 CPU (16KB versus 8KB).
- A dedicated memory bus with separate address and data lines eliminates contentions between memory references and I/O traffic, and between address and data transfers.
- Faster semiconductor technology decreases the gate delays for the 8600, as compared with the 11/780. Gate delays are 1 and 3 nanoseconds, respectively.

Pipelined Instruction Processing

The solid boxes on the diagonal in Figure 6 show the successive actions the processor takes to perform most instructions; that is, those that involve a single operation carried out on one pair of operands, represented in the instruction by the opcode and two operand specifiers. In small, low-speed computers, there is no pipelining. The processing, from fetching the instruction to storing the result, is performed for one instruction at a time. For example, the fetch of the next instruction does not occur until the result of the current instruction has been stored. The hardware devoted to each specific activity is used only during that corresponding step and then remains idle until needed for the next instruction.

Larger computers, like the VAX-11/780 system, shorten their execution times by prefetching instructions: whenever a cache cycle is

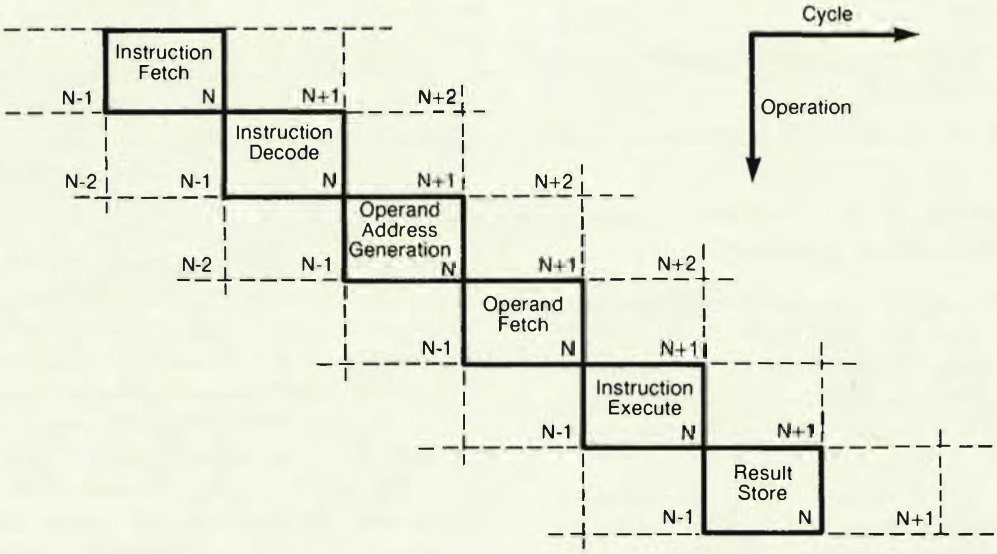


Figure 6 The VAX 8600 Instruction Pipeline

available, the instruction box continues to prefetch more of the instruction byte stream from memory while activities for the previous instructions are proceeding. Thus, the next opcode is ready for decoding as soon as a result is stored. This simple level of pipelining decreases the total time required for getting the instructions.

The 8600 carries the pipelining technique much further by pipelining the entire sequence of instruction activities shown in Figure 6. As indicated by the dashed boxes above and below the solid ones, the processor circuits for each type of activity are normally busy processing successive instructions. Of course, movement through the pipeline cannot always be at top speed. Various stages must sit idle whenever a cache miss requires waiting for data from main memory, or when a multiplication or division ties up the E Box for a whole string of cycles. Even the common instructions that take one cycle to execute still require a total of six cycles to complete (480 nanoseconds); a string of such instructions, however, can store a result in a register location during every cycle (80 nanoseconds).

As an example, consider the instruction ADDL2 (R0),R1, which uses two source operands and stores the result in the location of the second. This involves the steps in the I, E and M Boxes outlined in the following steps.

1. The I Box fetches ADDL2 from the instruction stream in memory.

2. The I Box uses the opcode from ADDL2 to address the decode RAM.

3. The I Box gets the virtual address of the first operand from register R0 and sends it to the M Box.

4. The M Box translates the virtual address into a physical address, retrieves the data from the cache, and sends it to the I Box. (If the cache does not have the data, the procedure must wait at this stage for the M Box to get the data from storage.)

5. The E Box receives operands from the cache and R1, and adds them.

6. The E Box stores the result in R1. (If the result were to be stored in memory, the I Box would supply the address.)

Reduced Memory Access Time

Those factors that contribute most to reducing the memory access time in the 8600 are the dedicated memory bus, pipelined references, and greater cache hit rate.

The dedicated memory bus has decreased the access time to the memory array by more than two thirds–the extra time taken for a cache miss is typically 500 nanoseconds, as opposed to

1600 for the 11/780. This happens for the following reasons:

- The bus itself is faster than the SBI (80 versus 200 nanoseconds).
- There is no interference between memory and I/O traffic.
- Addresses and data are transferred simultaneously rather than in sequence.

All memory operations–addressing, data read, and data write–are pipelined in the 8600. Latency is still at least two cycles, one each for address generation and cache lookup, but a cache reference can be completed during every cycle.

Finally, the cache hit rate of the 8600 has been improved simply by making its cache twice the size of the one used in the 11/780. Some time has also been saved by using the writeback strategy as compared with the write-through strategy of the 11/780. In write-through, both the cache and the memory array are updated on every memory write.

Technology Contributions to Improved Performance

The processor cycle time has been reduced mainly by (a) using a faster semiconductor technology; (b) decreasing the wire length on both modules and backplanes; (c) using faster RAMs for the registers, cache, control storage, and memory array.

The semiconductor technology in the VAX 8600 processor is emitter-coupled logic (ECL). This logic is nonsaturating; it is, therefore, much faster than the VAX-11/780 transistor-transistor logic (TTL), in which state changes require either full charge or full discharge. The logic design takes advantage of the very fast ECL state changes because the effects of signal reflections were greatly reduced by minimizing interconnect delays, and wiring impedances were carefully controlled. ECL-TTL conversion is needed to interface to the SBI, the console, and the memory array (which uses 256K TTL-compatible MOS RAMs). The conversion is handled by dual-ported RAMs that serve as converting buffers; data goes in in one form and comes out in the other.

Instead of the flipflops employed in other VAX systems, the VAX 8600 system mainly uses latches in its registers and control logic. Performance is improved because latches are level sensitive, whereas flipflops can change their states only when clocked. In other words, no matter how quickly the inputs to a flipflop are set up, a new output configuration cannot be sent along to the next logic stage until the next clock. With a latch, however, the outputs can change when the inputs change, allowing a faster setup at the next stage. Despite the requirements for holding gating levels for some minimum time, this characteristic of latches is responsible for a reduction of about 10 nanoseconds in the cycle time. Usually, more latches than flipflops are needed to implement a given logic function; latches, however, cost less than flipflops, so the cost per logic function using either type of circuit is almost equal. Hence, the only real cost when using latches is the greater difficulty in performing timing analyses. Given the significantly increased performance, this cost is well worth it.

Macrocell Arrays

Until now, the semiconductor industry has used three approaches to meet the demand for LSI digital circuits: standard, off-the-shelf circuit families; custom circuits; and gate arrays. Standard circuits are economical but insufficient for the complex, specialized functions required by the 8600. Custom circuits, on the other hand, are quite expensive and take one to two years to design and produce. Fortunately, gate arrays have a shorter production time, since the basic array can be fabricated up to the point of metalization; unfortunately, the interconnecting metal makes the chip larger and increases the propagation delays. To circumvent these problems, Digital and Motorola created the so-called "macrocell array" approach to custom LSI. This approach decreases the cost and time to develop custom circuits and avoids many of the deficiencies of conventional gate arrays. Among the various technologies evaluated, the macrocell array best met the requirements of the 8600.

As explained at the beginning of this article, the macrocell array is actually an extension of the gate array concept. Each cell in the array contains a number of unconnected transistors and resistors that can be connected to form specific logic functions or "macros." The cell library contains 85 macros: 54 for major cells, 14 for interface, or input, cells; and 17 for output cells. A single array can contain 106 cells: 48 major, 32 interface, and 26 output. If full adders and latches are used in all cells, a single MCA may contain 1,192 equivalent gates; if flipflops and latches are used in all cells, it may contain 904. Typical power dissipation is 5.0 watts, 4.4 milliwatts per equivalent gate. Contributing to the high performance of the system as a whole is the extremely low propagation delay in major and interface cells: 1.2–1.8 nanoseconds maximum, compared to 3.5–6.0 nanoseconds for 10K ECL. The high density of 100 gate equivalents per square inch, compared to 20–30 for MSI, is also important. Higher density reduces interconnect delays, thus further enhancing performance, and lowers packaging costs as well.

Reliability and Data Integrity

Although we have not been able to eliminate hardware errors entirely, the VAX 8600 system goes a long way toward eliminating their effects on the user. Features are built into the 8600 at every level to guarantee the integrity of the data in the system and to promote its reliability, availability, and maintainability. These features range from minor characteristics within individual circuits to major provisions that embrace the entire system. Some of the more significant features are listed below.

- Inherent reliability is achieved through having a low component count, logic design for the worst-case situation, and high-reliability parts.

- Dynamic error reporting, by means of an error logger, aids in identifying the sources of intermittent failures. The error log is used for both hardware and software malfunctions and is kept in a disk file.

- Instruction retry is used whenever it is appropriate to the error type. For instance, four copies are kept of the general purpose registers. Therefore, on a GPR parity error, the instruction can be retried using a copy from the corresponding GPR in another box.

- Additional related software features include (a) automated patching and updating procedures; (b) powerfail-restart support; (c) user-mode diagnostics; (d) extensive protection facilities; and (e) dynamic memory configuration to exclude bad pages.

- Single-bit error correction and double-bit error detection are used for the cache and the memory array, with automatic rewriting of the corrected word.

- There is parity checking at RAMs and buses, and parity continuity is carried through all major datapaths. Parity is kept not only for data, but also for physical addresses and the microcode. (Bad data in a control RAM or the control store is corrected by the console from its bootstrap files.)

- Address parity and a bad-data flag are "folded" into the error correcting code; thus, the storage words themselves contain information about error sources.

- There are separate selects to each memory array board, so the control logic for storage selection is all in one place, and faults can be isolated to an individual board.

- The memory battery backup has a capacity of ten minutes. The backup time can be set shorter to save on battery recharge time, thus allowing the alternative of riding out multiple short power failures by taking the chance of going down during a long one.

- Continuous self-testing is performed by the FPA when it is not in use.

- The system can be reconfigured without the FPA if floating-point failures are experienced.

- There are fast, accurate diagnostics with first-failure fault isolation to the board. (Subsequent depot-level servicing can isolate to within ten chips, on the average.)

- Signals can be monitored from the console via the diagnostic bus.

- An environmental monitoring module (EMM) gauges the physical operating environment of the system. The EMM measures temperatures and voltages and reports out-of-tolerance conditions to the console, which can shut down the system before permanent damage occurs.

These features make it highly likely that errors will be detected and corrected, thus limiting their impact. If a transient error occurs, the instruction execution will pause and the machine state will be saved in memory for processing by an error-analysis program that provides information to Field Service for quick on-site or remote repair.

The hardware contains the various status flags used by the operating system to determine whether the instruction stream can be restarted following an error or some of the process context has been lost. Since most VAX instructions store results only upon completion, errors, in most cases, cause only intermediate results to be lost; the process can, therefore, be restarted at the specific instruction in which the error occurred. Sometimes an entire process will have to be stopped, although this will not affect the operations of other processes. In the worst case, some errors–infrequent, but overwhelming–may require restarting the entire system. This strategy of graduated error catching and recovering, coupled with a technologically sound, worst-case design, creates a system with very high reliability and availability.

The console is essentially a separate maintenance processor that runs the system for diagnosing and isolating faults. By means of the serial diagnostic bus, the console can scan all signals needed for chip fault-isolation. (These signals are made available through multiplexers contained in the signal-terminator chips.) Also, the console keeps snapshot files of the long-run state of the machine. It has two programs to help system managers to avoid future difficulties. One program monitors the error log to warn of impending problems even if the system is recovering from current situations. The other program displays a graphic image of the system to highlight any faulty components; this is especially useful in a fault-tolerant system, which will not crash to signal a component failure.

Environmental Monitoring Module

Devices for sensing various environmental conditions are located throughout the cabinet. The electronics and indicators associated with these devices are on the environmental monitoring module (EMM), mounted in the power-supply rack. In most cases, out-of-tolerance conditions are reported to the console for appropriate action.

A principal environmental concern is overheating in the logic, since the junction temperature in the MCAs directly affects their failure rate, which doubles with every rise of 20 degrees Celsius. To guard against overheating, precision thermistors monitor the ambient temperature of the incoming air and the temperature gradient across the card cage. By comparing the temperature of the inlet air with that of the air above the cage, the EMM can determine the temperature rise incurred by cooling the system logic. Should the inlet air temperature below the cage reach a preset value, the EMM will issue a warning to the console. If the inlet temperature reaches a danger-zone value or the gradient across the logic exceeds a prescribed amount, the EMM will issue another warning and, one minute later, will shut down system power unless the problem has been alleviated.

Another important function of the EMM is measuring the output voltages of the power supply. Power-supply voltages must be the correct values to ensure reliable system operation. If any of these is out of its operating range, the EMM will report the violation to the console. Voltages are measured continuously so that any out-of-tolerance conditions will be known and can then be reported to Field Service.

Other environmental features include devices for detecting an overheated regulator, a failed blower, and inadequate air flow. Regulator overheating, whether due to faulty operation or excessive ambient temperature, causes the closing of a thermal switch that shuts down the main power control. Unless accompanied by a temperature problem, other, less drastic failures are reported so that the system manager can resolve them.

Besides its monitoring functions, the EMM controls power sequencing, both on and off. The computer has an electronic keying system that detects a board plugged into the wrong slot, and the EMM will not allow logic voltages to go on unless all modules are installed correctly.

Packaging Innovations

We had to make significant changes in the current levels of package design, from the semiconductor devices to the cabinets, to capitalize fully on the new circuit technology. Therefore, we incorporated new techniques in interconnect, packaging and cooling in order to complement the semiconductor technology and to meet new environmental and safety regulations. These efforts were undertaken by Digital's own technology development team with, in many cases, the cooperation of other internal groups and external vendors.

Our efforts to meet the stringent density and electrical requirements at the device level led to the development of LSI packages that serve not only our needs, but also those of others in the industry. By employing extensive computer modeling of the system's thermal characteristics, we designed an integral heat sink that mounts directly on each MCA chip. At many critical locations, ICs are installed in high-reliability sockets that facilitate field repair. This decreases the system's downtime, a fact which helps to minimize the life-cycle cost of the system without jeopardizing its inherent reliability.

Up to six layers of wiring are required to interconnect the devices mounted on a printed circuit board. This wiring is maintained at a controlled (transmission line) impedance to guarantee signal integrity. To ensure uniform cooling of the components, we used wind-tunnel techniques to develop device placement algorithms, and computer analyses of each module design to provide thermal profiles of the integrated circuits. By implementing unique power connectors, rather than using many edge-connector pins in parallel, we gained sufficient signal pins for the density of components on the modules. In addition, the multivoltage bus bar that distributes power on the board also acts as a stiffener to maintain flatness.

Both the modules and the backplanes they plug into are supported and located by a precision, one-piece card cage that also acts as a plenum for the cooling air flow. The backplanes contain 16 layers of printed wiring in a laminated structure. To improve backplane reliability and ease of repair, all connectors are the solderless press-pin type; they utilize compliant pins to ensure long-term electrical contact to the circuit board. Power distribution is handled by large, copper bus bars for the predominant voltages and by the cast backplane frame for ground return. Again, solderless press-pin technology is used to assemble power and ground connectors to the distribution system. Power-supply regulators are located above the logic assembly to facilitate power distribution and to allow a straight, single-path air flow. Along with acoustic treatment, this provides a simple, reliable cooling system that satisfies the latest regulations, including the noise limit recommended for a computer-room.

Special care was taken to design the system's cabling to ensure that, in most cases, cables are not disturbed when any logic or power modules are removed. Furthermore, all external cables interface to an external bulkhead, both to facilitate rapid installation and to meet electromagnetic radiation regulations. Cabinets were redesigned to improve site assembly and to help contain electromagnetic emissions. At the same time, backward compatibility with other VAX systems has been accomplished, so that previously purchased expansion cabinets can still be attached to the processor. Overall, an 8600 with over 16 megabytes of memory is actually smaller than a comparable 11/780, although the new machine does operate with one kilowatt more power.

Mario Troiani
S. Stephen Ching
Nii N. Quaynor
John E. Bloem
Fernando C. Colon Osorio

The VAX 8600 I Box, A Pipelined Implementation of the VAX Architecture

The VAX 8600 CPU has four times the performance of the VAX-11/780 CPU by using high-speed ECL technology and an internal organization with a four-stage pipeline. In this pipeline, up to four simultaneous instructions can be in several stages of execution at any time. At its heart is the instruction and operand fetch unit, the I Box. Under favorable conditions, the I Box can deliver one instruction every 80 nanoseconds to the instruction execution unit, the E Box, yielding a peak execution rate of 12.5 MIPS. Special attention is given to the internal organization of this I Box as it differs from those in previous VAX implementations.

The VAX 8600 computer system is the first pipelined implementation of the VAX architecture.[1] Like its nonpipelined predecessors, the VAX 8600 CPU implements the full VAX instruction set and runs under the VMS and ULTRIX operating systems. In addition, the VAX 8600 CPU provides higher performance and reliability than its predecessor, the VAX-11/780 CPU.

In this context, the performance improvement factor needs to be clearly defined to avoid the confusion that usually arises when discussing performance. First, let us define a given program's improvement factor as the time it takes to execute that program on the VAX-11/780 CPU divided by the time to execute on the VAX 8600 CPU. The VAX 8600 CPU's "true" measure of performance improvement is then the average of such improvement factors over all programs. Since the universe of all programs is too large, one has to select a proper subset of favorite benchmarks for the comparison. This subset of benchmarks can be labelled as the constant unit of work (CUW), and its selection is often the reason for conflicting reports in articles on computer performance. The execution time of this CUW in our model is the product of three quantities: the number of instructions, the average number of cycles per instruction, and the cycle time of the machine under evaluation.

The performance goal of the VAX 8600 project team was to reduce the average number of cycles per instruction from 10 (in the VAX-11/780 CPU) to 6, and also to reduce the cycle time of the machine from 200 nanoseconds (in the VAX-11/780 CPU) to 80 nanoseconds. In order to achieve the goal of reducing the cycle time of the machine, custom ECL gate arrays and standard 10K ECL logic were utilized throughout the design. This technology improved the performance by 2 ½ times. The remaining performance gain of 1 ½ times was achieved by reducing the average number of cycles per instruction through the use of a four-stage pipeline. This pipeline is capable of overlapping the fetching of instruction stream data with the decoding of instructions, the prefetching of operands from memory, and the execution of instructions. In the VAX-11/780 CPU, on the other hand, the stages for the operand address calculation, operand fetch, and operand write are all merged into the execution stage. In the VAX 8600 CPU, up to four simultaneous instructions can be in several stages of execution at any one time.

The remainder of this paper is organized as follows. First, a limited description of the VAX instruction set is presented. Then, an overall description of the VAX 8600 CPU internal organization is provided to familiarize the reader with the general environment of the topic. Here definitions are given of the concepts, mechanisms, and building blocks that will be referenced in the examples of the pipeline model. Further on, an abstract model of pipelines is introduced and a description of the VAX 8600 CPU in terms of such a model is presented. Finally, the details of the internal organization of the instruction unit (I Box) and its associated control structure are presented, including an example of a section of code flowing through the pipeline.

The VAX Instruction Set

The VAX architecture[1] has a rather rich and powerful instruction set. Each instruction, in general, consists of one byte of opcode, optionally followed by one to six operand specifiers. These specifiers can represent the accessing scheme for an operand, the displacement in a branch instruction, or the target address in a call type of instruction. The data type and usage of each specifier is derived from the opcode. There are also two-byte opcodes for multiprecision floating point operations, instruction set extension, and user-defined operations. The instruction set is standardized so that each VAX implementation is able to execute the same software image as well as the same operating system environment. This compatibility is the basic goal for all VAX implementations, including the 8600.

The VAX 8600 Environment

Functionally, the CPU (Figure 1) consists of four separate microcoded units for memory and I/O (M Box), for instruction fetches and preparations (I Box), and for instruction execution (E Box and F Box). The F Box is a coprocessor for high-speed floating point execution. These subsystems and their interconnecting buses are now described.

M Box – The Center of System Communication

The primary purpose of the M Box is to link the main memory, the cache, the CPU ports, and the I/O subsystem. In this capacity, the M Box is the communication center at the system level.

The M Box contains a physical cache for instructions and data and a virtual address translation buffer (TB). It also has exclusive access to the memory array. These resources are accessed by three fixed-priority CPU ports and an I/O port, as shown in Figure 2. The M Box, as the system communication center, must contend with several concurrent activities requiring communication services. To cope with

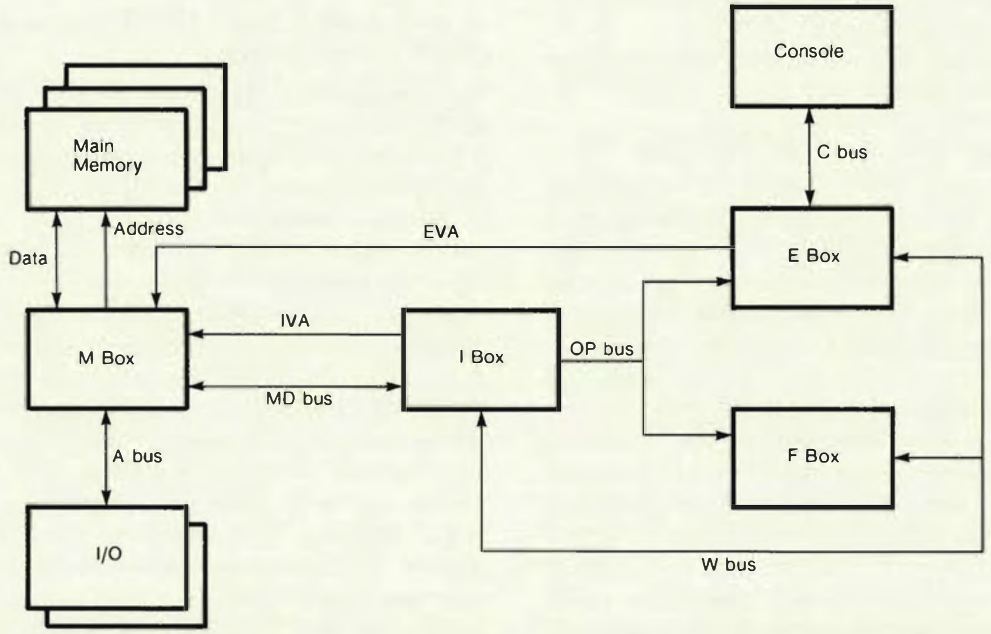


Figure 1 VAX 8600 CPU Organization

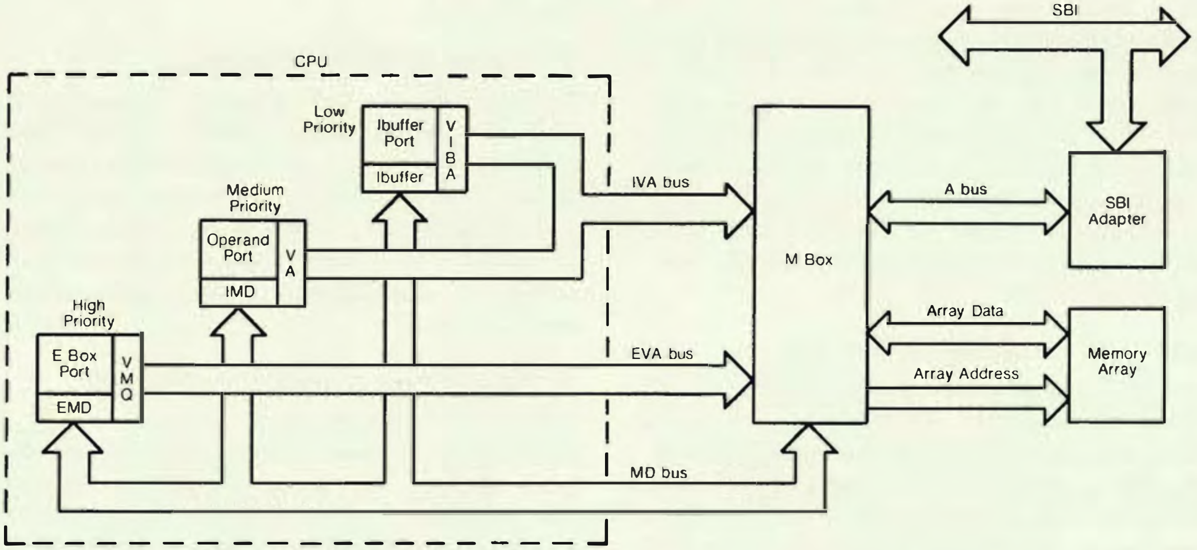


Figure 2 Port Organization

these numerous requirements, the M Box is heavily microcoded and occasionally calls upon E Box microcode to assist with some memory management functions. The M Box has the capability of queuing a number of memory requests from both the instruction fetch and execution units. Both the I Box and E Box can request M Box service through their own memory ports and buses.

A more detailed description of the M Box can be found in reference 2.

I Box–The Heart of the Pipeline

The primary purpose of the I Box is to continuously feed microcode dispatch addresses and operands to the E Box and F Box so that they may execute the VAX instruction set. To do that, the I Box must prefetch the instruction stream from the M Box and then interpret it: parse the specifiers, fetch the operands and build the dispatch address (Efork) for the E Box. Three of the four pipeline stages, including a microcoded operand address calculation engine, are used to implement these functions at high speed. Extensive control logic is needed to synchronize the flow of data and control through the pipeline. Furthermore, the I Box contains the logic to maintain the many program counters representing the different instructions executing concurrently in the pipeline.

The virtual ownership of the pipeline, including the critical E Box dispatch interface, the control of most of the CPU-to-M Box interface, and the maintenance of the program counters, makes the I Box the heart of the pipeline and the object of much of the complexity of the VAX 8600 CPU.

E Box and F Box–The Essence of the VAX Architecture

In general, the E Box and F Box consume the dispatch addresses and operands set up by the I Box and perform only the operations as specified in the opcode of a macroinstruction. In this way, these boxes are isolated from memory access and freed from specifier evaluation and operand fetching. They can thus be optimized for high-speed execution. The E Box also performs the secondary function of managing the boundary conditions for both the hardware (machine checks, such as single- and double-bit memory errors and parity errors) and the VAX architecture (interrupts and exceptions). In particular, most memory management boundary conditions are handled by the E Box. TB misses, page faults and access violations, page crossings and unaligned E Box memory references are detected by the M Box but are all serviced by the E Box. In this respect, the execution units are the essence of the VAX architecture.

System Buses

There are a number of internal buses that are key to the organization of the VAX 8600 CPU and to understanding it. These include the following:

1. IVA bus–I Box virtual address bus, which carries virtual addresses from the I Box to the M Box during instruction fetch, operand fetch, and I Box-write operations
2. MD bus–Memory data bus, which carries data for both reads and writes to the M Box subsystem
3. OP bus–Operand bus, which carries operands from the I Box to the E Box and F Box
4. W bus–Write bus, which carries results from the execution units to memory (via the I Box) or to the general purpose registers (GPRs)
5. EVA bus–E Box virtual address bus, which carries virtual addresses from the E Box to the M Box during E Box operand references and certain memory management routines
6. A bus–I/O bus, which interfaces the CPU to the I/O subsystems

So far we have briefly introduced the fundamental building blocks of the VAX 8600 CPU. We will now analyze it from the more abstract level of its microarchitecture, that is, its pipeline structure. To this end, a model of pipelines is first developed.

The Pipeline Model

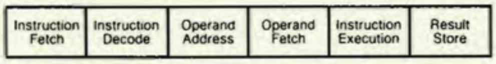
Pipelined computers are not new. From the early days of the IBM Stretch[3] and the IBM 360/91[4] to the scalar units of the CDC[5] and CRAY[6] machines, pipelining has been a proven, if expensive, method for performance enhancement. Such enhancement is achieved by replacing the sequential execution of each instruction step in a single functional unit, with the concurrent execution of some or all steps in multiple functional units.

In most Von Neumann processors, the instruction fetch and decode functions are performed sequentially in the only "stage," the execution unit, which is also the entire CPU. A typical example is the PDP-11 system, in which the concurrency is microprogrammed. (See Figure 3a.)

Most existing VAX implementations have added a stage for instruction prefetch, thus reducing the instruction fetch latency; the prime example is the VAX-11/780 CPU. (See Figure 3b.)

The VAX 8600 CPU is the first implementation of the VAX architecture that separates instruction preparation (for example, effective address calculations and operand fetches) from instruction execution itself. (See Figure 3c.)

The significance of the VAX 8600 design lies in the successful resolution of the implementation difficulties that stem from the combined complexities of the VAX architecture and the



Figures 3a PDP-11 Instruction Execution

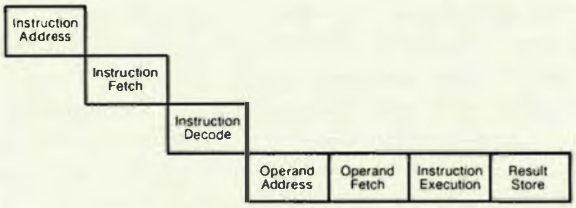


Figure 3b The VAX-11/780 Instruction Pipeline

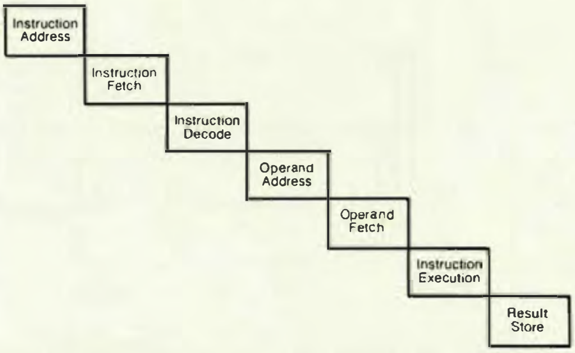


Figure 3c The VAX 8600 Instruction Pipeline

pipeline approach: the more complex an architecture (that is, the greater the control and data dependencies), the more difficult it is to pipeline it.

While the basis and fundamentals for such designs can be found in references 7 and 8, and a more recent pipeline model is discussed in reference 9, we present here a simplified model for the purpose and scope of this paper. Then, using such a model, we describe the VAX 8600 pipeline.

An Ideal Pipeline Model

In this section we define a simple model of a pipeline. Examples from the section on the Simplified VAX 8600 Pipeline Model, described later in this paper, are used to illustrate the abstract concepts presented in this section.

Let us define a pipeline stage, depicted in Figure 4a, as an entity with four fundamental attributes: function, hardware residency, precedence, and the number of stage elements.

The *function* of a stage is usually an input buffer, an output buffer, and a mapping between the two. For example, the function of the operand access unit (OAU) stage is to compute an operand effective address, fetch it from the M Box, and then load it into the output buffer, the I Box memory data (IMD) register.

The ***hardware*** residency of a stage is where it resides in the hardware. For example, the OAU stage resides in the I Box hardware.

The *precedence* of a stage is its position in the sequence of stages. This precedence is fixed and means that the instruction decode stage, for example, is a successor of the prefetcher stage. Note that the precedence relation is a logical concept and not a physical one. For example, although the memory write function of the execution stage is part of the last stage of the pipeline, it shares resources with the OAU stage.

Finally, a stage function is implemented by one or more *elements*. Under optimal conditions, an element processes an item in one physical cycle. However, more than one physical cycle may be needed when the function that the element implements is a complex one, or when the element has to wait for certain resources.

Let us now define a few concepts that are key to the understanding of the pipeline model.

The *logical cycle* of a stage is the number of physical cycles needed to process an item. Under optimal conditions, this number is usually the same as the number of elements in the stage. The reason for this distinction between logical and physical cycles will become apparent with the following examples.

1. In the first example, the OAU stage processes a simple specifier, such as register deferred mode (Rn). In this case, one logical cycle equals two physical cycles: one to compute the operand address and another to fetch the operand itself.

2. As a second example, consider again the OAU stage's processing of a complex specifier, such as longword displacement deferred indexed, @LD(Rn)[Rx], with a cache miss in the indirect reference. In such a case, one logical cycle will equal N physical cycles, where N is directly dependent on the state of various system resources.

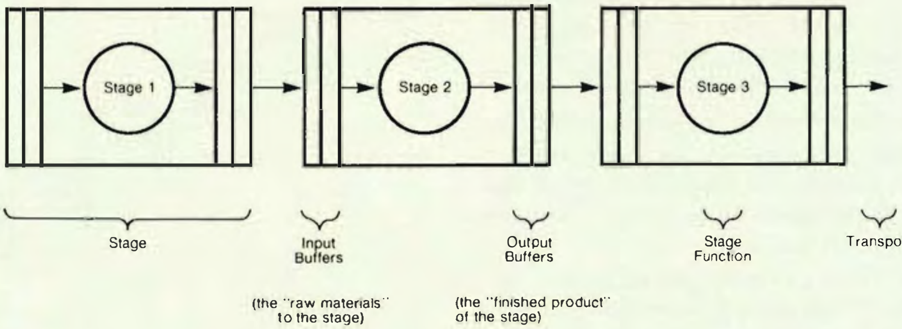


Figure 4a An Ideal Pipeline Model

A *pipeline* is a sequence of stages connected by "transport" mechanisms, which move an item from the output buffer of one stage to the input buffer of the next. Except for the first and last stages, such a structure can be partitioned into a current stage, all its precedent stages, and all its subsequent stages. One can also define the predecessor stage as the immediately preceding stage, and the successor as the one immediately following.

What has been described so far is the datapath of a pipeline.

Control of the Ideal Pipeline

The datapath of the pipeline model just discussed is a somewhat simple concept that belies the complexity of the mechanisms needed to control it. In the ideal case shown in Figure 4a, the relatively simple synchronization is based on "local control" and is implemented by the stall conditions defined below.

Local control is defined as the control of a flow of items through the pipeline by flags that are transported together with the items. These are the valid flags of the input and output buffers. The two basic operations of loading and draining can give such flags the values of either "empty" or "full." These two values are called also "invalid" and "valid" respectively.

Loading occurs at the completion of a logical cycle, when a stage writes an item into its output buffer and sets the buffer's valid flag to full.

Draining occurs at the beginning of a logical cycle, when a stage reads an item from its input buffer and sets the buffer's valid flag to empty.

Depending on the operation and on the values of these flags, one of two stall conditions can occur.

1. An input stall takes place when the valid flag of the input buffer is empty and the stage wants to drain it. Then the stage must avoid loading the output buffer, since it would be loaded with invalid data.

2. An output stall takes place when the valid flag of the output buffer is full and the stage wants to load it. The stage must then stop to avoid data overrun.

Even in the case of an ideal pipeline, an important performance issue is that of elasticity of the pipeline. *Elasticity* is the ability of the pipeline to deliver results at full bandwidth in spite of its irregularity. Irregularity results when different stages in the pipeline have logical cycles of different duration; hence the time to process an item in each stage is variable.

Rigidity, the reciprocal of elasticity, measures the dependence of a stage on the stalled state of another stage. In other words, the rigidity is related to the speed with which the stall flags "ripple" through the stages, in either direction. Rigidity is counterproductive in that it stifles concurrency. For that reason, extra buffering is sometimes used; this allows a stage to execute even if some output buffers are already full, thus reducing output stalls. This also means that the input buffers to the successor stage will be able to be "preloaded," thus reducing input stalls as well.

However, simple FIFO extra buffering may introduce the negative effect of increasing the pipeline *latency* (that is, the number of physical cycles needed by an item to travel through the entire pipeline). This effect can be minimized by the use of "bypass" circuitry, as described in reference 9, at the cost of a significant amount of control complexity. To minimize such complexity, one can reduce the number of input and output buffers in a stage to just one output buffer. In this case the single-stage buffer functions both as the output buffer of that stage and as the input buffer of the successor stage. The VAX 8600 design is very close to this model.

A Model with Pipeline Dependencies

All pipeline models have embedded, via the precedence attribute, the "trivial" dependency of a stage; that is, its dependence on the output buffer of its predecessor stage. However, a more realistic pipeline model (see Figure 4b) must include nontrivial dependencies as well, that is, dependencies of a stage on other than the output buffer of its predecessor stage. Such dependencies can be classified according to their type (data or control) and direction (forward or backward).

A stage has a data dependency if it needs data values produced by a stage other than the predecessor stage. For example, the OAU stage must wait until the E Box has updated a GPR before it can use that GPR in the address calculation, as shown in Figure 4c.

A stage has a control dependency if it needs control produced by a stage other than the predecessor stage. For example, the OAU stage,

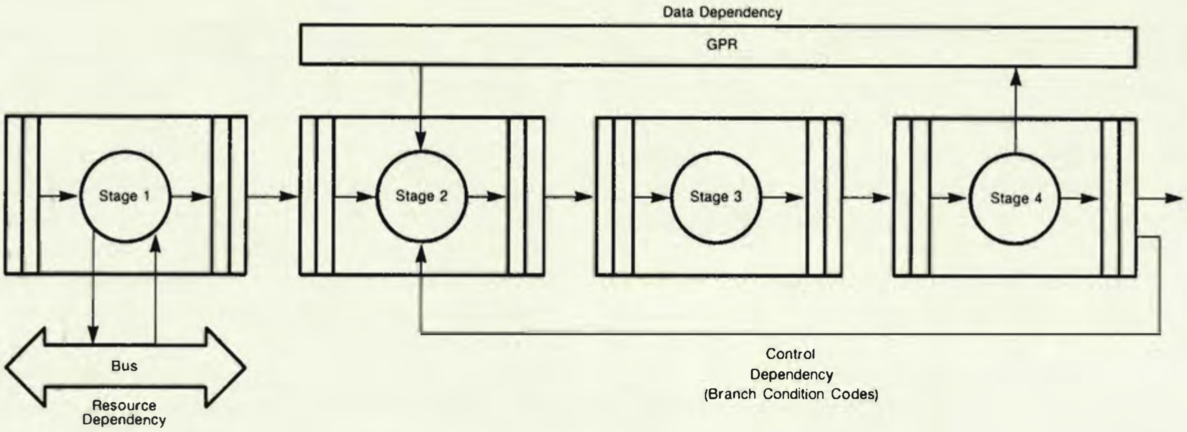


Figure 4b Pipeline Dependencies

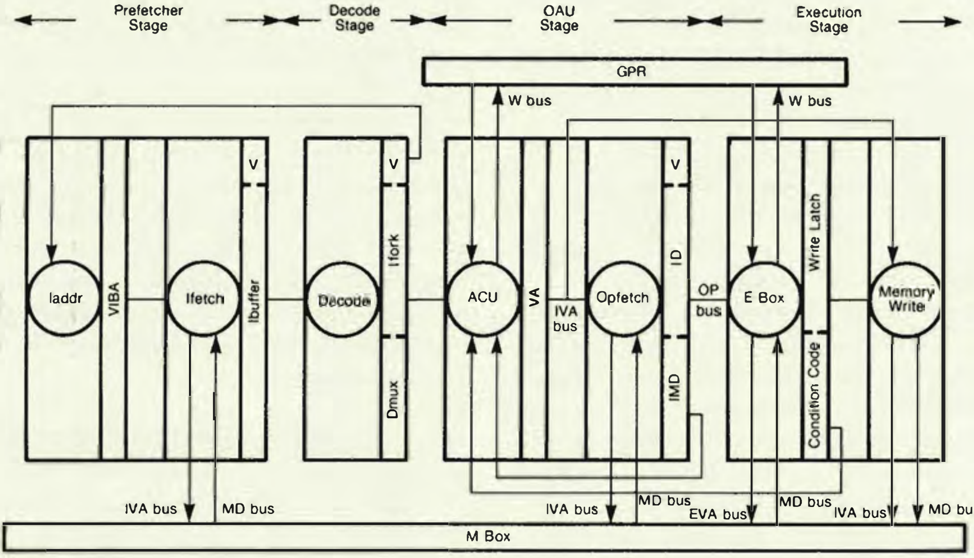


Figure 4c Simplified VAX 8600 Pipeline Model

which also processes branches, must wait until the E Box has generated the condition codes for the instruction preceding the branch. Once the condition codes are set, the OAU can resolve the branch, as shown in Figure 4c.

Each of these dependencies can operate in a forward or backward direction. In a backward dependency, a piece of a data or control item affects a precedent stage. Either example for the control or data dependency illustrates the point. In a forward dependency, a piece of a data or control item affects a subsequent stage. An example is the I Box-write address dependency, which is described in the next section.

In addition to the above, there are resource dependencies, which occur when a stage needs to use a resource shared among many stages. The memory in the M Box, for example, is a resource shared by three of the VAX 8600 stages.

All of these dependencies make the implementation of a pipeline more difficult than in the ideal case. However, they sometimes allow a more efficient global control of the pipeline. This is the control of the flow of items through certain stages by key flags that are broadcast by another stage. Note that this mechanism operates in conjunction with the local control.

In the next section the concepts just introduced will be used to represent the VAX 8600 CPU in terms of the simple abstract pipeline model just described.

The Simplified VAX 8600 Pipeline Model

A simplified model of the datapath portion of the VAX 8600 pipeline is shown in Figure 4c. In this model the F Box is not shown, as its locus of control is very similar to that of the E Box. This four-stage design has two fundamental resource dependencies, which embody much of the logic to control the pipeline: the M Box, which is used by the instruction prefetch (prefetcher) stage and sometimes by the OAU and execution stages; and the GPRs, which are used normally by the OAU and execution stages.

Before discussing the simplified model, let us follow an instruction as it goes through the pipeline.

At the beginning of instruction processing, assume that all the I Box buffers are invalid. In this case the E Box dispatches the instruction prefetcher at the new instruction stream address. The prefetcher stage starts prefetching and loading instructions into the instruction buffer (Ibuffer). This is actually a simplification; the detailed mechanism is described in the Instruction Prefetch section. The instruction decode stage, called the decode stage, drains the Ibuffer and from the opcode generates a dispatch address (not shown in Figure 4c) for the E Box microcode. The operand address calculation unit (ACU) element in the OAU stage parses the operand specifiers and computes their effective addresses, in the process reading and possibly modifying the GPRs (e.g., autoincrement mode, (Rn)+). The operand fetch (Opfetch) element fetches these operands at that effective address and passes them to the E Box. The E Box then executes the instruction it was dispatched to; in doing so, if the destination is a register, it drains the operands and writes the result into the GPRs. If the destination is memory (and only in that case), the memory write (Mem-write) element is used. It takes the result data from the E Box and writes it to memory via the operand port (see Figure 2) at the address forwarded by the ACU element. Such a mechanism is called an *I Box-write*.

Let us now look at each stage of the pipeline of Figure 4c in more detail.

The prefetcher stage is composed of the instruction address calculation (Iaddr) element and the instruction fetch (Ifetch) element. The Iaddr element computes the next value of the virtual instruction buffer address (VIBA) register and issues an Ibuffer request. The Ifetch element fetches a longword from the address pointed to by the VIBA register and loads it into the Ibuffer. The prefetcher stage resides in the I Box. Its logical cycle lasts two physical cycles in the case of a cache hit, or N physical cycles otherwise, where N depends on the memory access delay.

The decode stage is composed of only one element and its logical cycle always lasts one physical cycle. It decodes opcodes and specifiers from the Ibuffer and loads control data into the Ifork buffer (the Ifork will be defined in the Instruction Decode section) and instruction stream data into the data multiplexer (Dmux) buffer. The Ifork and Dmux buffers together form the output buffer of the decode stage. The decode stage resides entirely in the I Box.

The OAU stage is composed of the ACU and Opfetch elements. The ACU element computes an operand effective address, loads it into the virtual address (VA) register, and issues an operand request. The Opfetch element fetches the operand from the M Box and loads it into the I Box memory data (IMD) register. The OAU stage also forwards the VA to the Mem-write element. Note that this stage can contain two instructions at any given time. The OAU stage resides in the I Box, and its logical cycle lasts a minimum of two physical cycles.

The execution stage is composed of the E Box and the Mem-write elements. The E Box element executes instructions and stores results into either the GPRs or the write latch for memory writes. In the latter case it initiates an I Box-write command. The Mem-write element actually performs the write operation at the address forwarded by the VA register in the OAU stage. The execution stage resides in the E Box, F Box, and partially in the I Box for memory writes. Its logical cycle lasts a minimum of one physical cycle; for example, in the case of register destination instructions, such as MOVL (Rx),Ry. It will last at least three physical cycles in the case of memory destination instructions, such as MOVL Rx,(Ry), or even longer in the case of complex instructions.

In the simplified model each stage has only one output buffer, which functions also as the input buffer of the successor stage. Thus a drain operation is implemented as an interstage drain signal. Note that in this case the elasticity of the pipeline is reduced to a minimum. In the worst case, if the pipeline is full and the last stage stalls, then all the stages in the pipeline will stall.

Moreover, since a stall condition must be detected before loading the output buffer, the output stall condition is more stringent in certain cases, as defined below, than the one introduced earlier. The output stall is here defined as a condition in which the valid flag of the output buffer is full and the stage wants to load it, AND the successor stage is not draining it. This means that the stage will stall less frequently. However, note that the input stall condition remains the same as defined earlier.

In such a model there are some interesting examples of nontrivial dependencies.

- The prefetcher stage has two backward control dependencies, the decode and OAU stages, that affect the Ibuffer requests to the M Box. The issuance of such requests by the prefetcher stage requires the knowledge of the validity of the decode stage's output buffer and also whether or not the OAU stage is draining it. These two dependencies are fundamental because they take the place of the prefetcher stage's trivial dependency on its predecessor stage, which does not exist.
- The OAU stage has a backward data dependency, the execution stage, that affects its ability to resolve branches. The OAU stage must wait for the condition codes from the E Box, after completion of the instruction preceding a branch, in order to resolve it and start prefetching at the target address.
- The execution stage has a forward data dependency, the OAU stage, when they together execute an I Box-write command. In this case the OAU stage forwards the destination address to the Mem-write element (as far as hardware is concerned, the address stays in the VA register). When the ACU takes many cycles to compute the effective address, the E Box may have to wait for the disposing of the data.

The VAX 8600 I Box

The three pipeline stages residing in the I Box are physically composed of the following structures:

- An instruction prefetch stage (prefetcher in Figure 4c), which prefetches the instruction stream for the Ibuffer. (This stage is also used to fetch string data in string instructions.)
- Decoding logic, which produces dispatch addresses, based on opcode and its specifiers, for the operand address calculation unit micromachine and the E Box. (This is the decode stage as defined in the pipeline model.)
- A micromachine, called the ACU micromachine, which implements the functionality of the OAU stage and part of the Mem-write element. (This functionality includes operand address calculations, operand fetches and results writes.)

Notice that part of the Mem-write element resides in the I Box. This part maintains the memory write address for results operands and shares responsibility with the E Box element to perform the actual results write.

Furthermore, the I Box maintains the following items:

- Program counters for tracking different instructions being executed at different stages in the pipeline
- A local copy of the GPRs for operand effective address calculations and operand sourcing
- A register scoreboard for resolving register access conflicts
- A register log (Rlog) for register state restoration during exceptions and interrupts
- A branch decision mechanism
- Control mechanisms to synchronize the pipeline

The importance of the VAX 8600 I Box lies in the many functions it has to perform and the extensive controls required to correctly synchronize all four stages of the pipeline. Figure 5 depicts the datapath of the I Box. The following sections describe the functions of many of its features.

Instruction Prefetch

The prefetcher has an eight-byte Ibuffer and associated addressing and control logic. It attempts to initiate a prefetch whenever an "empty" byte is detected inside the Ibuffer. The VIBA register contains the next address in the instruction stream to be fetched from. Prefetch request addresses share the IVA bus with the ACU via the Ibuffer port and operand port respectively. (See Figure 2.) Since an operand fetch is a result of executing an already decoded instruction, it has a higher priority in using the IVA bus. Prefetches, on the other hand, can be postponed and thus have lower priority.

The memory subsystem queue can accept a second prefetch even if a previous prefetch is still in progress. This mechanism results in better utilization of the available memory bandwidth. Data received through the MD bus

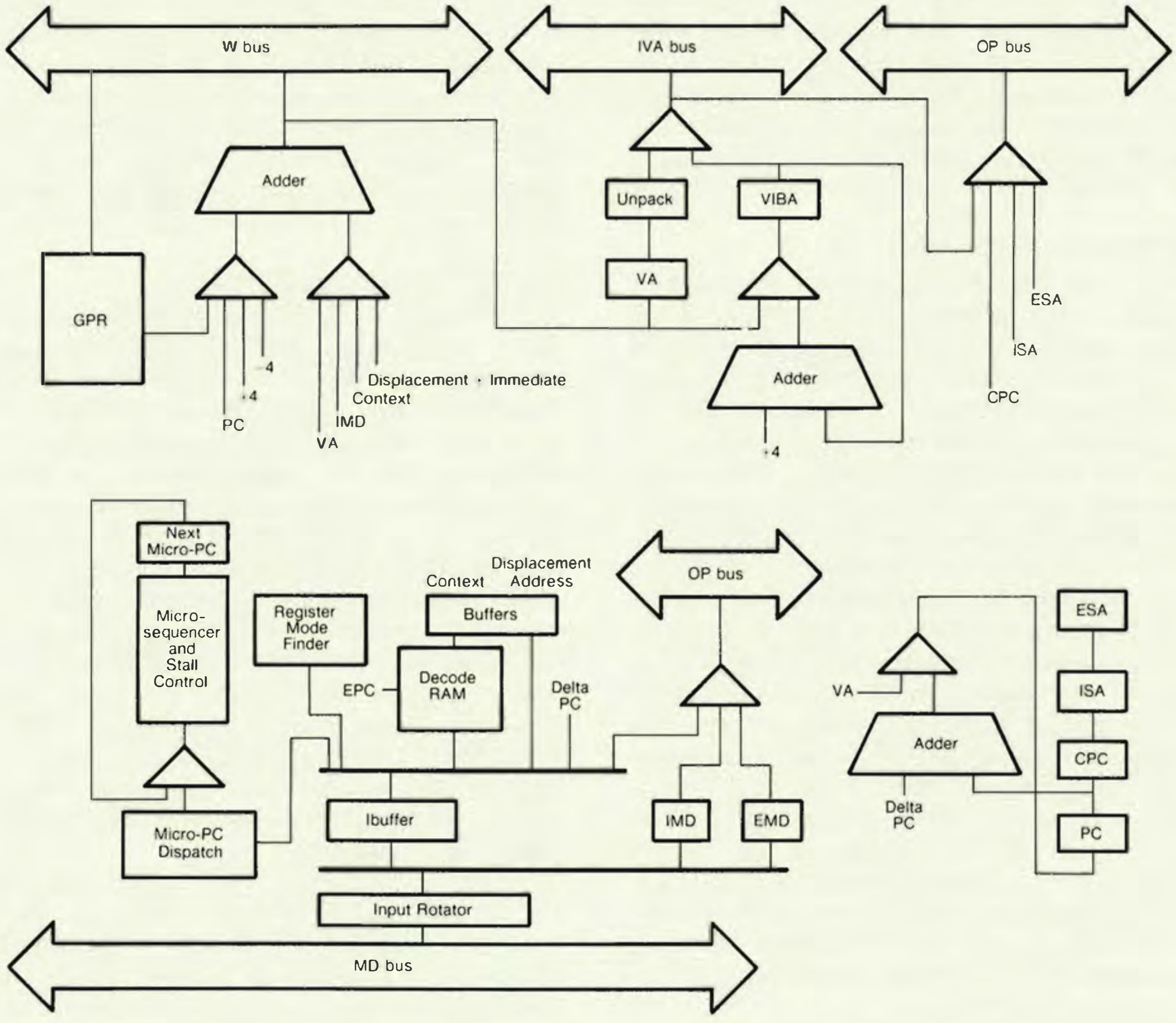


Figure 5 VAX 8600 I Box Datapath

is loaded into the appropriate location inside the Ibuffer. The VIBA register is updated to form the next address whenever a prefetch request is accepted by the M Box.

During a cold start, after an exception, or for certain branches (such as the CASE instruction), the prefetch sequence must start from a new instruction address. In this instance the E Box places the new address on the W bus and dispatches the ACU micromachine to an I Box startup sequence. Instead of loading the address to the VIBA register and starting the prefetching process, the ACU micromachine initiates two consecutive requests before handing the control of the prefetching process over to the prefetcher stage.

For some instructions requiring stream data (e.g., character string instructions) or a stream of operands (e.g., the popping of the GPRs from the stack in the RET instruction) to be read consecutively from memory in their execution, the Ibuffer becomes a high-speed data buffer supplying operands to the E Box through the OP bus.

Instruction Decode

Instruction decoding in the VAX 8600 CPU is similar to that in the VAX-11/780 CPU, in the sense that the operand specifiers are decoded sequentially. When the Ibuffer contains prefetched instructions, byte zero contains the opcode of the current instruction, and byte one the first byte of the specifier currently being decoded. An instruction is decoded by looking up information from a decoding RAM (DRAM), which is organized as an array of 512 blocks, each of which has eight entries. Each entry is addressed by its block and entry index. The opcode byte plus an extended opcode, if there is one, will address the block. The execution point counter (EPC), which is a pointer indicating the position of the current specifier in the instruction, will select the particular entry. The output of the DRAM consists of information specifying the data context (byte, word, longword, etc.), data type (address, integer and different floating point formats), and accessing mode (such as read, write or modify) for each specifier. It also provides the Efork dispatch address to the E Box.

After each specifier decode, the Ibuffer shifts out the consumed specifier and shifts the next specifier into the decoding position. The decode stage also increments the EPC so that the new decode points to the next DRAM entry. The output of the DRAM plus data extracted from the specifier field in the Ibuffer, such as GPR information and literal value, is buffered for the OAU stage.

Using the specifier byte during decoding, a dispatch generation mechanism creates a dispatch address, called Ifork, for the ACU micromachine. This process will continue until the last specifier of the instruction is decoded and consumed. (A bit in the DRAM output will indicate such an occurrence.) When this happens, the Ibuffer shifts out byte zero and the last specifier, thus allowing a new instruction to be shifted in.

To clarify the concepts above, note that an Efork dispatch is generated from the opcode. The dispatch is then given to the E Box to point to the E Box microflow that implements that instruction's algorithm. A similar mechanism is used to process specifiers. An Ifork dispatch is generated from each specifier and is given to the ACU micromachine to point to the ACU microflow that implements that specifier's algorithm.

The ACU Micromachine

With reference to the simplified pipeline model (Figure 4c), the ACU, Opfetch, and Mem-write elements are described here together. In this way, their functionality and synchronization mechanisms can be better understood. The Ifork saved in the decode stage provides the entry to the proper microsequence routine in the ACU micromachine. Using the buffered DRAM and specifier data, the ACU micromachine performs the necessary computations to calculate the effective virtual address, and to initiate operand reads from memory or from the GPRs, if necessary. A copy of the GPRs, which is also called a GPR file, is maintained in the I Box so that register access can be done locally, which is faster. This also allows simultaneous register accesses (reads) by the I Box, E Box, and F Box.

For an operand that comes from a register source, data read from the GPR file, after passing through the ACU adder, will be loaded into the I Box data (ID) register. Immediate data, which comes from the Dmux buffer in the decode stage, takes a similar route through the unpack logic to the same ID register. The operand data is then ready for the E Box via the OP bus. The unpack logic is used to convert fixed point short literals to a floating point format.

For an operand fetch from memory, the ACU micromachine loads the operand effective virtual address from the adder into the VA register and issues an operand fetch request through the IVA bus. The IMD register holds any operand data returned from the M Box before forwarding it to the E Box through the OP bus. If the addressing mode is indirect (e.g., autoincrement deferred), the returned data in the IMD register will be the final virtual address of the operand. Then, the ACU micromachine loads the IVA bus with the IMD register data and issues another operand fetch request. The E Box memory data (EMD) register serves a similar function, but holds memory data returned as a result of E Box requests. Placing the EMD register physically in the I Box eliminates the need for the E Box to interface with the MD bus directly.

The ACU microsequences for many simple and frequently used specifiers take one cycle, so that one specifier can potentially be processed in each cycle. Some examples of such specifiers are (a) the register mode, Rn; (b) the register deferred mode, (Rn); and (c) byte, word, and longword displacement modes, B^D(Rn), W^D(Rn) and L^D(Rn) respectively. The successful processing of an operand specifier in the OAU stage also loads the earlier buffered Efork into a register accessible by the E Box.

The logical cycle of the OAU stage may take many physical cycles. This may happen if the algorithm that implements the addressing mode is a complex one, or if the operand fetch is from memory and it results in a cache miss. In this case the execution stage may have already started executing the Efork microsequence, thus attempting to read and use the source operand, which is not yet available. To resolve this, the OAU stage provides additional operand data-valid flags.

The ACU micromachine also issues the actual operand write request for most instructions. In this case the micromachine saves the calculated destination address and waits until operand results are ready from the E Box. When the results are ready, the E Box will write them, via the W bus, into a register called the write latch, internal to the I Box. The E Box also releases the ACU micromachine to issue the appropriate operand memory write request.

Multiple Program Counters

The VAX 8600 CPU maintains a number of program counters for each of the instructions under execution in the pipeline. This is necessary so that instruction restart is possible after an exception service sequence. The program counters consist of the following items:

- Program counter (PC), which points to the opcode, operand specifier, and immediate data or addresses as they are decoded.
- Current program counter (CPC), which points to the instruction to be executed next in the OAU stage. Normally, this is the instruction currently being decoded.
- I Box starting address (ISA), which points to the instruction being executed in the OAU stage.
- E Box starting address (ESA), which points to the current instruction being executed in the E Box and F Box.

The prefetcher maintains its own instruction stream address pointer, the VIBA register, for requests to fill the Ibuffer.

The updating of the CPC, ISA, and ESA happens when an instruction enters the decode, OAU, and execution stages respectively. In general, the CPC will be loaded with the address of the beginning of the instruction to be decoded. The ISA will be loaded with the CPC when the OAU has started processing that instruction. Similarly, the ESA will be loaded with the ISA when the E Box begins to execute that same instruction.

Instruction Backup and Unwinding

In the VAX architecture, an exception may occur during the execution of an instruction. An example of an exception is a page fault on a memory read. For most instructions the VAX architecture requires that the program state be restored to what it was prior to the execution of the instruction so that, after the exception service sequence, the same instruction can be restarted. For some types of instructions, such as the string instructions, total program state restoration is impossible. In those cases, however, enough of the state is saved and restored so that the instruction can continue its execution from where it was interrupted.

In the VAX 8600 CPU, the parts of the program state that must be restored consist of those

GPRs that have been modified during address calculation, and the instruction starting address. Some addressing modes, such as autoincrement and autodecrement, will modify the GPRs; such modifications are kept in the Rlog. During instruction unwinding (also called instruction backup), the ACU micromachine will restore the affected GPRs from the Rlog. Since a number of instructions can reside in different stages of the pipeline simultaneously, the Rlog has enough entries to allow register restoration for multiple instructions. The PC for the instruction in question will also be restored from either the CPC, ISA or ESA, depending on the state of the pipeline stages. This restoration mechanism is also used to handle interrupts.

Branch Instruction Processing

For most branch instructions, the I Box also calculates their target addresses and performs the branch decisions. These instructions include conditional (e.g., BEQL and BNEQ) and unconditional (e.g., BRB) branches, as well as computed branches (e.g., ACBL and SOBGTR). Such decisions are made by looking at the appropriate condition code bits that result from an execution prior to the branch. The branch prediction scheme used here is biased towards branch taken, which is based on measured frequency of branching data. Figures 6a and 6b show an example of the microinstruction sequence for a branch instruction.

During a conditional branch, the ACU micromachine holds the branch target address in the VA register and will attempt to initiate an instruction fetch from that address before it can make the branch decision. A condition code synchronization signal (CCSYNC) from the E Box signifies that the condition code will be ready in the next physical cycle. In cycle 3, when a CCSYNC is received, the ACU micromachine will issue the first request of the branch target instruction stream. In the next cycle, when the ACU receives the condition codes, it will use them to decide whether or not the branch is to be taken. If the branch is not to be taken, the decision will not be known early enough to inhibit the instruction fetch issued in cycle 3, due to signal delay. In that case a correction must be performed in cycle 4.

A branch-taken decision (Figure 6a) means that the instruction prefetch request was correct, and additional requests can be issued. The I Box then flushes the prefetcher and decode stages, which still hold the old instruction data, and allows the new instruction stream to be loaded and decoded.

A branch-not-taken decision (Figure 6b), on the other hand, causes an abort of the prefetch request initiated earlier in cycle 3 from the target address, thereby allowing the prefetcher and decode stages to resume the processing of the current instruction stream. There is no penalty for branch-not-taken here if the current instruction stream is already in the Ibuffer; the

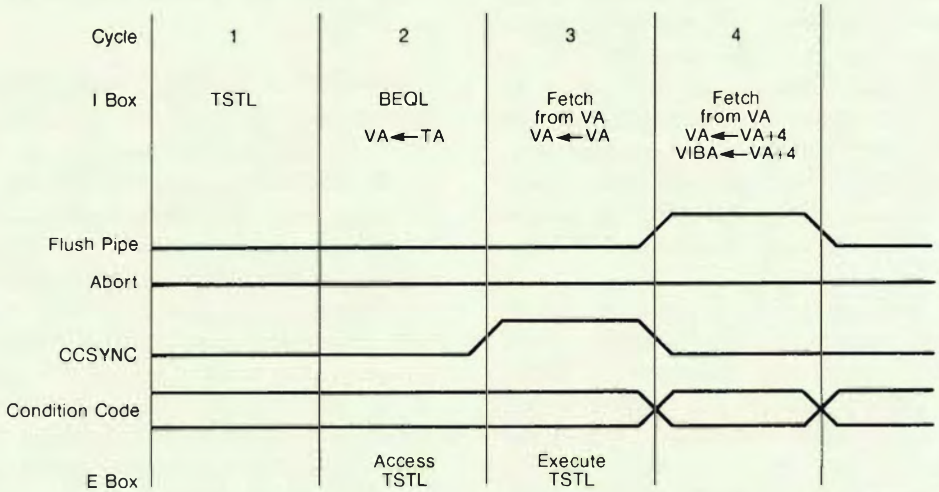


Figure 6a Branch Instruction Taken Sequence

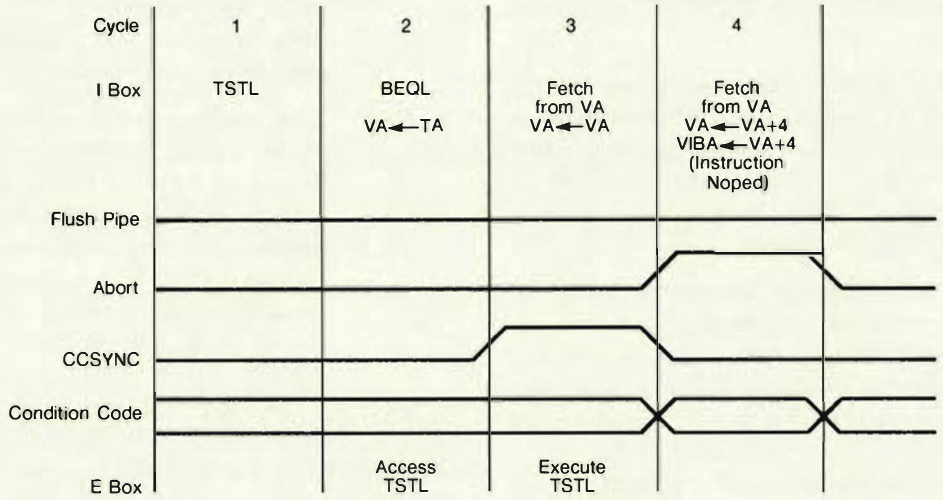


Figure 6b Branch Not Taken Sequence

cost of starting a new instruction stream is thus kept at a minimum. This scheme gives a simple, yet effective, mechanism to handle branches.

The E Box is responsible for handling the remaining types of branches and other instructions that can alter the instruction flow. This includes CASE instructions, subroutine calls, and returns. The mechanism used is the same as that described for cold starts in the Instruction Prefetch section.

Data Dependency Resolution

The use of pipelining in the I Box requires additional mechanisms to resolve data dependency among instructions. Data dependency can happen in many situations; two key examples are the following:

- Register conflicts, which happen when a source operand uses a register that is also the destination register of the previous instruction. For example, in

  MOVL R0,R1

  MOVL (R1),R2

  the sourcing of R1 by the ACU in the second instruction must be inhibited until the first instruction is completed in the E Box.

- Memory conflicts if an out-of-order memory access is allowed. For example, in

  MOVL R0,(R1)

  MOVL (R2),R3

  if R1 equals R2, then the operand read for the second instruction must be postponed until the write in the first instruction is issued. This also mandates that additional collision-detection logic exists.

The VAX 8600 I Box uses a register scoreboard and a single operand port to resolve both types of conflicts. The scoreboard provides a simple reservation-table mechanism to accomplish this resolution. The ACU will enter the GPR number to the scoreboard for every register destination specifier the ACU processes. For every subsequent ACU sourcing from a GPR, the scoreboard is checked to detect any conflict. If such a conflict exists, the sourcing operation is temporarily inhibited via a scoreboard stall. A write to the GPR by the E Box will remove that GPR from the scoreboard, thus allowing the previously stalled sourcing operation to resume. In the VAX 8600 CPU, the scoreboard can be looked upon as a two-entry associative memory structure.

Figure 7 shows an example of the functions of the scoreboard for the instruction sequence discussed in the first example above. The functions performed in each cycle are described below.

Cycle 1 The ACU is processing the MOVL R0,R1 instruction. The scoreboard at this time is assumed to be empty. The ACU reads R0 and loads the ID register. The cycle is completed without problems.

Cycle 2 The scoreboard is loaded with R1 as a result of the previous cycle. Since cycle 2 requires using R1 as the address source, the I Box control discovers that there is a scoreboard "hit" on R1 and the ACU micromachine stalls. It will subsequently attempt to execute the same microinstruction during the next cycle.

Cycle 3 The E Box can now execute the first MOVL instruction, but the result will not be available until the beginning of cycle 4. As in cycle 2, the ACU micromachine still stalls in cycle 3.

Cycle 4 The execution of the first MOVL instruction in the previous cycle by the E Box causes R1 to be drained from the scoreboard. The ACU can now continue and finishes the second MOVL instruction.

Cycle 5 The scoreboard is loaded with R2. As in the earlier stalled cycles, the ACU micromachine will not be able to complete the next MOVL if the next instruction uses R2 in operand evaluation. In that case the ACU micromachine will stall until a write to R2 is completed.

Memory conflicts will not happen in the VAX 8600 CPU because the ACU micromachine controls both the operand read and write for most instructions via the operand port. The micromachine is sequenced in such a way that out-of-order memory access from the I Box is impossible.

Certain instructions whose operand addresses may not be known at the time of decoding (e.g., bit field instructions) will be handled by the E Box. Operand fetching is done directly by the E Box via the E Box port (see Figure 2). In those

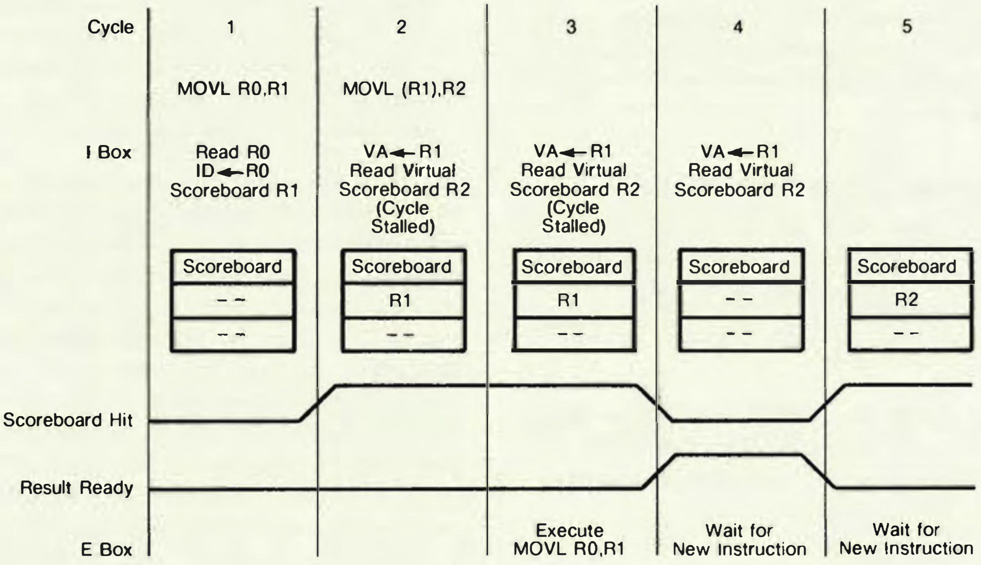


Figure 7 Scoreboard Example

instructions the I Box suspends itself after the completion of the address calculation for all specifiers. Any new operand fetch requests from the operand port will be inhibited by an I Box suspension. This prevents a potential memory conflict from occurring when the I Box attempts to read operands for the next instruction while the current operand result has yet to be written by the E Box.

Instruction Optimizations

The I Box generates a number of optimizations that improve the performance of the CPU. For instructions using a GPR as the results destination, the decode stage will also consume the GPR specifier during the decoding of the specifier immediately preceding it and present a single dispatch address to the execution stage. In addition to the source operand, the I Box also supplies the destination GPR address to the E Box. The E Box will use that address to access its local GPR file. This optimization removes one dispatch from the flow to the E Box.

Another form of optimization eliminates scoreboard stalls when the source operand is in the same GPR to be updated in the future by the previous instruction. In this case the ACU will ignore the scoreboard stalling situation and will signal this fact by presenting a modified dispatch address to the E Box. The E Box will subsequently access the correct updated GPR value from its own local copy.

Pipeline Stage Synchronization

As described earlier in the section on the VAX 8600 pipeline, interstage communication in the VAX 8600 CPU is done through a number of drain signals, as well as a few global flags. Each stage of the pipeline sets the valid flag of its output buffer to full when data is ready. The drain signal indicates to the stage that the buffer is going to be consumed by the successor stage. This will make the valid flag "empty." The global flags are generally broadcast to most other stages. This interlock mechanism provides the basis for the synchronization among the pipeline stages.

Since each stage of the pipeline may take a varying number of physical cycles to complete, there are, at times, empty or full conditions in any of the pipeline stages. An empty condition occurs in a pipeline stage when it wants to drain its input buffer while it is empty. This condition will cause an input stall or a micromachine idling for lack of dispatch addresses. A full condition occurs in a pipeline stage when it wants to load its output buffer while that buffer is full. This condition will cause an output stall. Other reasons, such as resource contention, will also cause idling or stalling.

Each stage uses a different scheme to handle such conditions. In both the prefetcher and decode stages, internal flags are maintained to indicate empty or full conditions. The prefetcher stage keeps track of the number of valid bytes in the Ibuffer and initiates a new prefetch, if necessary. Data removed from the Ibuffer by the decode stage will decrease the number of valid bytes, whereas new prefetched data will increase the number. When the Ibuffer is full, the prefetcher stage will have an output stall (i.e., no new prefetch requests will be issued). The decode stage loads the output buffer valid flag after each decode. It will assume an output stall if the buffer is not being drained by the ACU element. That element, in turn, can drain such a buffer during its execution and clear the valid flag, thereby allowing decoding to be resumed.

The ACU micromachine contains the most complicated stalling and idling mechanisms in the entire CPU. Most resource contention and dependency conflicts, as well as full and empty conditions, can occur in that micromachine.

There are three types of stalling and idling in the ACU micromachine.

1. Resource contention and busy, and dependency conflict stalls. Resource contention examples are (a) the simultaneous update of a GPR by the OAU and execution stages, and (b) the use of certain buses by two resources at the same time. This is best exemplified by the register-dependency conflict detection in the scoreboard. Another form of this kind of stall can result from memory requests not being accepted due to the M Box being busy (that is, while it is servicing previous requests). A full condition, which prevents any further progress of execution, is another example. In general, for this type of stall, the micromachine will suspend the execution of the current instruction and

resume it when the stall condition has been removed.

2. Idling and nops. Empty conditions happen in the ACU, for example, when the instruction decoder cannot provide a dispatch address due to insufficient valid bytes in the Ibuffer. Another nop condition is microtraps, which can be caused by unaligned data references or by the flushing of the pipeline. In both cases the micromachine will execute the instruction, in the sense that a new micro program counter will be loaded, but none of the pertinent machine state will be modified. In the next cycle the micromachine will normally execute a new instruction generated through microtraps or the availability of the next dispatch address.

3. Special stalls. In certain cases in which the purpose of the execution is only to supply dispatches to the E Box, the micromachine will stall to prevent an undesired modification of the state. Part of the state, such as Efork loading, is still allowed. This kind of stall occurs most often for single-byte instructions without any specifiers. In this case a superfluous dispatch address to the ACU micromachine is generated from the specifier field in the Ibuffer, but that address must not be executed lest it modify the state unintentionally. However, the dispatch to the E Box must still be loaded and the appropriate program counter updated.

An Example

An example is given in this section in order to get a more global view of the whole process of executing a piece of code in the VAX 8600 pipeline. The program segment, shown in the E Box in Figure 8, employs two key mechanisms of the design: a branch and an I Box-write. The purpose of this example is to show the following aspects:

- The flow of many instructions through the pipeline, including their uses of the stages, elements and resources
- The state of the pipeline in any given physical cycle, in order to understand the interaction among the various instructions active in the pipeline

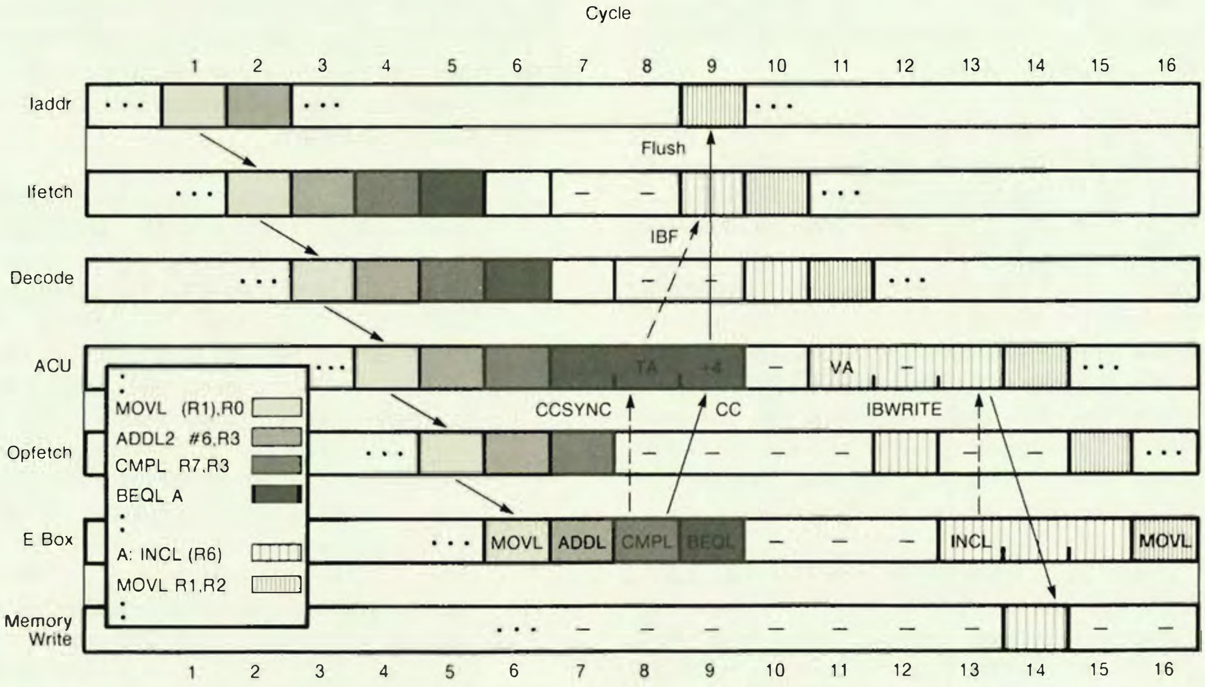


Figure 8 An Example of the VAX 8600 Pipeline

Figure 8 shows how simple instructions, such as the first three in the example, flow through the pipeline in a straightforward way, using only one physical cycle per element. All pipeline elements are then kept busy constantly, thus achieving the VAX 8600 CPU peak throughput of 12.5 MIPS, corresponding to the pipeline executing one macroinstruction per physical cycle. Notice that in this case the results are written to the GPRs, so that the Mem-write element is not utilized. Also, simple memory reads do not stall the pipeline but are performed in only one cycle in the Opfetch element. Moreover, the ACU micromachine immediately starts processing the next specifier after having issued a memory read request; related memory problems, if any exist, will be handled by the E Box.

The branch instruction that follows in the example is one in which the branch is taken. It is therefore processed according to the mechanism described in the Branch Instruction Processing section and in Figure 6a. At the beginning of cycle 8, the CMPL instruction in the E Box sends a CCSYNC to the ACU element, which in turn issues an Ibuffer request (IBF in Figure 8) from the branch target address (TA in Figure 8). This request will result in the fetching of the INCL instruction by the Ifetch element in cycle 9. Also in cycle 9, the condition codes (CC in Figure 8) computed by the CMPL instruction arrive at the ACU element, where they determine that the branch is to be taken. The ACU element then issues a "flush" command to the prefetcher and decode stages to make room for the new instruction stream. Notice that instruction execution will resume in the E Box only four physical cycles after the branch. This is a relatively small penalty for a branch, given that the pipeline latency is normally six physical cycles.

The INCL instruction that was prefetched by the branch mechanism arrives in cycle 11 at the ACU element, where the operand effective address is loaded in the VA register. In the same cycle a memory-read request is issued and the operand address is kept in the VA register until the E Box is ready to do the write. The operand is fetched in cycle 12 and passed to the E Box in cycle 13. Then the E Box performs the increment function, sends the result to the Mem-write element (into the write latch) and issues an I Box-write command (IBWRITE in Figure 8) to the ACU micromachine. This in turn issues the memory-write request to the M Box via the operand port (see Figure 2). The E Box waits two extra cycles after having issued the I Box-write command in order to handle potential memory problems, such as a page fault, before the ESA register is overwritten by retiring the instruction. Execution of the remaining instruction stream resumes normally in cycle 16.

Summary

In this paper, a simplified model of pipeline implementations was introduced. In this model, a pipeline was described as a sequence of stages connected by a transport mechanism, which moves an item from the output buffer of a stage to its successor (i.e., a partial ordering). In connection with this model, the issues crucial to designing a pipeline were discussed, specifically in reference to the implementation of the VAX 8600 CPU and its instruction and operand fetch unit, the I Box. The most important of such issues are as follows:

1. The hand-off of items from one stage to the next–the issue of local versus global control
2. Buffering, which relates to the number of items within a stage
3. The contention for resources and associated stall conditions
4. The dependency of one stage on the activity of another stage (e.g., forward and backward dependencies)

The significance of this implementation, and of the design presented here, lies in the successful resolution of the complex design problems that occur in the pipelined implementation of modern architectures, such as the VAX architecture. Specifically, the use of a register scoreboard to prevent the use of stale register data, a facility to recover in the presence of exceptions, and synchronization mechanisms to deal with VAX-architecture specifics, such as unaligned references, can be considered a major accomplishment. The capabilities of this design–a fourfold speed improvement over the VAX-11/780 CPU, and under favorable conditions, the ability of the I Box to deliver one instruction every 80 nanoseconds to the E Box, which means a peak execution rate of 12.5 MIPS–certainly make the VAX 8600 system a major engineering achievement.

Acknowledgments

A project of this magnitude requires diligent efforts from a large number of people, from the architects and designers to the technicians, and from different support personnel in CAD to manufacturing. The authors are particularly grateful to the following people: Bob Glorioso, Jud Leonard, John Derosa, Rich Glackemeyer, Elaine Hanson, Frank McKeen, Tryggve Fossum, Bill Bruckert, and Jim Lacy.

References

1. *VAX Architecture Handbook* (Maynard: Digital Equipment Corporation, 1981).

2. W. F. Bruckert et al., "The Virtual Memory and Cache Unit of the VAX 8600 CPU," Digital Equipment Corporation Internal Technical Report, Marlborough, Massachusetts (1985).

3. E. J. Block, "The Engineering Design of the STRETCH Computer," *Proceedings of the EJCC* (1959): 48–59.

4. D. W. Anderson et al, "Papers on the IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *The IBM Journal of Research and Development* (January 1967): 8–24.

5. R. G. Hintz and D. P. Tate, "Control Data STAR-100 Processor Design," *Proceedings of the IEEE COMPCON,* no. 72CH0659-3C (1972): 1–4.

6. Cray Research Inc., "CRAY-1 Computer System, No. 2240004," Cray Research Inc., Bloomington, Minnesota (1976).

7. P. M. Kogge, *The Architecture of Pipelined Computers,* (New York: McGraw-Hill, 1981).

8. D. P. Siewiorek et al, *Computer Structures: Principles and Examples,* (New York: McGraw-Hill, 1981).

9. B. W. Lampson et al, "An Instruction Fetch Unit for a High Performance Personal Computer," XEROX Palo Alto Research Center, Palo Alto, California (1981).

Tryggve Fossum
William R. Grundmann
Virginia C. Blaha

The F Box, Floating Point in the VAX 8600 System

The VAX 8600 system contains a processor–the F Box–that performs fast, accurate floating point calculations. The F Box logic design and algorithms are more efficient than those in the VAX-11/780 system, a fact that greatly improves the performance of the 8600. The F Box has adder and multiplier modules that use macrocell array technology to perform the arithmetic calculations and polynomial evaluations. Logic control is achieved with microcode, which decreases the hardware required. Some interesting tradeoffs were made, especially to merge the microcode into the macrocell arrays. The resulting F Box design is a very reliable hardware and software package.

One of our key design objectives for the VAX 8600 processor was to continue the dominant position of the VAX Family in the scientific computing market. That objective required the development of a floating point subsystem that met user demands for increased performance and reliability. This paper describes how we achieved that objective in the VAX 8600 floating point accelerator (FPA) and the considerations that went into its design. We believe that the particular floating point algorithms chosen fit nicely with the component technology to yield a high-performance FPA with a relatively low cost.

The F Box Operations Flow

Figure 1 shows the flow of operations in the VAX 8600 CPU. The F Box receives source operands over the operand bus (OP bus) from the I Box and delivers results over the write bus (W bus). These results are stored in memory or in general purpose registers (GPRs) in the E Box and I Box, and in the F Box itself. The CPU allows two source operands to be processed in a single cycle by passing GPR identifiers between boxes. Each box has its own copy of the contents of all GPRs. Therefore, the I Box needs to pass only the number of a source operand GPR rather than the whole operand itself. This passing technique speeds up the flow of floating point instructions through the pipeline. The I Box passes the opcode of the instruction to the F Box along with the operands. There, the opcode is transformed by the F Box Dispatch RAM (FDRAM) into decoded information that is used by the F Box control logic.

The M Box has a 16KB cache that contains both instructions and data. This box performs the translation of virtual addresses into physical addresses, and it connects to the input/output (I/O) bus and the memory arrays. The E Box executes non-floating point instructions and controls the overall operation of the system. The E Box assists the F Box in executing instructions and handles any overflow and underflow problems.

For more information on VAX architecture, see reference 1.

VAX Floating Point Formats and Instructions

The VAX architecture supports four floating point formats: F, D, G, and H. F and D are

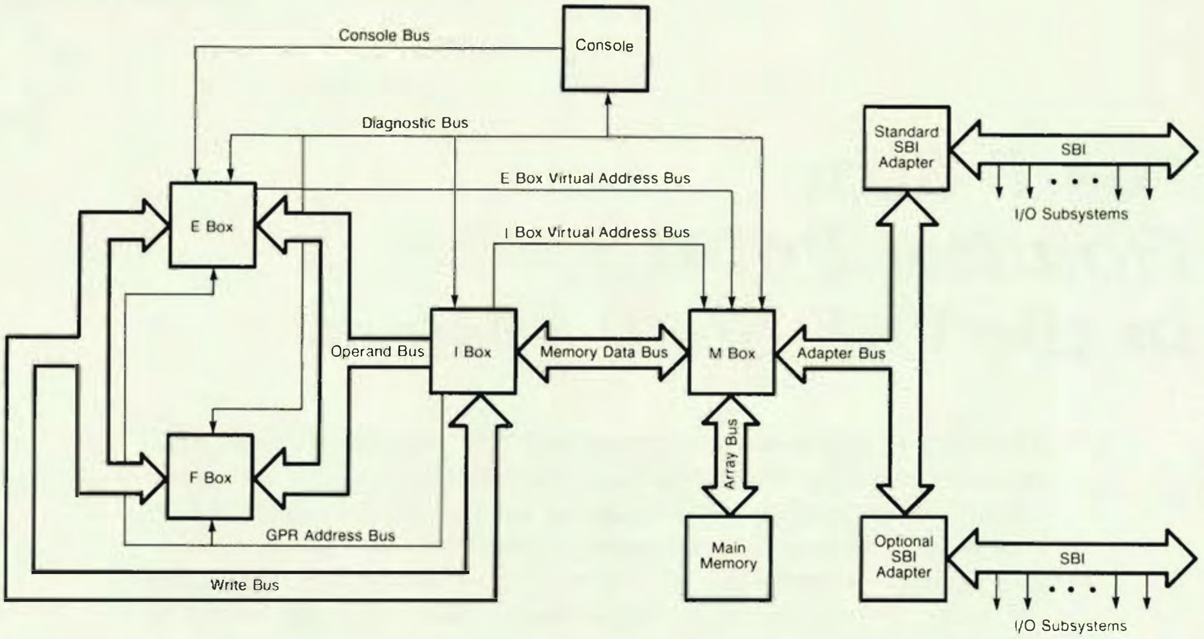


Figure 1 VAX 8600 Operations Flow

the formats from the original PDP-11 floating point processor (1971). These formats are 32- and 64-bits wide respectively, and both have 8 bits of exponent. The G and H formats were added later to the VAX-11 architecture. These formats are 64 and 128 bits wide respectively, the G having 11 bits of exponent and the H having 15. To achieve fast, efficient processing, fractions are always normalized, and the leading bit–the hidden bit–is not stored.

F format instructions execute the fastest of all the floating point instructions on any VAX system and are used in most programs that require adequate precision (24 bits) and range (2^{127} to 2^{-128}). The D and G formats extend that precision and range. The D format provides 56 bits of precision, 3 more bits than the G. Usually, however, the extra range in the G format (2^{1023} to 2^{-1024}) is more useful in performing calculations. The D format is used in programs in which compatibility with earlier VAX systems and PDP-11 systems is important.

In the 8600, the D and G formats have approximately the same execution time, but H format instructions execute more slowly than the others. These H format instructions are implemented in the FPA but are intended for use as a backup format for intermediate calculations in the D and G formats.[2] That use ensures that the final calculation result has sufficient precision and avoids overflow or underflow problems.

The VAX architecture uses either 2- or 3-operand instructions for the basic operations of add, subtract, multiply, and divide. In the 8600, the 2-operand instructions execute faster and are used by the compiler whenever practical. That is certainly the case when the second operand is from a GPR, for then the I Box can optimize the passing of operands by passing the GPR number.

In addition to the simpler instructions mentioned above, the 8600 implements the complex EMOD and POLY instructions for argument reductions and series evaluations.[2] EMOD multiplies two operands that have extended precision and separates the result into integer and fraction components. POLY takes an argument, a degree, and a coefficient table and performs a series evaluation to yield a result. (Both instructions are executed with extra bits of precision.) Complex mathematical functions can be completed in a few steps by using these instructions. For instructions involving integer multiplications, the F Box performs the actual computation while the E Box handles the rest of the instructions. Those overlapping operations decrease the execution time for the MULL, EMUL, and INDEX functions.

For programs in F and D formats, the execution speed of the 8600 is about four times that of the 11/780. For programs in G and H formats, the execution speed is about twelve times faster, since those formats are not accelerated in the 11/780. Table 1 contains the execution times for some typical instructions.

Table 1 Execution Times

Instruction	Operands	Execution Time (nanoseconds)
ADDF2	Mem, R	160
MULF2	Mem, R	320
DIVF2	Mem, R	1300
POLYF	argument, degree, table	(1300 + 6*degree*80)
ADDG2	Mem, R	400
MULG2	Mem, R	800
INDEX		1000
EMULL		640

Macrocell Array Technology in the F Box

The component technology used in the VAX 8600 system is the macrocell array (MCA), which provides about one thousand gate equivalents with a typical gate speed of one nanosecond. MCA utilizes emitter-coupled logic (ECL) technology in a 68-pin package that is one inch square with a maximum power dissipation of 5.0 watts. MCA technology is an extension of the gate array concept; but instead of gates, each cell in the array contains initially a number of unconnected transistors and resistors. By creating interconnecting patterns with these components, a designer can transform them into small-scale and medium-scale integration (SSI/MSI) logic functions, or "macros." These macros take the form of standard logic elements such as dual D-type flipflops, dual full adders and quad latches. All are series-gated ECL structures used in the 8600 to achieve optimized performance.

The F Box has two modules, each containing MCAs. The F Box adder module has 24 MCAs and the F Box multiplier module has 21; in all, the F Box contains 17 different types of MCAs. The adder and multiplier modules are 8-layer (6 signal layers) printed circuit boards. Six signal layers were needed because the amount of etched interconnect on these boards, up to 9000 inches, could not be routed on our traditional 4-layer boards. The interconnect is maintained at a controlled (transmission line) impedance to guarantee signal integrity. We found that the lowest failure rates are obtained when the integrated circuit components on the boards are cooled in a uniform manner. To achieve that cooling, we used wind tunnel techniques to develop algorithms that showed the optimum placement of those components. Moreover, for each module design, we ran computer programs to analyze the thermal profiles of the integrated circuits. These techniques allowed us to determine the best component placements to ensure the highest reliability.

An integral part of the module design is a multivoltage bus bar that distributes power and also acts as a stiffener to maintain board flatness. On its edge, each module has 282 pins that can connect it to a 16-layer backplane.

The connections from the F Box to the rest of the CPU had to be minimized in order to reduce the loading and propagation delays on the signal lines. Therefore, only the adder module and the GPRs have interfaces to the W bus and OP bus. The adder module handles exponent calculation, normalization, rounding, and packing of results. Since only the adder module connects to the CPU, the multiplier module must receive all of its operands from the adder. To increase the speed, we chose algorithms to minimize the signal crossings between modules and between MCAs within a module; for example, addition calculations are done entirely within the adder module while multiplication calculations stay within the multiplier module. The physical partitioning within each module required us to slice the various functions into "pieces" that fit into one MCA. To minimize the number of operational shifts involved, the MCAs on the adder module were partitioned by functions, or horizontally. The MCAS on the multiplier module were sliced by data, or vertically. Figure 2 illustrates the physical partitioning of the macrocell arrays in the F Box, as well as the MCAs on the adder and multiplier modules.

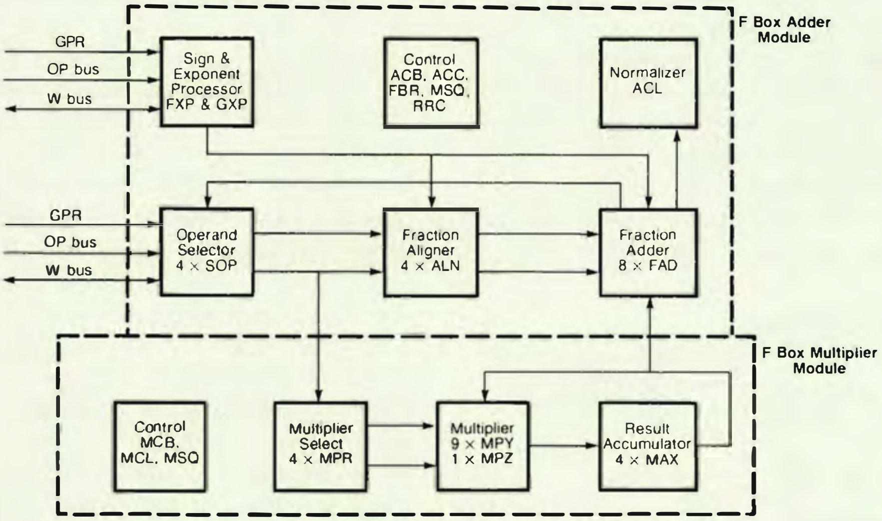


Figure 2 F Box Physical Partitioning

Each module is controlled by its own microcode, which is stored in 256 × 4 RAM components with access times of 7 nanoseconds. The RAM outputs are wired together in pairs to give 512 memory locations. Each microword in the adder module has 48 bits, while each microword in the multiplier has 40.

The 8600 has an 80-nanosecond cycle time, and each cycle has four subphases: T0, T1, T2, and T3. The F Box cycle time is half as long, 40 nanoseconds, and each cycle has just two subphases: T02 and T13. The storage elements in the F Box are level-sensitive latches with the clock pulses set as wide as possible without overlap. That technique yields a lot of flexibility in the placement of the latches without slowing the data flow. Thus we got a simple and reliable clock system by having consecutive latches clocked with alternating clocks. Each MCA needs only two pins for clock signals; thus, more of the available pins can be used for data signals.

Arithmetic Algorithm Processing

Addition and Subtraction Operations

During an addition operation, groups of 32 bits come to the F Box from either the OP bus, the W bus, or a GPR and go into the fraction operand select logic (SOP). Each group also goes to the exponent processor (GXP, FXP) and the sign processor (GXP). Figure 3 depicts the MCA component connections on the adder and multiplier modules.

The exponent processor calculates the exponent difference of the source operands to determine which is the larger; the absolute value of that difference is used to align the fraction of the smaller operand. The alignment and unpacking of each number are combined into one shift by including the unpacking constant in the exponent calculation. The alignment count is passed on to the fraction adder (FAD) MCAs. The larger of the two exponents is kept by the exponent processor to complete the exponent calculation.

In turn, the fraction bits are steered through the SOPs to the alignment circuits in the fraction alignment (ALN) MCAs. Here the hidden bit is restored, the exponent bits are cleared, the larger fraction is unpacked, and the smaller is partially unpacked and aligned. There are four of these ALNs, each containing eight bits of the data path, sliced such that every fourth bit is found in the same ALN. The alignment operation is done in two phases: a byte shift by the ALNs, followed by a bit shift in the FADs. The data is then bit-shifted to complete the unpack and align operations and added or subtracted by the fraction adder logic in the eight FAD MCAs, with four adjacent bits to a slice.

If an addition is being performed, the F Box sends a data-ready signal to the E Box to request access to the W bus for the next cycle. This

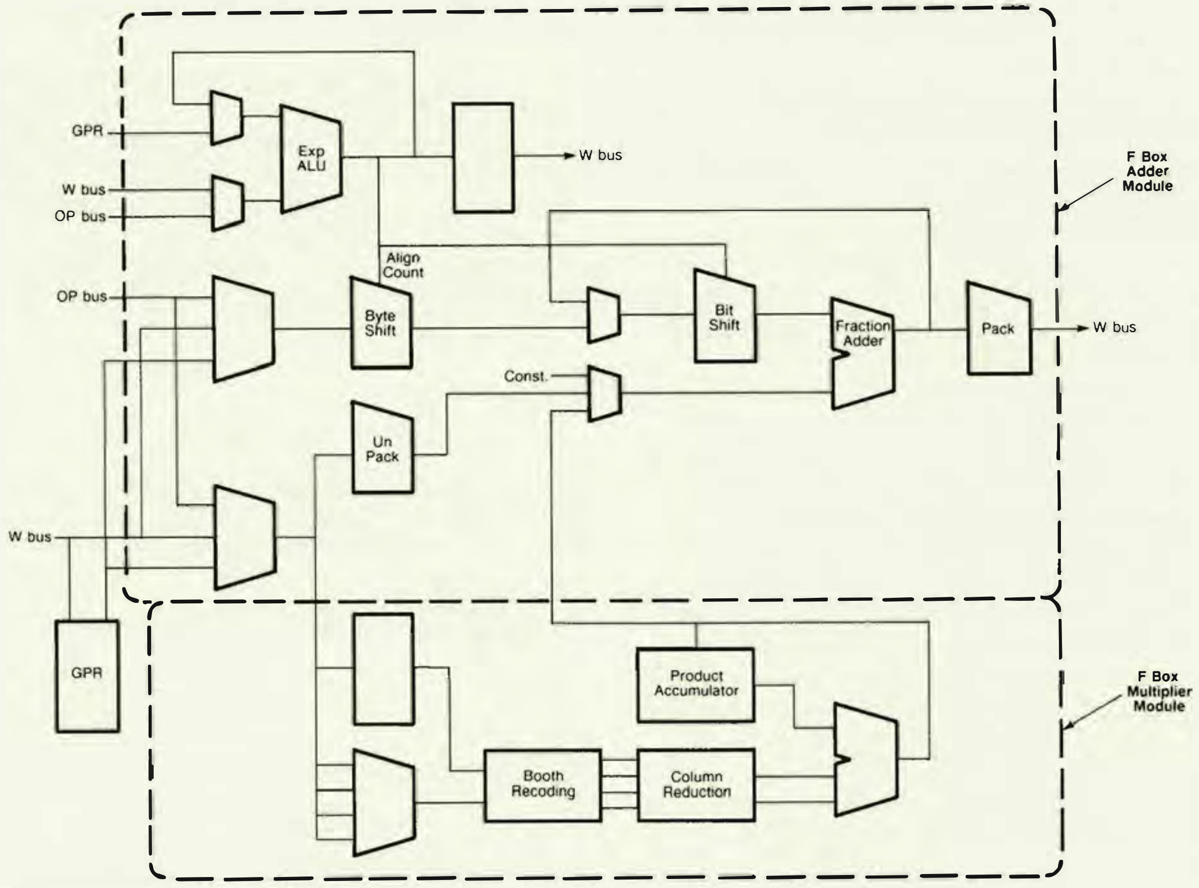


Figure 3 F Box MCA Components

signal is also sent if a subtraction is being performed in which the exponent difference and the high-order bits indicate that the result will be ready within one CPU cycle. On the other hand, if the subtraction is performed on two numbers that are nearly equal, a large number of leading zeroes will result. Those zeroes must be normalized and the exponent must be adjusted before the data-ready signal can be sent. That process takes an extra CPU cycle.

The fraction adders have a bit shifter for both alignment and normalization. In most cases, the number of leading zeroes is less than eight, so the bit shift and the rounding-constant add can be done in one pass. Simultaneously, the exponent processing logic receives the number of leading zeroes and adjusts the exponent for the final result. Then the hidden bit is masked and the result goes back to the four SOPs. There the result is packed into the F format and driven onto the W bus. The SOPs are sliced such that each contains every fourth bit of the result to allow shifting to within the nearest nibble (4-bit piece). The adder module can execute a typical ADDF (an add in F format) in four F Box cycles, or two CPU cycles.

The hardware is arranged so that the "average" floating point instruction executes quickly. The microcode steps through the sequence mechanically while enabling branching to be performed whenever exceptional conditions are encountered. This branching will happen only when something atypical has occurred; for example, when the number of leading zeroes is greater than eight after the add. In that case, the result is passed through the SOPs, and back through the ALNs to be byte-shifted. Then the FADs complete the bit shift and the rounding. This process requires an additional CPU cycle to complete.

The major difference between add operations in the F format and those in the D or G formats is the handling of 32 additional bits of data in the latter two. Rather than making all the datapaths 64 bits wide, we opted to double-cycle the F Box relative to the rest of the CPU. Thus the first group of 32 bits of a number in D or G format is handled during one cycle and the second group is handled during a second cycle. As the first step of the path, the exponent processor calculates the exponent difference, an 8-bit operation in F format and an 11-bit one in G format. Then the high-order fraction bits are unpacked, aligned and stored in a register in the ALNs and in another register in the FADs. As the low-order fraction bits arrive during the next CPU cycle, they are unpacked and aligned through that same path and merged with the appropriate bits in the FAD registers.

The low-order fraction bits are then added together and that result is passed to the SOPs to be held in an internal register. In turn, the high-order fraction bits are added, and the low-result bits are passed back through the ALN to the FAD inputs–the assumption being that the number of leading zeroes is less than eight (no byte-shift is required). Once the high add is completed, a leading-zero detector determines if that assumption is correct, which in most cases it is.

Immediately after the high add, the low normalize-and-round is done. If it turns out that the number of leading zeroes is greater than eight, this result will be discarded. The microcode will guide the old sum through the byte shifter to the input for the fraction adder, yielding a normalization of up to 32 bits. If the microcode has not branched, the high normalize-and-round is done. At the end of this cycle, the hidden bit is masked and the result is passed to the SOPs, which then pack the high-result bits and drive them onto the W bus. One CPU cycle later, the low-result bits are driven onto the W bus.

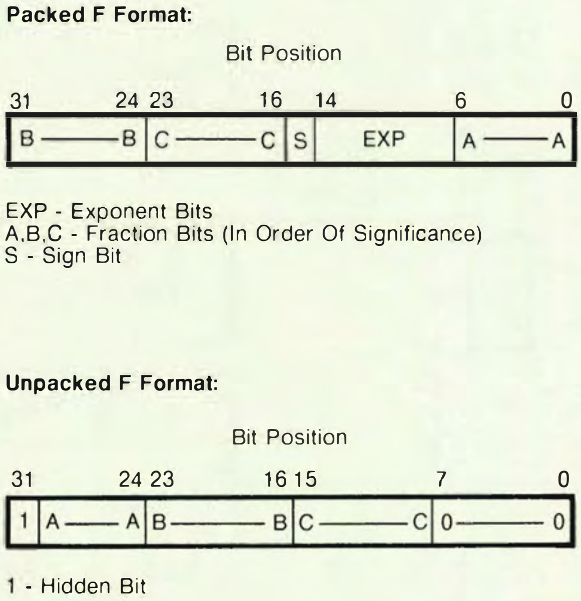
The total time spent in the F Box to perform operations on D and G formats is ten cycles, or a total of five CPU cycles.

Multiplication Operations

To perform multiplications, the operands are sent from the SOP MCAs to the F Box multiplier module. There the floating point formats are unpacked as follows: the leading bit is placed in the most significant position; the fraction bits follow the leading bit, in order of significance; and finally the cleared exponent and the sign bits. Figure 4 illustrates the conversion of a number in F format.

The conversions of the D and G formats are similar, although they have 64 bits instead of 32. Figure 5 illustrates the packed D and G formats, where the G format has three extra exponent bits. The H format has 1 sign bit, 15 exponent bits, and 112 fraction bits.

Four multiplier select (MPR) MCAs are used to store the source operands. The MPRs feed the

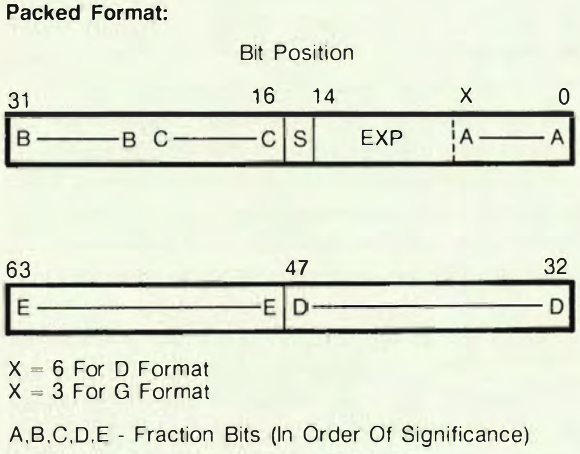

Packed F Format:

EXP - Exponent Bits
A,B,C - Fraction Bits (In Order Of Significance)
S - Sign Bit

Unpacked F Format:

1 - Hidden Bit

Figure 4 F Format Conversion


Packed Format:

X = 6 For D Format
X = 3 For G Format

A,B,C,D,E - Fraction Bits (In Order Of Significance)

Figure 5 D and G Format Conversion

multiplicand in 32-bit pieces and the multiplier in 8-bit pieces to the multiply logic.

On the multiplier module there are 10 MCAs (9-MPY, 1-MPZ) that perform the actual multiply operation, each one generating a 4-bit slice of the product. Each MCA has column reduction logic that consists of a 4-bit, 5-stage adder that adds the partial products, carries previous partial products, sums, and carries from the previous stage to create a new partial sum. All five stages are performed during every cycle. Each slice receives a byte of the multiplier and 12 bits of the multiplicand. A trailing zero and two leading zeroes are concatenated to the multiplier. Then it is divided into five groups of three bits each, called "triplets," to determine the Booth encoding. Finally, each triplet is multiplied by the multiplicand according to the Booth algorithm. Figure 6 shows the eight bit combinations and the corresponding products.

As each byte is multiplied, a 40-bit partial product is held in an accumulator latch. As the processing sequences, the product of the next 8-bit multiplication is added to the last partial product in the accumulator latch, thus producing a new partial product. This cycling continues until the multiplicand has been multiplied by all the multiplier bytes. The normal execution time is reduced by one cycle because the last byte of multiplier has the cleared exponent bits in it.

In F format, the first 8-bit × 32-bit partial product is formed, then shifted 8 bits to the right and loaded into the accumulator. The next 8 multiplier bits are multiplied by the multiplicand, then added to the accumulator and shifted right by 8 bits, and finally stored. A third such product is formed, added to the partial product, and the result is stored in the extended accumulator latches, ready to go to the adder module.

The D and G formats are processed in a similar manner except that sixteen 8-bit × 32-bit multiplies are required to accomplish that task. After all of the multiplier bytes have been multiplied by the least-significant 32 bits of the multiplicand, they then have to be multiplied by the most-significant 32 bits. Prior to the start of that multiplication, the partial product is shifted left by 24 bits to align it for subsequent addition to the next partial product.

The Wallace Tree in Figure 7 illustrates the D and G format processing.

Booth Encoding			Product
Booth Pair		Carry-In From Previous Encoding	
0	0	0	0 × Multiplicand
0	0	1	+ 1 × Multiplicand
0	1	0	+ 1 × Multiplicand
0	1	1	+ 2 × Multiplicand
1	0	0	− 2 × Multiplicand
1	0	1	− 1 × Multiplicand
1	1	0	− 1 × Multiplicand
1	1	1	0 × Multiplicand

Figure 6 Booth Encoding

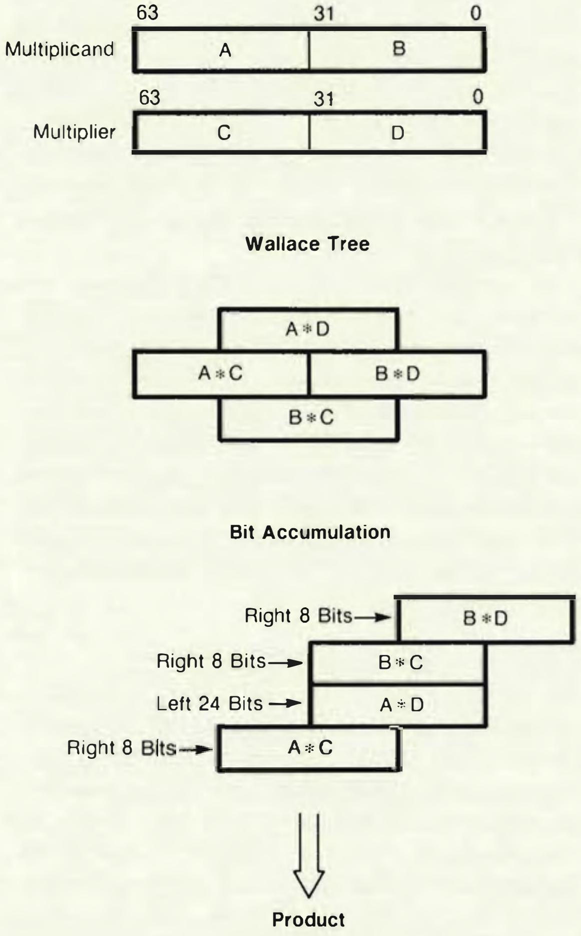


Figure 7 Wallace Tree

Since only 40 bits can be stored in the MPY slices, the overflow is sent to four extended accumulator chips in the result accumulator (MAX). During every cycle, the MAX receives the least significant byte from the accumulator in the MPY slices if a right-8-bit shift is being performed. Or, if a left-24-bit shift is being performed, the MAX receives the most-significant 24 bits of the accumulator and gives MPY the 24 least significant bits from previous accumulations. After a left shift, the MAX sends the most significant byte to the accumulator for the succeeding right-8-bit shifts. After all the multiplications have been completed, the 64-bit result is stored in the MAX, ready to go to the adder module to be normalized, rounded, and packed.

Division Operations

The fraction adder (FAD) performs a non-restoring division algorithm, one bit per F Box cycle. A control input in the FAD causes the adder module to do an add or a subtract, depending on the carry out of the previous fraction adder operation. The bit shifter will keep shifting the dividend to the left by one bit every cycle. For the F format, this shift produces a quotient bit every F Box cycle, while the double precision formats, D and G, get a quotient bit every other F Box cycle.

To save hardware on the adder module, the quotient bits are sent to the multiplier module, where a counter (a split between the MPR MCAs) and several shifters (in the MAX MCAs) are used to manipulate the quotient bits into the correct form. That is, the most significant bit is placed in bit position 31, the next most significant bit in position 30 . . . down to the least significant bit. Then, the bits are sent back to the adder module for normalization, rounding, and packing.

Exponent and Sign Processing

For all operations, the exponent processors (FXP and GXP) calculate the result exponents based on the input operands and normalization counts. Each processor has an 11-bit datapath for exponent operations and a 2-bit counter for accumulating carries and borrows out of the leading bit. Counters are used at the end of the instruction to detect overflow and underflow problems. A non-zero counter number indicates that a problem has occurred. In that case, the F Box sends a signal to the E Box when the calculation result is transmitted over the W bus. In turn, the storing of that result is prevented, and a section of the E Box microcode is "trapped" to a routine that reads several F Box status registers in the FBR MCA. The microcode then identifies the problem and initiates the exception processing.

The sign processor in the GXP is a 1-bit datapath, modeled in a fashion similar to the exponent datapath; in fact they share the same control and microcode signals. Instead of an adder, however, this processor uses a multiplexer and an exclusive OR (XOR) gate to perform sign operations.

Polynomial Evaluations

Polynomials are evaluated using Horner's Method, through a series of multiplications and additions. In the VAX 8600 system, the I Box prefetches coefficients from the M Box, and the E Box keeps track of intermediate results, decrements the degree, and deals with exceptions and address translations. The F Box performs the arithmetic steps described in the Addition and Multiplication sections above. All of these operations are done in parallel.

Microcode Control in the F Box

Like every other subsystem in the 8600, the F Box is controlled by microcode. Microcode offers a structured yet flexible and economic way of implementing the control functions. For complex instructions–such as polynomial evaluations–microcode is essential for sequencing through the various steps. Even for the basic operations like add and multiply, microcode is helpful in dealing with unusual conditions. The achievement of a compact hardware design depended on the use of hardware units like adders and shifters for multiple purposes, and microcode provides sufficient control to achieve that design. Moreover, microcode control allowed us the flexibility to implement fault detection and fault isolation procedures so that manufacturing and field service could effect repairs using microcoded diagnostic programs.

We had to make several design restrictions in order to cycle the control store during each F Box cycle. For example, each module needed its own microsequencer and control store. And except for initial dispatching, the two microcodes run independently.

We latched the microword internals to the MCAs that used them in order to save propagation time and to eliminate the need for additional MSI components. The microfields were highly encoded due to the limited number of MCA pins available. That high level of encoding allowed us to make the whole control store relatively narrow–48 bits for the adder module and 40 bits for the multiplier module. That makes it easier for the F Box to access the control store during each cycle. Inside the gate array, the F Box can decode the microcode into a large number of control functions, some of which are applicable over several cycles. The control signals are pipelined along with the data and the F Box gradually decodes those signals at each stage (see Figure 8). The results of these data operations are sometimes fed back into later decode stages. This microcode style was needed, in particular, to accommodate the pipelined structure of the datapath, where several operations take place simultaneously.

The result of those design restrictions was a scheme in which the microcontrol bits follow the data for several cycles, being further decoded at each stage. For the majority of cases, the microcode is little more than a decoding of the opcode, allowing the hardware to do alignments, additions, normalizations, and roundings. The microsequencer takes over only if the instruction does not fit the standard path and creates the needed result by using the available hardware functions.

We had to define the operations at each cycle early in the design stage in order to get this tight fit between the microcode and the hardware. That was possible due to the relatively small number of operations involved in floating point processing.

The short cycle time of the F Box complicates the control of microcoded branching. Each control store location contains a NEXT ADDRESS field. To change control flow, the microcode selects up to three branch conditions at a time. The OR of these conditions and the low three address bits select the next microword to be executed. The selectors are controlled by a branch enable (BEN) field in the microword. The BEN field of a microinstruction does not affect the next micro-PC but does affect the one following it. (This is called "delayed branching.") The delayed-branching algorithm complicates the microprogramming, since branches-in-progress always have to be accounted for. Figure 9 shows the different inputs and how they affect the next microaddress.

The microsequencer contains no stall signals. Instead, the microcode branches on conditions that will force it to change flow. Again, that microcode design simplifies the hardware design, since stall conditions can be encoded into normal control signals.

The I Box sends the opcode of the instruction to the multiplier module. There the opcode is used to address the dispatch RAM containing

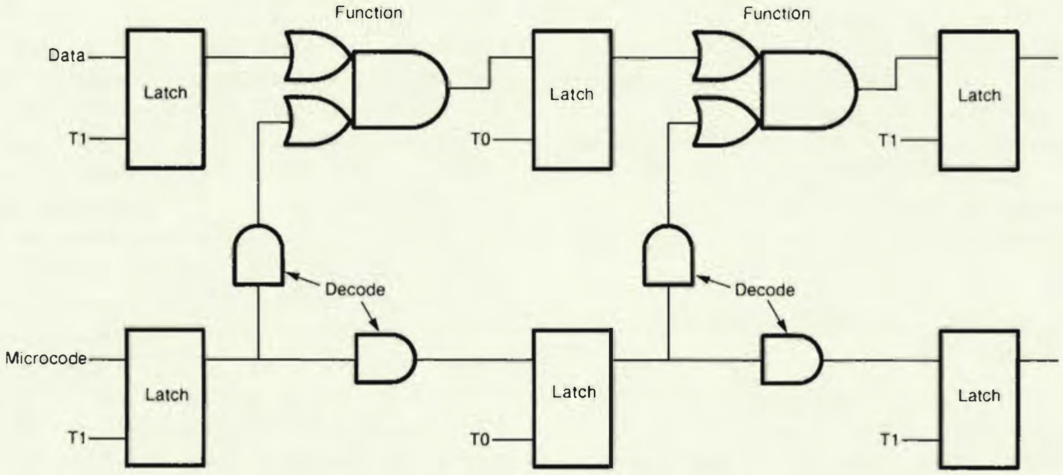


Figure 8 Microcode Control in the F Box

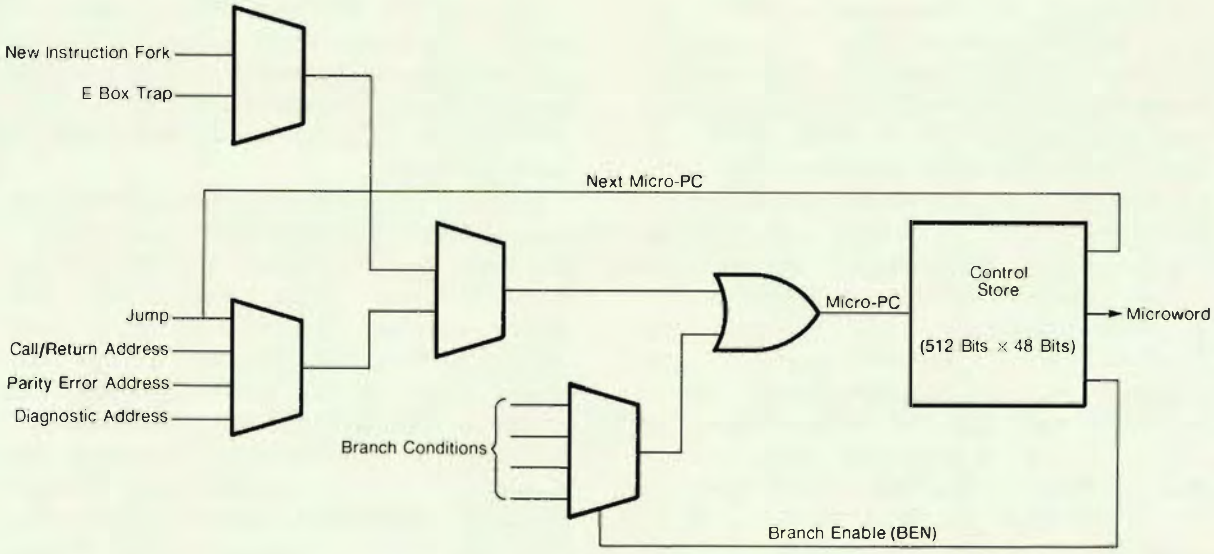


Figure 9 The F Box Microsequencer

the starting microcode address for the instruction. The same starting address is used for both microsequencers. The dispatch RAM also contains format bits that are used to control certain hardware operations. An instruction register decode (IRD) signal from the E Box triggers the start of a new instruction. A "flush" signal from the I Box is used to reset the microsequencer in case of a change in the instruction stream, normally due to a branch or an exception. Without this signal, due to the pipelining of instructions, the F Box might have started on a floating point instruction following the branch. Such an action would have put the F Box out of sequence with the I Box and E Box.

The E Box has the ability to trap the F Box to various microcode routines. That ability is useful when the program wants to use the F Box to execute subroutine functions in complex instructions, or when the program wants to write customer-originated microcode in the E Box.

Error Checking and Reporting Using Microcode

High reliability was a major goal for the whole VAX 8600 system. We used very reliable parts, conservative design rules, and a small number of components to design an inherently reliable machine. Furthermore, we implemented extensive checking for errors throughout the CPU. Our primary recovery strategy was to retry the macroinstruction. If an error is detected, the CPU will make every effort to preserve its state so that the macroinstruction can be restarted after the error has been logged.

The F Box has sufficient idle time to run diagnostic tests on itself while non-floating point instructions are executing in the E Box. This idle time exists because the F Box is involved in the execution of only a subset of the total instruction set. In these tests, the opcode is used to address the FDRAM, and a dispatch address for microcode is generated for a test of appropriate length. Operands are gathered from the OP bus to create a variety of test patterns. The microcode test runs through the basic floating point operations and checks the result. If an error occurs, it will be logged by the F Box and reported to the E Box the next time that a floating point instruction is encountered. In this way, although the CPU is not disrupted, the F Box cannot be used until the error has been evaluated by the VMS operating system.

The error analysis software processes the error report. Since the CPU does not require the F Box in order to run, it can be temporarily disabled by the operating system if the error frequency is sufficiently high. In that way, computing can continue until the F Box can be repaired.

Like the other subsystems in the 8600, the F Box is connected to the maintenance processor, the console, over the serial diagnostic bus

(SDB). The console and SDB are used to initialize the control store and other RAMs. The SDB is also used to alert the CPU to signals required to diagnose failures encountered in manufacturing test or at customers' sites. Parity errors in the control store are corrected on-line by the operator using the console.

Acknowledgements

Many people worked on the development of the VAX 8600 F Box. Major contributions were made by Ray Boucher, Jud Leonard, Dan Stirling, Milt Shively, Steve Root, Linda Pinto, and Larry Herman.

References

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

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

James B. McElroy

Packaging the VAX 8600 Processor

Important packaging decisions were made early on the VAX 8600 project. First, the numbers of gates and parts were estimated to size the CPU. Then, a packaging evaluation method was developed to weigh the effects of various design factors. Packaging the components to control temperature gradients was an important task. Several techniques for mounting devices were tried and the pin grid array was chosen. The module design is an equilibrium between component density and the number of signal layers. The tools developed for packaging decisions and the cooperation engendered among engineering disciplines will help future design projects.

The role of packaging in the product development process has changed significantly in recent years. Today, the electronics packaging engineer must get involved earlier than ever before. He must make a vital contribution toward creating the actual design process, in addition to performing the traditional role of hardware design and evaluation.

Accomplishing this expanded function requires the creation of effective and flexible tools for testing and evaluation, in addition to rigorous adherence to the best traditions of good engineering practice in the management of a large and complex project. The importance of such tools was compellingly demonstrated during the development of the VAX 8600 processor. The tools developed and the lessons learned from designing the packaging for this machine can assist future computer design efforts by making product development more predictable. As a result, new systems can be developed in less time, with less cost and risk.

At the beginning of a development project, little reliable information is available about the physical characteristics of the product. Generally, packaging engineers are forced to rely on extrapolations from previous products and early estimates by system designers. But this initial information is the basis for packaging and interconnect decisions that must carry through the development cycle and often through the life of the product as well. On the other hand, from time to time, it may be prudent to make "midcourse" corrections based on current developments and maturing technologies. Hence it is necessary to implement a design process that constantly inspects the "state of the design" and provides early warning of potential problems.

Ascertaining the Task

Among the initial questions to be answered for any design project are, (a) what is the size of the task? (b) what will the product be made of? (c) what requirements must it satisfy? In addition to the many safety and EMI regulations that the 8600 had to meet, we decided it was necessary to package the system in accordance with the new European standards for noise emissions in data processing equipment. These standards are considerably more stringent than those by which any previous Digital computers were built.

Another early decision was to implement the CPU with LSI macrocell arrays (MCAs) supported by small-scale and medium-scale integration (SSI/MSI) emitter-coupled logic (ECL) and RAMs. An internal Digital maintainability study indicated that costs for spares could be reduced substantially by providing for on-site replacement of MCA and RAM chips. Therefore, it was agreed that those components would be mounted in sockets.

To determine the size and organization of the CPU, we worked initially with the logic designers to estimate the counts of gates, parts and modules and to determine the makeup of memory and the I/O ports. Table 1 compares the numbers of gates and parts in the VAX-11/785 CPU with the early estimates for the VAX 8600 CPU. The last column gives the same data for the final product; some estimates were fairly close, others were not. Much of the increase in gate count comes from the increased use of pipelining to improve performance and from additional diagnostic features. This trend will continue in future design projects.

Table 1 Gate and Part Counts

	VAX-11/785	VAX 8600	
		Early Estimate	Final Design
Gates	68K	88.5K	104K
RAM bits	1.06M	1.05M	1.04M
SSI/MSI	2600	260	1100
MCAs		141	145
Modules	26	10	17

To estimate the number of MCAs, besides the gate count estimate, we would have to have known the design efficiency factor–it is improbable that each array will use 100% of the available cells due to routing inefficiencies and power/thermal limits. Initial component estimates are rough at best, so a conservative safety factor was included to prevent difficulty when the actual counts became known.

Evaluating the Choices

Once it was determined what was being built, we faced a multitude of individual implementation decisions related to choices of sockets, heat sinks, connectors, cables, and so forth. To facilitate the decision process, we developed a procedure for comparing the effects of the various alternatives in each instance and thus to help us select from among them. The first step in utilizing this procedure is to determine which system factors are significantly affected by the decision and the relative importance or "weight" of each (such that the weights sum to unity). Then for each factor some method is devised for quantifying the effect of each alternative to arrive at a rating on a scale from 1 (low) to 10 (high). Finally, in order to be able to compare the total "scores" of the alternative solutions, the ratings were converted to "normalized" values by multiplying each by the corresponding weight.

The alternatives and their impact on the various factors can be listed in a matrix; an example of this is shown in Table 2. Here the choice is between two overall packaging/interconnect structures, one using individual heat sinks to cool the MCAs, the other employing a heat-pin planar approach (both are discussed later). Different parameters play a role in different decisions. Often these parameters are difficult to quantify early in the project. It is important, however, to understand the relative differences between the competing concepts so that a rating can be attached to each factor.

Thermal Design

Thermal design in the VAX 8600 processor was especially critical because individual MCAs can dissipate up to five watts. Both project risk and marketing considerations required using air convection for heat removal. We investigated two approaches to the problem. One employed an individual heat sink, or exchanger, on each MCA, wherein heat was conducted through the device carrier to an omnidirectional heat sink mounted by a thin layer of epoxy. The other was a large, finned heat sink covering the entire back of the module. Conductive pins protruding through the board conducted heat from the MCAs to the exchanger. In either case all other components were to be cooled in the traditional way, by forced air convection. Figure 1 depicts the "heat-pin" arrangement. Using heat-dissipating dummy devices, we conducted temperature and airflow experiments to determine the thermal densities and device placements that would be used for the product. To predict temperatures, we used a thermal analysis tool developed by Digital's Thermal Engineering Group to model the actual modules as they would be in real operation.

There were two possibilities for using the ganged heat exchanger. One involved a single

Table 2 Packaging and Interconnect Evaluation

		Individual Heat Sink Packaging		Heat-Pin Planar Packaging	
System Factor	Weight of Factor	Rating	Value	Rating	Value
Reliability	.20	6	1.20	7	1.40
System Performance	.15	5	.75	5	.75
Risk	.20	7	1.40	4	.80
Development Cost (Engineering and Manufacturing)	.10	6	.60	3	.30
Spares Cost	.10	5	.50	4	.40
Design Process	.05	5	.25	3	.15
Acoustics	.05	5	.25	7	.35
Product Cost	.10	5	.50	6	.60
Signal Integrity	.05	5	.25	4	.20
Total Value			5.70		4.95

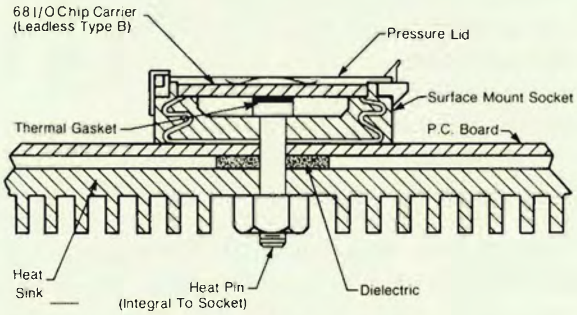


Figure 1 Heat Pin Detail

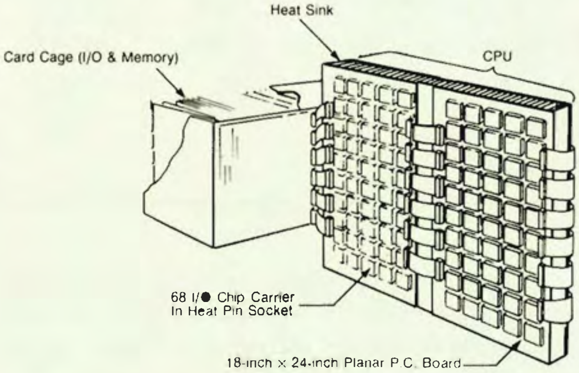

Heat Sink
Card Cage (I/O & Memory)
CPU
68 I/● Chip Carrier In Heat Pin Socket
18-inch x 24-inch Planar P.C. Board

Figure 2 Heat Pin Planar Packaging

exchanger on each module, with the module plugged into a backplane in the usual fashion. A novel planar approach was also considered in which all the CPU modules would be mounted on two sides of a large air heat exchanger. As shown in Figure 2, each plane contains several modules interconnected by flex circuitry, which also connects from one side of the plane to the other. This approach provided access to all of the components without disturbing interconnect or cooling.

Based on the weighting of the various parameters in Table 2 plus other program considerations, we proceeded with the individual heat sinks and the standard module-to-backplane configuration.

Regardless of the configuration selected, the cooling system had to deliver sufficient cooling air while conforming to the European noise reduction standards. To meet these needs, we devised a single-motor, four-wheel blower system to circulate the necessary air volume at appropriate pressure. An acoustic damping treatment applied to the enclosure doors reduced the noise emissions to an acceptable level. This packaging design not only met the acoustic noise regulations, but also yielded a much quieter machine than any previous Digital computer of this size.

Device Packaging

To meet the objective of on-site replacement of LSI and ECL RAM devices, we decided to provide sockets for them. Unfortunately, the reliability of sockets for MCAs was not well established, so it was necessary to provide an alternative scheme to hard-mount them. A 68 I/O leadless chip carrier (LCC) met all of the requirements.[1, 2] Even soldered-on clips could be used if necessary in place of the sockets.

Since SSI/MSI and RAM devices were widely available only in DIP format, we decided to use that package type. Thus DIP sockets, several of which were already qualified in Digital, were used for RAM replacement.

This mixture of component types forced us to choose a through-hole solder assembly technique because Digital has no mixed soldering process (for surface-soldered and through-hole components on the same board). Therefore, both the socket (Figure 3, on the left) and the solder clips (Figure 4) for the MCA were designed in the through-hole configuration. To reduce the inductance, the socket has a parallel path for the device ground through the cover.

During the course of the project, two problems arose relative to mounting the MCAs. The first was that the solder clip had to be installed by hand. At about the same time, Motorola indicated that they would develop a pin grid array (PGA) package for the MCA. By working closely with the vendor, we obtained a package (Figure 3, on the right) that matched the electrical performance and footprint of the LCC socket, allowing the substitution of the PGA for the solder clip as our backup. The next issue that arose was that sockets for the MCAs would not be available at sufficient quality levels within an acceptable time frame. At that point we switched to the PGA as the primary packaging technique.

It was originally intended that the MCAs would themselves incorporate diagnostic hardware, but this feature was discovered to impair the yield. The solution to that problem–providing supplementary hardware for diagnostics–created another: getting maximum hardware into minimum space. The module partitioning was already solidly established by the time we learned of the need for supplementary hardware. Fortunately, a SIP design, mounted with 40-mil centerline chip carriers, enabled us to install the diagnostic hardware in the limited space available.

Module Packaging

The initial module choice was one similar to the printed wiring board used in the VAX-11/750 system. It was the right size for our partitioning and density needs. However, to

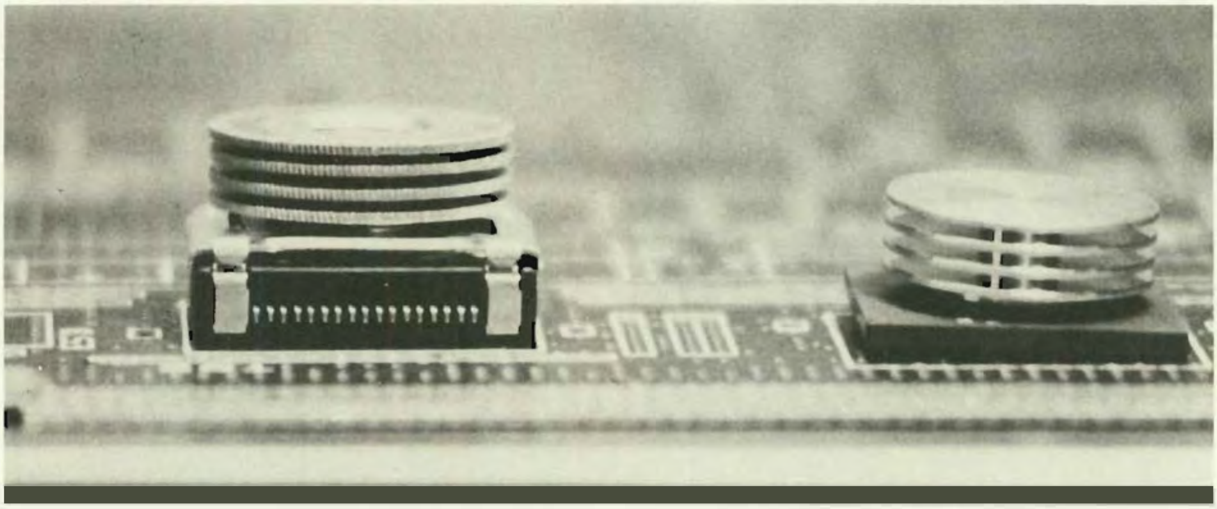


Figure 3 LCC with Socket and PGA Package

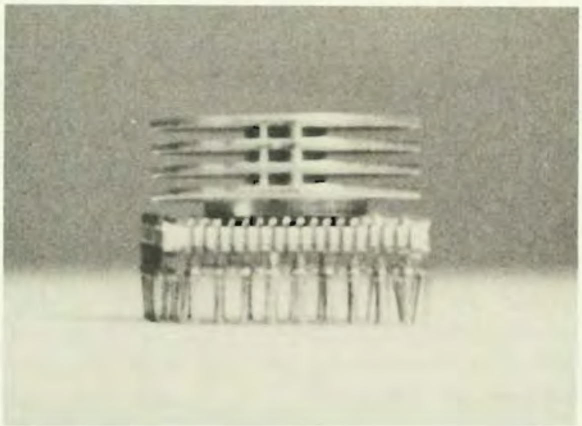


Figure 4   LCC with Solder Clips

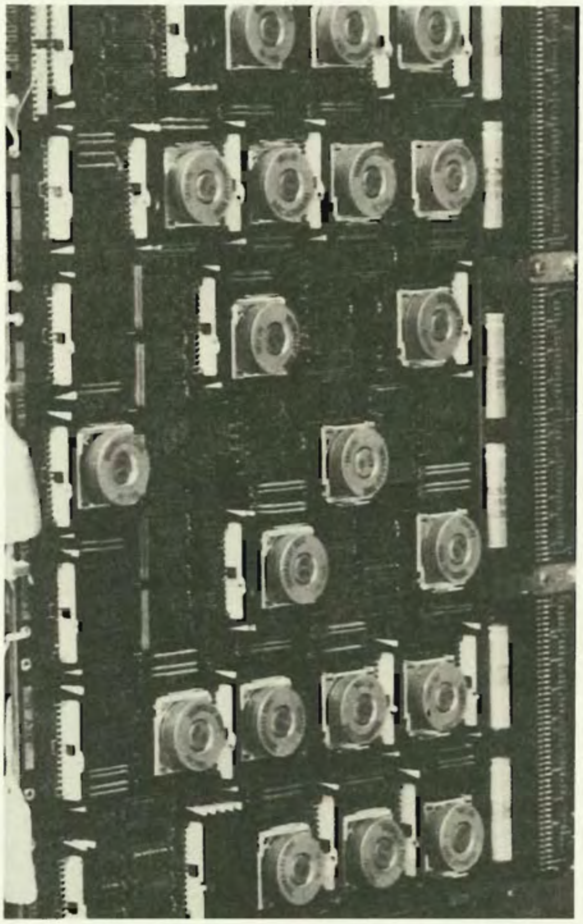


Figure 5   LSI Module

provide the maximum number of edge finger pins for signals, supplementary power and ground connectors were developed. In this way we could get signal pins sufficient for the logic that would be put on the board (some of the 282 pins are used for ground, but none for power). Figure 5 depicts the module, which is of controlled impedance construction and has eight layers, four of which are for signals. To ensure interconnect capacity, several trial layouts were done on early designs.

As the system design progressed, the number of gates needed to perform the required functions grew significantly, as demonstrated by Table 1. Eventually all spare slots were used and more were needed. But in some areas additional module crossings were unacceptable for reasons of system performance. So we decided to violate the rules for component density on the modules and added the extra gates to the modules already in place. Significant margins existed in power and cooling, but the interconnect was not adequate. We therefore had to add two signal layers to some of the modules. That posed a problem because, with traditional edge connectors, the extra signal layers had to be provided without any change in edge thickness. Two solutions to this problem were proposed. One involved a graduated layup in which the module itself would carry two more layers (a total of ten) while maintaining the eight-layer thickness at the connector. The other was an eight-layer construction with six, instead of four, layers for signal paths. When prototypes of each alternative were tested, the uniform eight-layer arrangement proved to be the satisfactory design, as it was easier to produce and less expensive.

Backplane

For the backplane we used a printed wiring board with the same routing grid and controlled impedance as the module boards. But the backplane has sixteen layers of which eight are for signal traces. To prevent problems due to Z-axis expansion during soldering, we used only compliant press-pin connectors. This also meant no drilling would be needed to add or delete nets because the press pins have wire-wrap tails for wire adds. Figure 6 shows a backplane assembly mounted inside the system enclosure. Also visible is the power distribution structure, which can provide up to 400 amperes of −5.2 volt current to the processor.

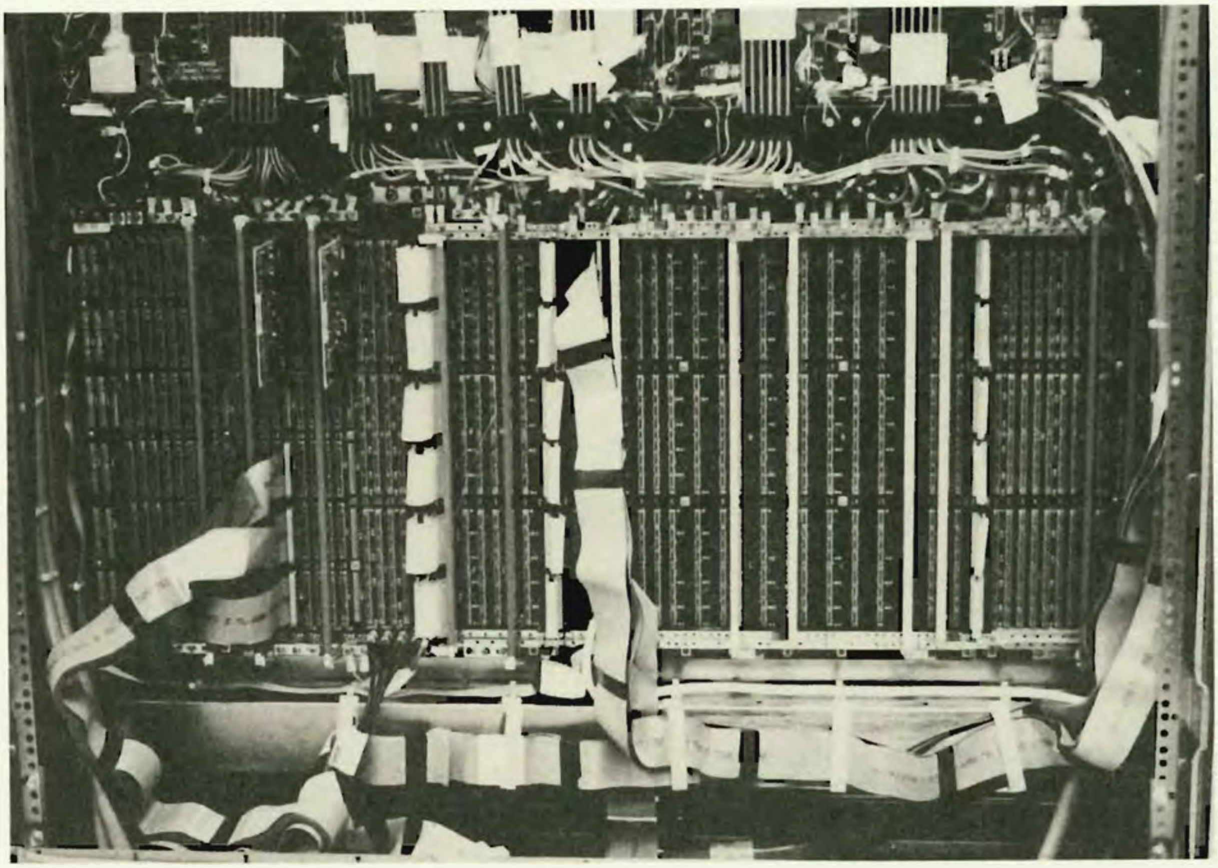


Figure 6 Backplane Assembly

Lessons for the Future

The experience of developing a physical design for the VAX 8600 processor demonstrated both the value of tools available to the package designer (e.g., the weighted comparison process and the thermal analysis software) and the need to improve those tools.

In particular, the events of the device-level packaging phase indicate the need for a design database approach offering numerous built-in test points or decision thresholds. This process allows earlier identification of problems, enabling engineers to switch from one strategy to another without disrupting the project schedule.

Similarly, the use of routing-prediction software derived from proven interconnect algorithms[3] reduces the incidence of routing inefficiencies. And it provides adequate safety margins in estimating gate and part counts at the beginning of a project.

Two other product development objectives were revealed as a result of VAX 8600 design efforts. The first is the need to ensure that connector technology is not dependent on module thickness. Then extra layers can be added without greatly affecting related hardware. Second, larger safety margins must be provided to reduce module routing difficulties.

Overall, the greatest need is for tools that provide accurate monitoring of design evolution as a whole, and also in the individual regions of development. This is especially true given the great increases in complexity from one project to the next. Many people are involved in building a sophisticated computer system like the 8600, and everyone must know what the others are doing.

As the industry continues to evolve and mature, it becomes essential that package designers communicate actively with system and logic designers, as well as manufacturing, marketing, and customer support personnel. The development of tools and systems that provide expanded insight into the progress of a whole project will assist packaging engineers in becoming creators of design processes as well as developers of hardware.

Acknowledgement

This paper presents the work of a large number of technology people within Digital. In many areas there were cooperative efforts among several Digital groups and our vendors. Without their hard work, these accomplishments would not have been possible.

References

1. D. I. Amey and J. W. Balde, "New Chip Carrier Concepts will Impact LSI-based Designs," *EDN*, vol. 23, no. 17 (September 20, 1978): 119–126.

2. B. Weaver and R. Moore, "Electrical and Mechanical Considerations in the Design of a Leadless Type A Chip Carrier for High Performance Applications," *Proceedings, IEPS*, 1st Annual Conference (1981): 16–24.

3. D. Schmidt, "Circuit Pack Parameter Estimation Using Rent's Rule," *IEEE Transactions on Computer Aided Design*, vol. CAD-1, no. 4 (October 1982): 186–192.

John H. Hackenberg

Signal Integrity in the VAX 8600 System

Maintaining signal integrity in ECL is necessary for fast execution speeds. ●n the VAX 8600 project, software tools were developed to eliminate signal problems before hardware was constructed. The number of signal layers was determined by modeling the components and routing channels. The worst-case noise margins were set on the basis of noise immunity. Power distribution can affect the margins, so special care was taken to limit transients. Temperature changes, which also cause signal level shifts, had to be limited. Waveforms and their reflections were modeled to identify the transient response. Another model identified crosstalk problems in parallel runs.

To achieve the performance goals set for the VAX 8600 CPU, emitter-coupled logic (ECL) was chosen for implementing the design. This consists principally of custom macrocell arrays (MCAs), and standard series 10K logic and RAMs. The challenges and problems that utilization of this technology presented were investigated by studying an earlier ECL design at Digital. This investigation resulted in the allocation of signal noise margins and the recognition of the need for new software tools for noise summation, and reflection and crosstalk analysis. As the design of the machine progressed and problems were encountered, we improved the new software to analyze whole networks and to allow as much flexibility as possible without risking the time to market.

Printed Wiring Board Characteristics

The first tasks were to select the characteristics of the printed wiring board (module) to be used and to determine the number of components that could be interconnected on it. The characteristics chosen were the following:

1. The board will be the same height and width as that in the VAX-11/750 and VAX-11/780 systems.
2. Board thickness will be limited by the card edge connector chosen for the system.
3. The routing grid will be 50 mils to guarantee a maximum of 5 percent backward crosstalk.
4. Interconnect impedance will be maintained at 55 ±5 ohms.

Items 3 and 4 also apply to the printed wiring backplane that carries the signals between the modules. The minimum desirable impedance is 50 ohms to match the minimum output drive capability of the MCAs (the MCA 25-ohm drivers are strictly for double-ended buses, where the lines in each direction are 50 ohms). The higher the impedance, the thicker the dielectric must be for a given signal conductor cross-sectional area. And the thicker the dielectric, the fewer layers that can be incorporated into a board of the maximum thickness (180 mils). Thus 55 ±5 ohms fits the requirements neatly, and within this constraint the backplane actually reached the limit in number of layers.

The number of components is obviously limited by the available space–the area of the board. But it also depends on the number of interconnections that can be made among those components. In investigating this issue, special consideration was given to signal IR drops due to interconnect length, as the voltage drop along a conductor directly subtracts from the noise margin at the input to the receiving gate. To solve this problem, different line widths were used in the different signal layers of each board. Signals could then be assigned to particular layers depending on the length of the signal path. Thus, longer lines could be assigned to wider signal traces to equalize the IR drops.

With this information and the early component estimates from the logic designers, we determined the number of components on each board and how many signal layers would be needed to interconnect them. Then, from specifications of the amount of power consumed by each component, the total power drawn by each board and by the entire CPU were estimated. In turn, these estimates allowed us to determine the thickness of the copper in each module and in the backplane. At this point making layups of the hardware could begin. The result was that different modules in the CPU vary from two to six signal layers, and the CPU backplane has eight signal layers.

Although this early analysis was useful, in the actual layout of the modules we ran into board routing problems. To solve them, a program was written based on Schmidt's article on estimation of circuit pack parameters using Rent's Rule[1]. As input the program requires the number of components of each type on a board, the number of signal pins on each component type, the size of the board, and the number of routing channels between adjacent component pins. From this information, the program determines the number of signal layers required to route the board.

We also created new programs to obtain better correlation between calculated printed wiring impedances and measured impedance values (in other words, to obtain better prediction). These so-called "field" programs employ electromagnetic theory to simulate the inductance, capacitance, and resistance of conductors of arbitrary size and shape in two and three dimensions. From these simulated characteristics, the programs compute the electrical parameters for microstrip and stripline configurations, and the crosstalk between conductors. The three-dimensional program also computes the crossover capacitance of signal conductors that are on adjacent layers and routed orthogonally to each other. This last computation is important because the crossover capacitance increases the propagation delay of signal traces and lowers their impedance. Additional enhancements are being planned for these programs to better analyze signal reference planes from an alternating current (ac) viewpoint.

Noise Margins

To design a reliable system, it is necessary to understand the direct current (dc) noise margin for the ECL gates being used. Different logic families have different characteristics in the way tracking rates of input and output levels depend on variations in temperature and supply voltage. These variables were used to determine the worst-case dc noise margins, depicted in Figure 1. However, if a system were to be designed around worst-case dc noise margins, then all the noise contributions summed together could not exceed those margins. This would be far more restrictive than necessary for system integrity and would be devastating for system performance. That is, the gates would have to be so far apart that the interconnect delay between them, on which system cycle time depends, would be unacceptable.

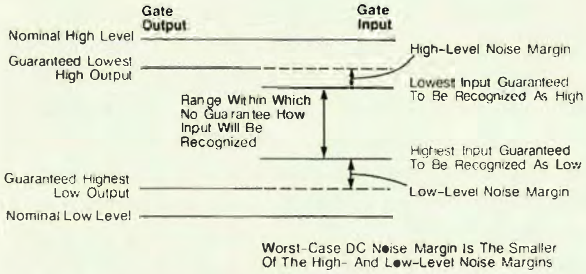

Gate Output
Gate Input
Nominal High Level
Guaranteed Lowest High Output
High-Level Noise Margin
Lowest Input Guaranteed To Be Recognized As High
Range Within Which No Guarantee How Input Will Be Recognized
Highest Input Guaranteed To Be Recognized As Low
Guaranteed Highest Low Output
Low-Level Noise Margin
Nominal Low Level
Worst-Case DC Noise Margin Is The Smaller Of The High- And Low-Level Noise Margins

Figure 1 DC Noise Margins

On the other hand, by understanding the dc noise margin for a given gate, one can also obtain its ac noise margin. In particular, for each gate one can derive an input-signal ac noise immunity curve (Figure 2), which shows what amplitude of input noise is required to switch the gate output at any noise pulse width. Based on this relationship, if the sum of all input noise contributors for each gate in the system is less than the noise required to switch

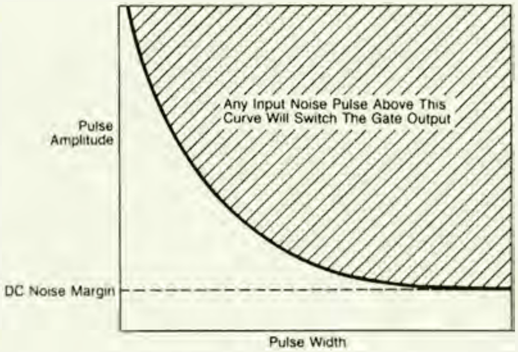


Figure 2 AC Noise Immunity

the output, then the integrity of the system can be guaranteed. This criterion is much less restrictive than dc noise margins; in other words, any point below the curve of Figure 2 is acceptable. Therefore, ac noise immunity was used to set the worst-case noise margins for the ECL logic in the 8600.

Based on all known noise contributors, we determined the ac noise margin for the system. To set up the design rules, we then assigned an amplitu de to each noise contributor; that is, the noise was allocated among the various sources, as shown in Table 1. This allocation allowed us to define a routing grid on the printed wiring boards and backplane, and to select connectors and transmission line cables.

Table 1 Noise Budget

Noise Contributor	Allocation in Millivolts
Load reflections	100
Crosstalk	100
Interconnect mismatch impedance	100
Simultaneous switching of outputs	150
−2.0 Vac noise on signal line	25
Signal IR drop	25
V_{CC} IR drop	14
Gate feed-through	50
Output voltage adjustment to thermal variations	6

Finally, we wrote a program to sum all noise contributions (worst case but without taking signal timing into consideration) for each ECL network in the system. Those networks identified as potential problems were analyzed by hand using timing information to determine the impact on the system. Real problems were resolved by reducing one or more of the noise contributions (such as crosstalk from adjacent signal traces) or by spacing loads farther apart on the transmission line to reduce the amplitude of load reflections.

Controlling Noise Sources

The largest allocation in the noise budget is the one that little can be done about: the simultaneous switching of outputs, which generates 150 millivolts of noise. Other sources could be just as noisy, but the allocations for them reflect the fact that action can be taken to reduce them. Besides the use of different width traces to equalize signal IR drops, the major efforts lay in power distribution, load reflections and crosstalk.

Power Distribution

Power distribution is an especially important factor in designing an LSI system with ECL. Supply regulation is implemented through remote sense points located near the logic circuits. But the number of such points is necessarily limited, and an excessive supply drop between a sense point and any ECL gate would adversely affect the noise margin. Of course, there must also be sufficient decoupling of the supply voltages.

To obtain a reasonable dc noise margin on the ECL gates, a goal was set that all factors contributing to variation in the supply voltage at any point in the distribution could cause no more than ±3 percent variation in the nominal V_{EE} voltage. Table 2 lists these factors and the allowable variation in each.

ECL gates wired together are particularly sensitive to V_{CC} voltage differences because the reference for both output and input thresholds is itself referenced to V_{CC}. Furthermore, any ac noise on a V_{CC} line not common to both gates may reduce the noise immunity. To minimize V_{CC} differences and equalize ac effects, full ground reference planes were used in both the modules and the backplane. These planes keep the inductance in the V_{CC} path between chips as low as possible.

To reduce the total power required by the system, we employed a smaller supply voltage, −2.0V, for the terminators. This allowed us to use a terminating resistance that matched the line impedance better. But it also created the possibility of large changes in terminator current over an entire module, a situation that would produce large transient voltages. Any noise in the terminator voltage is coupled in part onto the signal wires. To reduce these transients, decoupling capacitors for both high and low frequencies were distributed throughout the modules. The specification is enough decoupling to limit transients to 50 millivolts on V_{EE} and V_{TT}. Table 2 also shows the allowable variations in the factors affecting the terminator supply.

Table 2 Power Supply Variation

	Variation in	
Factor	V_{EE} −5.2V	V_{TT} −2.0V
Regulator tolerances Line/load regulation, ripple, long-term change in dc regulator output	1.0%	2.0%
Noise transients due to load current changes	1.0%	2.5%
Distribution IR drops	1.0%	1.0%

Thermal Considerations

The signal output and input levels of circuits shift with changes in temperature. To hold the dc noise contribution from this factor within its allocation required limiting to 10°C the air temperature difference between any two devices connected together through any network. The thermal engineers attempted to guarantee adherence to this criterion by holding the temperature rise across every individual module to 10°C. Since the heat generated by the different modules varies considerably, this goal turned out to be unattainable. But a thermal analysis of every network, including those that extended over multiple modules via the backplane, showed that the fundamental requirement relative to any two devices in any network was met.

Load Reflection Analysis

To analyze load reflections, we created a simulator that models a transmission line in the time domain. This program is specifically for ECL circuitry, and it gives results similar to those of SPICE[2] but takes much less CPU time. To model a waveform at any point on a line, the simulator divides the total delay into many increments and calculates a set of values for the waveform corresponding to those increments. The calculations take into consideration the (a) impedance and propagation delay of the line, (b) input and output impedances for each gate, (c) package capacitances, and (d) electrical parameters of signal connectors. Besides the set of values representing the generated waveform propagating along the line, the program also calculates a second set representing the reflected waveform. In a manner analogous to the result of a waveform and its reflection on the line, the corresponding values in the sets are summed. With this technique for waveform analysis, we can determine the transient response for each output and input on arbitrary networks. Using the appropriate differential equations to represent source and load models gives results that are comparable to those given by SPICE.

Once a good correlation was obtained between bench measurements and simulations, we added algorithms to calculate the minimum and maximum propagation delays along each ECL network in the system. When the gate delays, interconnect delays, and appropriate logic conditions were established, we could analyze the timing of the VAX 8600 CPU using worst-case parameters. These parameters included both minimum and maximum values for gate delays, output rise and fall times, interconnect delays, and impedances of the interconnect for each logic path in the CPU. The program that calculates interconnect delay can also analyze networks containing multiple sources (i.e., wire-ORs and buses).

Crosstalk and Interconnect

As boards become denser and switching speeds faster, crosstalk becomes an increasingly important source of noise. The program for calculating crosstalk, which can be used for TTL and ECL, finds all parallel pieces of signal etch on a board. It then calculates the crosstalk contribution to each victim segment from all parallel aggressor signal runs, within reasonable limits–it ignores those too far away. The calculations are based on the length and separation of parallel runs using crosstalk coupling coefficients rather than transmission line simulation. The voltages for each run are added and reported as the total crosstalk voltage coupled into the victim network. If this total exceeds a specified threshold, the report includes a breakdown of the crosstalk for each run.

Printed wiring that handles the signals between integrated circuits on boards and backplanes must be controlled impedance to obtain the best system performance. To meet the goals of the VAX 8600 system, at each interconnection we permitted no more than 100 millivolts of reflection due to mismatches in impedance as a signal moves from one interconnect to another.

Summary

The initial performance goal for the design of the 8600 was a program execution speed at least four times that of the 11/780. One of the factors that made possible the realization of this goal was an investigation of the interconnect environment for the ECL logic used in the 8600. In doing so we gained a significant understanding of and control over the following parameters affecting the integrity of the logic signals in the system:

1. Propagation delay per unit length of line
2. Voltage drops from the source to each load
3. Crosstalk between parallel signal lines
4. Reflections due to loads on a transmission line
5. Reflections due to mismatched impedance characteristics of the line
6. Reflections due to connector impedance
7. Reflections due to mismatch between interconnect impedance and the terminator

This understanding and control allowed us to perform accurate simulations of the interconnect delays through all paths in the CPU, resulting in the elimination of a large number of potential problems. Accurate timing simulations of the interconnect allowed the resolution of logic delay problems before committing the design to hardware, significantly reducing design turnaround times.

Many people inside Digital worked diligently to generate programs and build test hardware to analyze the interconnect. Because of this, we were able to reach our goal of building a system with the caliber of the 8600.

References and Notes

1. D. Schmidt, "Circuit Pack Parameter Estimation Using Rent's Rule," *IEEE Transactions on Computer Aided Design,* vol. CAD-1, no. 4 (October 1982): 186–192.

2. SPICE is a general purpose circuit simulation program for nonlinear dc, nonlinear transient and linear ac analyses. It was developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

E. Brian Kalita
William English

Cooling the VAX 8600 Processor

Proper cooling is essential for reliability yet is constrained by acoustic requirements. Both are achieved here using a single centrifugal blower to move air through the cabinet, with modules spaced for suitable air flow. Thermal models were created to analyze temperature gradients on modules and across networks, thus guaranteeing the integrity of signal levels. Component temperatures received special attention since an MCA can dissipate five watts and thus needs a heat sink. The best heat-sink design was developed by measuring die temperatures using testing devices, each containing a free diode.

The VAX 8600 processor dissipates six kilowatts of energy, nearly all of it from one double-width cabinet. Since the functionality of the logic is temperature sensitive, cooling was a major concern in building a reliable system. Nevertheless, the 8600 runs (and was fully qualified) on a solid floor using computer-room air for cooling. Of course the system can also be cooled by conditioned air drawn through a raised floor. Much of our cooling design effort was aimed at satisfying acoustic goals while at the same time meeting cooling requirements. The 8600 is the quietest machine of its size that Digital has ever built.

Overall cooling of the 8600 is accomplished by the movement of air from bottom to top. Air at normal computer room temperature enters the cabinet through a perforated base panel and passes through an air-filter assembly that doubles as the UL drip screen. Should there ever be a fire inside the cabinet, the screen will extinguish the flames of any burning material that may drip from the equipment. From the screen the air passes through the card cage containing the logic and then through the power supplies. At the top of the cabinet is a double-dual centrifugal blower (i.e., a single device with a pair of wheels on each side). The blower pulls the air up through the cabinet and forces it out through a pair of acoustic mufflers mounted inside the rear cabinet doors. Mounting the mufflers as an integral part of the rear doors allows easy access to the logic and power backplanes. The mufflers have an expanding internal cross section to regain as much static pressure as possible from the high-velocity air exiting the blowers. The muffler entrance and the exhaust louver pattern, respectively, are tuned to reduce inlet pressure losses and exhaust recirculation. The entire path is closed and independent of the outer walls of the cabinet. Opening the cabinet doors does not impair the effectiveness of the cooling system.

The card cage is made up of four sections, as shown in Figure 1. From left to right, as viewed from the front, there are the memory, CPU, adapter bus, and I/O adapters and controllers that connect to the peripheral equipment. The memory and I/O sections have standard Digital 0.5-inch slot spacing. Spacing in the adapter section and some CPU slots is 0.6 inch. The remaining CPU slots have 1.0-inch spacing to provide the necessary component clearance and volume of cooling air flow for those modules containing macrocell arrays (MCAs). The greater clearance is required because each MCA must have an individual heat sink, and the high-powered MCAs require a greater volume of air for cooling.

Any VAX 8600 processor may have a number of empty module slots that can otherwise be used for various options, such as the floating

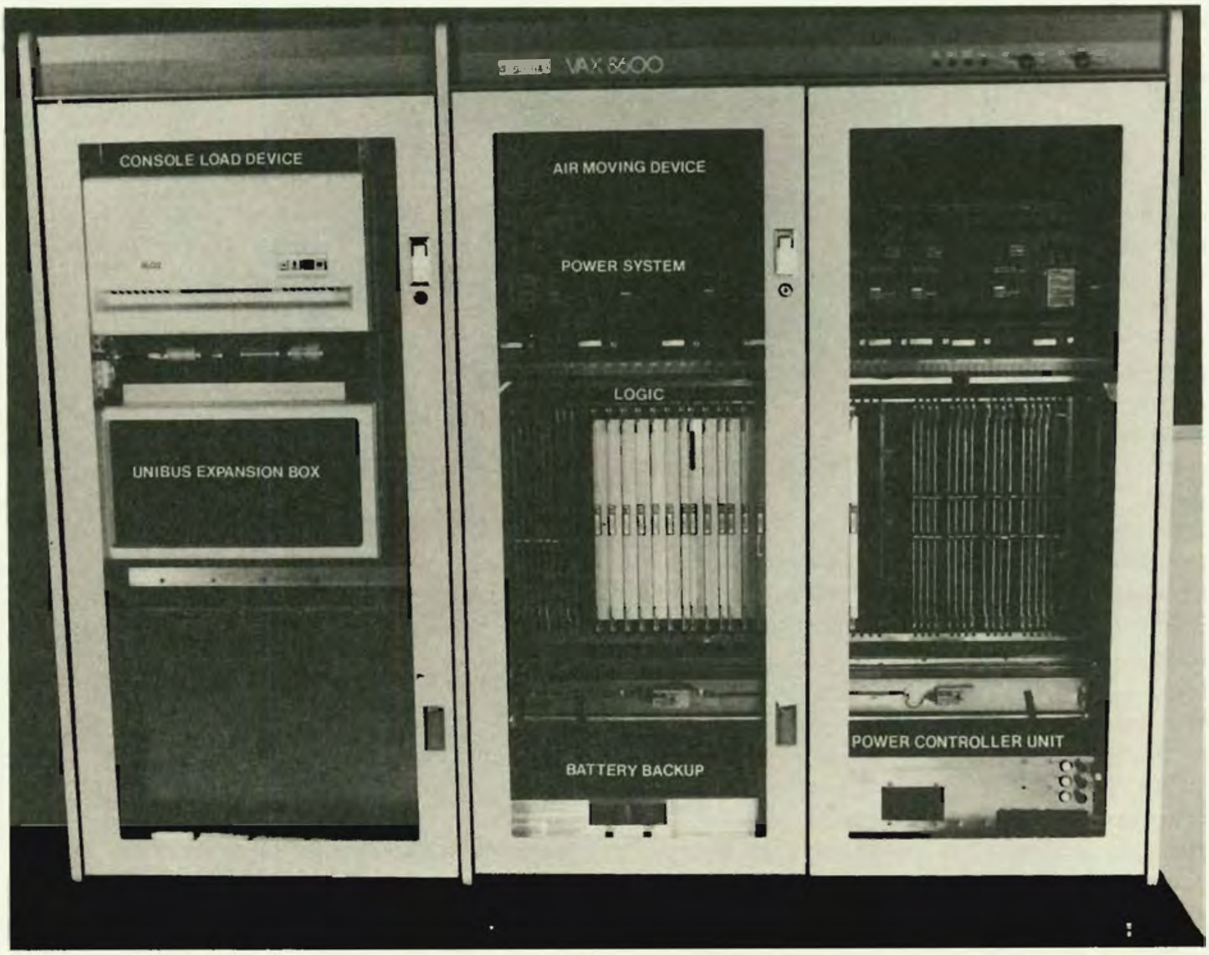


Figure 1 Card Cage

point accelerator, additional memory, a second connection to the adapter bus, and various I/O options. To prevent the cooling air from taking the path of least resistance through the gaps, plastic pseudo-boards are installed in all unused slots. Eliminating the gaps keeps the air flow close to the boards where it belongs–cooling the components–and also serves to make the air flow characteristics of all machines the same.

Module Thermal Design

The thermal design is predicated on two criteria related to temperature. The first is that signal levels in the emitter-coupled logic (ECL) components are shifted by changes in temperature. Limiting the temperature difference between any two components within a network to 10°C prevents the logic levels from shifting out of range at one component relative to another. The second is that component failures are proportional to temperature. Holding down the component junction (die) temperature yields higher reliability–a longer MTBF.

Early in the project we decided on two general goals to guide us in designing the 8600's cooling system to satisfy those criteria. One was to maintain a nominal air temperature rise across any given module at about 10°C. This would guarantee a maximum 10°C ambient difference between any two components on the module. The other was to guarantee that at least 90 percent of all die temperatures would be less than 100°C, even at the maximum ambient temperature of 32°C. Of course there were bound to be differences from one module to another; one module actually dissipates 180 watts, and the one next to it dissipates 146.

In those situations where the 10°C rise was exceeded, we analyzed the individual networks to determine the temperature gradients within them. On one board there was actually a 15°C rise, but no individual network exceeded 10°C; the goal for junction temperatures was met as well. By taking great care in the placement of components, we were able to configure the individual networks in such a way that even though we violated the general rule on temperature rise in some cases, we nonetheless always stayed within the critical limit on the temperature difference between two devices wired together.

To help the logic designers, we set up design rules aimed at satisfying the thermal requirements. To start, one rule was based on dividing a board into sections about two inches wide (approximately a single column of components) and three inches high. The rule required that the components contained within each such section should not exceed a given maximum power. The values for maximum power and section size were based on preliminary tests using a mockup board with prototype MCAs. With rising air flow, each component heats the one above it; we could not therefore allow the placement of a column of hot components, even if the rest of the board were cool, without evaluating each individual case. For example, to make routing possible, a designer may have needed to violate the section-power rule and put three five-watt devices right on top of each other. Cases such as this would be evaluated by considering the network and die temperature information.

At the next level of refinement, we used a thermal analysis tool designed by the Thermal Engineering Group at Digital in Maynard, Massachusetts. This tool utilizes different functions of thermal resistance versus air velocity to calculate junction temperatures for different kinds of component packages. Based principally on vendor data, these functions were developed for plastic packages, ceramic packages, and packages with special thermal characteristics. Within each package type, separate curves were derived for different sizes, correlated to the number of pins. To perform the analysis, we divided the board into as many as one hundred sections. The "model board" was then populated for a particular configuration by our specifying the components and assigning them to the sections. The analyzer first calculates the temperature rise of the air from environmental information, power data, and component placement. From this calculated temperature rise, the appropriate thermal functions by component type and size, and air velocity information, the program predicts the junction temperatures.

Although the analysis was extremely valuable, it was also very cumbersome to use. All the information on component type, size, power and position had to be entered by hand. A person required nearly a week to enter the data for one board. Once the value of the analysis had been demonstrated, we modified the program to take the component data from files supplied by the CAD tools that were already in place (drawing program and wirelister). Furthermore, a set of algorithms and software was also developed that performed the section assignment automatically from layout data provided by the component placement optimization software. Eventually the handwork was reduced to five minutes, the time it took to select the number of sections and specify the input files.

We also developed a network analyzer. Using this tool in conjunction with the module thermal analyzer allowed the inspection of the junction temperatures throughout an individual network on a board to determine whether the 10°C rule was violated between any two components within that network. We used the module analyzer and the network analyzer on every board in the VAX 8600 processor.

Once the logic design started to stabilize, we expanded the network analyzer to investigate individual networks that ran through multiple modules across the backplane. With the huge number of logic interconnections, this task was immense and complicated, but we did manage to complete it. Thus in the long run, thermal modeling was done at the device and board levels, and on the total machine.

Component Thermal Design

Most of the time and effort in component thermal design was devoted to the MCA. This was because each MCA package can dissipate up to five watts. We tried many approaches relative to heat sinks and packages, with and without sockets, before settling on the final designs. We eventually arrived at a solid socket design for an MCA in a leadless chip carrier, but the sockets themselves were finally dropped (in favor of a pin grid array package) because of insufficient availability.

The major part of the testing was done to determine what heat sink to use based on the requirements of die temperature and allowable component-to-component temperature differentials. The die temperature is equal to the product of the power and the thermal resistance for a package plus the ambient temperature. The vendor specification for the thermal resistance of the MCA is 10° ±2°C per watt. The LSI circuit is near the surface of the silicon. The major thermal path for the package extends (a) from the circuit; (b) through the silicon; (c) through the die bond, which is a gold-silicon eutectic solder; (d) through the alumina chip carrier; (e) through the epoxy that bonds the carrier to the heat sink; and (f) through the heat sink into the air. Other paths to the air include heat convection from the surface of the ceramic and conduction through the leads into the board.

Within these constraints, we had to select the heat sink and the epoxy to attach it. But in order to make these decisions, some way of actually measuring the die temperature was needed. The most promising technique seemed to be the traditional one of using a free diode as an internal thermometer. With constant current, a diode has a negative voltage/temperature curve that is linear over small ranges. Since the ranges of concern were small, if there was a free diode on the device, we could calibrate it in a bath and then use its leads to monitor the die temperature.

Unfortunately, an ECL device under power does not have any free diodes, so the vendor produced a special die just for thermal testing. This die was somewhat different physically from the MCA die, and it contained only TTL circuitry for making thermal measurements. Digital and the other companies using MCAs worked together to calibrate the die and use it to measure temperatures. At first we had considerable problems with instrumentation, learning what to do and how to do it, and getting good dies from which reliable measurements could be gained. We built our own test equipment and developed procedures that allowed us to "look inside" the MCA packages. The success of this effort enabled us to select heat sinks that maintain the MCA temperatures at the desired levels.

We decided to continue our testing on a device that better approximated the MCA in both size and structure. For this purpose, Digital's LSI facility in Hudson, Massachusetts provided two types of thermal test elements. The first incorporated the TTL die of a gate array used in the VAX-11/750 system. This device allowed us access to a free diode and was close to the right size for the MCA. It was mounted in the ceramic carrier of the MCA and allowed us to get a close thermal approximation to an 8600 MCA package. Later the Hudson plant created an actual MCA on which they placed a "free" non-ECL diode just for thermal testing. The diode is not used in the logic of the device, and in normal production, it is not bonded to the I/O pads of the chip carrier. Whenever packages are required for thermal testing, the diode leads are bonded in place of two of the MCA output connections. This process renders the package useless for any other function, but perfect for thermal testing, since it is the actual structure of interest–an MCA die–and dissipates the actual power of the devices used in the 8600. With this "real" MCA package, we verified our thermal design by building a module with these parts in place of the actual MCAs. This "thermal module" can be placed in a machine and powered as if it were actually functional. The MCA packages containing the special die can be monitored, allowing us to watch what really happens inside the machine.

The experiments with the test devices also enabled us to investigate die bonding, or wetting. We wanted to know how much of the piece of silicon was actually soldered to the ceramic. The result of these studies enabled us to establish the specification for a test procedure that inspects the temperature of the die after it has been powered for a specified number of seconds. If the die bond is poor, the heat will have to travel through a small void rather than through the higher conductivity solder; the die temperature will therefore be higher than a specified acceptable level.

Switch to the Pin Grid Array Package

Fairly late in the project, an acceptable pin grid array (PGA) package became available, and we decided that its advantages warranted using it. This meant that all the thermal investigations had to be repeated to verify that the PGA configuration met the goals.

Removing the sockets shortens the packages, so there is more space for air flow between those boards that already have the larger, one-inch spacing. It was feared that the new package configurations might actually run too much cooler. We already had a fairly solid logic design that worked in the thermal configuration then existing; a significant temperature shift in either direction was undesirable. Running hotter reduces reliability; running significantly cooler, although it improves reliability, might affect the signal levels to such an extent that the system would not work at all.

We studied the temperatures with the thermal module in every slot. Then we experimented extensively with a particular slot that was warmer than the others (the air flow is not exactly the same through all slots). The result of the investigation is a package in which the MCA runs slightly cooler than before, but still well within the signal level requirements. The heat sink is a single, four-finned unit, one inch in diameter. It is bonded to the top of the PGA package with an epoxy, and the whole assembly process is fully automated.

Summary

To cool a machine as large and as dense as the VAX 8600 processor requires the continuous movement of a very large volume of air. To do it with air at room temperature and go about it quietly is a significant feat indeed. It was accomplished by exercising meticulous care in the physical configuration of the system and by the creation of imaginative and thorough tools for thermal analysis.

William F. Bruckert
Ronald E. Josephson

Designing Reliability into the VAX 8600 System

The failure rate of a system is directly related to the number of components used in its design. Therefore, the designers of a large CPU must put emphasis on fault avoidance, fault tolerance, and fault minimization to ensure that the overall system failure rate is acceptable. The VAX 8600 system contains many features to assure its reliability. Conventional approaches, like parity checking, and nonconventional ones, like array address checking through ECC codes, were used to overcome the higher failure rate generated by having more components. This paper covers the most important steps taken to provide that reliability.

The cost of a failure is proportional to the size of a system, since more compute power is lost and more people are idled as size increases. Since the failure rate is directly related to the number of components in the system, a much greater emphasis must be placed on fault-tolerant designs in larger systems in order to keep the costs of failures at an acceptable level.[1] The VAX 8600 system is the largest, most powerful computer produced by Digital Equipment Corporation. We made customer satisfaction the most important engineering goal, thereby placing a high priority on the machine's reliability. In this paper, reliability is discussed from the customer's point of view, which covers a wider context than the usual definition of inherent reliability.

Computer reliability enhancement can be subdivided into four areas: fault avoidance, fault tolerance, fault minimization, and improved mean time to repair (MTTR). Fault avoidance is realized by reducing the system failure rate through improved quality of the components, interconnects, design, and manufacturing. Fault tolerance is the negation of the effects of faults through correction codes, redundant hardware, reconfiguration, and retry.[2] Fault minimization is the reduction of the effects of a fault by tagging corrupted data that has damaged the machine state or other data. Furthermore, fault minimization can be achieved by having the hardware give accurate and detailed fault information. The MTTR is improved through remote diagnosis, the reduction of the time to diagnose a fault, and the increase of diagnostic accuracy. The application of each of these four areas to the VAX 8600 design is discussed in detail in the following paragraphs.

Before these details are presented, however, a short explanation of the major parts of the 8600 architecture is warranted. The components in the VAX 8600 CPU are contained in four "boxes" that control operations and perform various functions. The E Box executes and retires instructions. The I Box prefetches and decodes instructions and prefetches operands. The M Box performs page translation, cache functions, I/O transfers, and memory array access. And the F Box performs floating point operations.

Fault Avoidance

Our first goal in designing a reliable system was to reduce the number of failures that occur in the machine. This involved getting components, interconnects, and power systems with the lowest failure rates. Reducing the failure rates also involved constantly monitoring the failures that were experienced and determining their causes.

A major influence on the IC reliability was exercised by specifying how the chips were to be stressed and tested. The DIPs and the macrocell arrays (MCAs) were required to be burned in before testing; thereafter, all chips were to be functionally tested. However, in debugging the early machines we discovered bad DIPs. We had expected to find only a handful of bad chips since they were all burned in. To identify the cause of these failures, all defective chips were analyzed. The problem was identified as static that was "zapping" our modules. Subsequently, the design was changed so that all machines come with static grounding straps.

We also examined the designs of previous CPUs to determine which problem areas were typical. The backplane is an example. Wire-wrapped backplanes are difficult to build and test. They have several failure modes–such as cold flow of the insulation, a nicked wire, and scraps of wire. They can also be damaged during servicing of the machine. All these problems often result in intermittent faults that slowly but surely become more solid. Improving the quality control on the wire-wrapping process to obtain the desired reliability was a very difficult task, since the process is comprised of a large number of repetitive but not identical operations. Moreover, a very small error rate still produces quite a large overall failure rate. Therefore, early in the project, we decided to replace the wire-wrapped backplane with a multilayer printed circuit card, which has a much lower failure rate.

In the power subsystem, fault avoidance was pursued by improving the alternating current (ac) input-power tolerance, the design testing, the manufacturing processes, and the environmental monitoring. In particular, manufacturing was a key area where the reliability of the power supplies was improved. A new power-supply tester was developed to improve our testing capabilities. It contains logic that can fully test the characteristics of a power supply and store the test data. The data includes line and load regulation and noise measurements.

A modular power supply (MPS) was designed to run from a single clock so that all regulators would be in synchronization. This synchronization allowed us to predict and control the output noise of the switching regulators. A new high-current connector that allows the regulators to be pluggable was also developed.

The power subsystem also contains the environmental monitoring module (EMM). The EMM was designed to monitor the status of the power supply and the environment inside the system. The EMM can measure the voltage output of every regulator, the inlet and outlet air temperatures, the air-flow velocity, and the ground-wire current in the primary power cord. The system protects itself by having the EMM monitor these conditions, log any deviations, and shut down the system if adverse conditions warrant it.

According to E.J. McCluskey, "Improper design of the hardware or software can result in a system which does not function at all. Such mistakes are, of course, quickly discovered and corrected. Other, less obvious design defects usually remain in any system even after it has been in service for a long time."[3] The results of design problems are logic circuits that either fail prematurely or sense signals falsely. The number of these types of errors is indirectly a measure of the quality of the tools used in the system's design.

At the beginning of a design project, rules are established to make sure that the goals for signal integrity and component failure rates can be achieved. It is usually impossible to develop rules that are both easy to check and at the same time don't overly constrain the design engineer. Often this results in complex rules. If they are inadvertently broken, the usual outcome is a decrease in the machine's reliability. The broken rules result in components that operate with excessive temperatures or signals that do not have adequate noise margins. A chip that runs too hot will fail sooner than anticipated; a signal that doesn't have adequate noise margin will sometimes be sensed incorrectly. Worse still is the fact that the component is blamed rather than the true cause, a violated rule.

As an example consider the operating temperature of an IC. There is a tradeoff between the maximum and minimum operating temperatures and the amount of noise margin available. If the temperature of an IC exceeds its maximum specified temperature, the amount of noise normally present from known sources, such as adjacent-run crosstalk, may be sufficient to produce a false signal. Therefore, it is important that all ICs stay within their specified operating temperatures. To ensure that, we developed a tool for use on the 8600 to check for chips that were getting too hot. If a chip was detected as being too hot, its layout was modified to correct the problem without changing the total power of the module.

A new timing analysis tool was also developed for the project. This tool enabled the designers to do a much more thorough job of timing analysis on this machine than had been done on previous projects. Using it involved running many separate programs that built a timing model of the machine from the schematics and the layouts of the modules, backplane, and MCAs. The results of the model were then used by a program that performed timing analysis of the design based upon a set of interbox timing specifications.

After the layouts of the modules were completed, every single run was analyzed to ensure that signal integrity had been achieved. The program computed the amount of noise generated from adjacent runs, reflections, and the like. Based on these results, we made a number of reroutings to increase the integrity of certain signals.

Fault Tolerance

All the efforts discussed in the previous section improved the machine's reliability. However, the logic could still fail; therefore, it was important to have mechanisms to recover from a logic fault whenever possible. Fault isolation and fault tolerance are highly correlated, not separate issues. Data integrity and retry operations depend on good fault detection. So does the ability to reconfigure the system when a fault occurs, a situation that requires accurate fault isolation as well.[4] It is important to know what type of fault was made and what processes may or may not have been affected by it. To accomplish fault isolation, we had to develop an effective fault detection and reporting scheme.

The design philosophy for the fault system had several major concepts. The first was that faults occurring synchronously with the program counter (PC) should be reported synchronously to it. Synchronous faults have a direct relationship to the current value of the program counter. For example, consider a write to an I/O register. Only one cycle is required for the M Box to accept all the information to perform the write operation. In the meantime, the E Box could continue processing instructions. The problem here is that if the I/O write has a fault, the current PC of the machine would have no fixed relationship to that fault, thus making recovery more difficult. To solve this problem, the microcode will stall the E Box on an I/O write until the confirmation of that write is received.

A similar problem exists with a translation buffer (TB) miss on a prefetch for the instruction buffer. If a branch is ahead of the TB miss in the instruction buffer and the branch is taken, the TB miss will not be a problem and should not be reported. In this case the design requires a delay in sending the TB miss signal to the E Box (which performs the memory management operations) until it attempts to execute the instruction whose prefetching caused the TB miss. In general, synchronous faults are reported via E Box microtraps.

Faults that are asynchronous to the program counter are reported asynchronously. Asynchronous faults are ones for which the value of the program counter has no definite relationship and which are usually reported through interrupts. Two examples of an asynchronous fault are a fault occurring on a disk write to memory and a parity error on a cache writeback operation.

At the time a fault is detected, it may not be known whether the fault should be reported synchronously or asynchronously. In that case, both fault-logging mechanisms are invoked: a microtrap for synchronous faults and an interrupt for asynchronous ones. Consider the case of a parity error on an instruction prefetch. If the E Box executes a branch prior to using the bad data, the synchronization will never be reached and the fault will be logged through an interrupt. In this case the microtrap condition will be cleared by the execution of the branch. If, however, the E Box attempts to execute the prefetched instruction with the parity error, an E Box microtrap will occur and the trap routine will clear the interrupt.

The second major concept used throughout the design was that hardware faults are considered to be process faults only if a process attempts to use or store corrupted data. For example, if corrupted data is detected during a writeback to memory from the cache, a fault will be logged. However, the process will not experience a fault until it attempts to either consume the corrupted data or store it on a disk. This logic imposes the requirement that corrupted data be marked for later detection, which is done with ECC code in memory. This subject is discussed in the Unique Reliability Features section.

Fault Minimization

When recovery is not possible, the next best thing is to control the amount of damage done by a fault. This tactic requires fault information that is accurate, relevant, and sufficient. Whenever a fault occurs, an error stack frame will be constructed by the E Box and placed in memory. The stack frame format is the same for all errors. We did not prejudge what would be useful in determining which information was relevant.

In the case of damaged data, fault reporting alone is not sufficient, since it is not possible to determine which process will access that data. Therefore, when data damage occurs, the logic marks it as "bad," and any future user of that data will be notified of that fact.

Mean Time to Repair

There are two kinds of machine failures: those having solid fault symptoms, and those having intermittent fault symptoms. Of the two, solid faults are easier to diagnose. To isolate solid faults, the console can examine the state of the signals that go from one module to another. Diagnostics are run to find the first failed test, which is then run in a single-step manner to look for the first incorrect signal. With the exception of multiple-source signals, the source of the first incorrect signal value is the failing module (since all of its inputs have been checked by this process). In this way faults can be isolated to the field replaceable unit.

Intermittent faults are much more difficult to diagnose, and they comprise between 80 percent and 90 percent of the faults. Diagnostics rarely provoke intermittent faults. But even when they do, the fault reporting can often be confusing. This confusion occurs because a logic fault will usually take place in a circuit after it has been tested and while another circuit is being tested.[5] The number of fault checkers in a machine affect its ability to know that a fault has occurred and to identify the failing unit. The probability of a fault occurring in the logic that any given checker has checked is not affected by whether the result is used or not. If an intermittent fault occurs on a path that is not being used, then no real fault has occurred. Therefore, the machine's overall reliability is increased by ensuring that fault checking is performed only on networks that are actually being used.

A detailed list of the checkers included in the VAX 8600 system is listed at the end of the paper.

If a failure occurs that requires immediate power shutdown, then remote diagnosis through the console cannot be used. This occurs when the regulators detect an overheating condition or the power for the EMM is out of tolerance. In these cases a magnetic indicator code that contains the failing regulator number will be displayed on the EMM module. This code enables a field service technician to know which regulator to replace.

Unique Reliability Features in the VAX 8600 CPU

In addition to the reliability features already discussed, the VAX 8600 design includes some not previously found on other Digital machines. These features are discussed under the four major areas used in the first part of this paper.

Fault Avoidance

The F Box executes self-diagnostics when it is not performing floating point instructions. These tests use "live" operands to enhance the detection of data-dependent faults. Both the E Box and the F Box are connected to a common source of instructions and operands. When the F Box detects that it cannot perform an operation, it will execute a diagnostic self-test. Exactly which self-test is performed depends upon the instruction. The number of machine cycles in the diagnostic routine is chosen to be equal to or less than the number of machine cycles used by the E Box. This ensures that the F Box will always be ready for the next floating point operation that will be passed to it. If a fault is detected, the F Box will be turned off, and the E Box will perform the instruction that would have been done by the F Box, only at a much slower speed.

Fault Tolerance

The 8600 supports instruction retry where possible. If a fault occurs that causes a microtrap during an instruction, a set of instruction retry flags will be passed along through the various fault recovery stages. The flags indicate whether or not the CPU has performed an operation that would make restarting the instruction impossible. An instruction retry would be inhibited if an abort bit is "on" for (a) an I/O read, (b) a memory write, (c) a state modify, or (d) the E Box. Otherwise, the instruction can be restarted.

The data cache can recover from single-bit errors. A cache data entry consists of 32 bits of data, 4 bits of byte parity, and 7 bits of ECC. The write of the check bits is pipelined and occurs in the cycle following the write of the data. The parity bits are used for fault detection and the ECC bits for error correction. The M Box always passes data to the E Box or I Box before any checking is done. If the data contains a parity error, then either the E Box or the I Box, as well as the M Box, will detect it. The M Box will then block the acceptance of any more requests and will execute a data correction sequence. The ECC code and the data are then sent to the array bus, and normal array-to-M Box data correction is applied. The "corrected word" is then written back into the cache. At some point the E Box will discover that it has been shipped bad data. The system will then retry the instruction if possible. The retry will be successful if the original fault was correctable.

An important goal of the power subsystem is to increase its tolerance of bad ac input power. The power input is a true three-phase input with very low neutral current. In previous designs the power-storage capacitors had been attached to the regulator outputs. The detection of power failures was performed by monitoring the ac line. In contrast, the VAX 8600 power system first converts power to 300 Vdc and then sends that power to regulators in order to produce the final output voltages. Power storage is done at the 300 Vdc level. This higher voltage allows more energy to be stored, since the storage is provided by capacitors. Power-failure detection is performed by monitoring the voltage level on the 300 Vdc power supply. When its voltage reaches the level at which there is just enough energy remaining to perform a power-fail sequence, then an ac power failure will be declared. This method allows continued operation regardless of the ac input waveform, as long as the machine receives sufficient energy, a fact that is especially helpful during brownout conditions.

Fault Minimization

The 8600 makes good use of the unassigned ECC codes (a 7-bit ECC can correct up to 57 bits of data). They are used to detect array addressing problems and to flag any corrupted data. When a memory write occurs (see Figure 1), the parity of the address and an indication of the quality of data are sent to the ECC generator. The quality of data is good if no faults were detected during its transmission to the M Box and bad if the machine suspects that

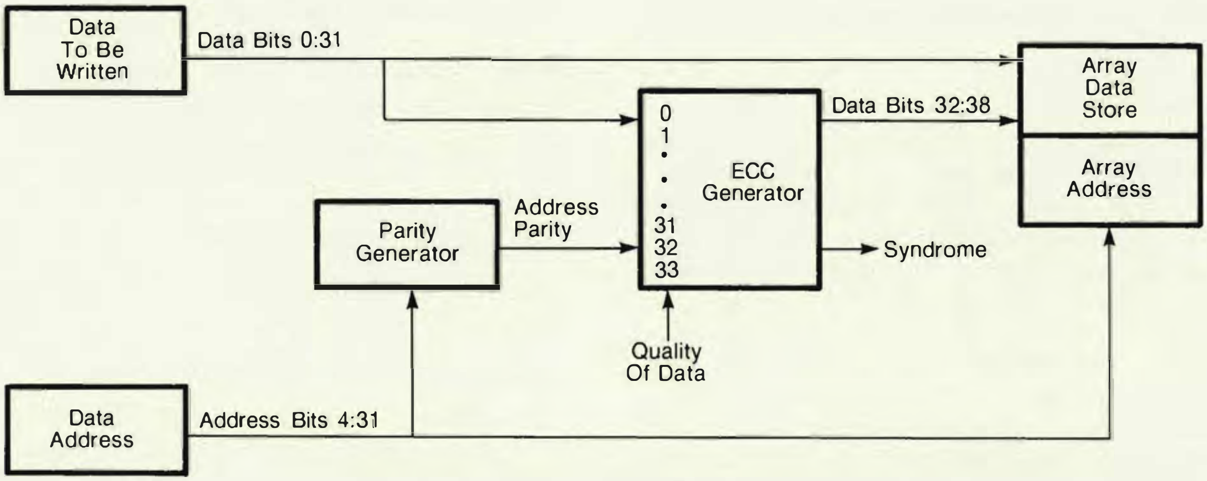


Figure 1 Array Address Checking in the VAX 8600 System

a fault is present. The address parity and quality information are inserted into the ECC generator by means of bits 32 and 33 of the data. Neither of these bits is stored in the array. When the data is read back, the computed address parity is sent along with a good-data signal to the ECC generator. If the computed syndrome is zero, the transaction is considered to be good. If the ECC generator decodes a single-bit error pointing to the address bit, then an address parity error will be declared. When that occurs, the word that was just received did not come from the address that it should have. Thus, the ECC generator can check the address lines from the M Box to the MOS array chips and detect the control faults that caused the M Box to access the wrong data word. If the chip thinks the quality bit needs correction, then the data word was faulty when it was received. The requester of this data will then be notified that the data is bad. If a normal single-bit error occurs on a data word that was stored with a code indicating bad quality, then the M Box will flag an ECC double-bit error.

Most of the internal buses in the VAX 8600 CPU as well as in the shifter and the arithmetic logic units (ALU) are parity checked. The ALUs are checked by triplication and parity checking the results. The I Box, F Box, and E Box each contain a set of general purpose registers (GPRs). When writes to the GPRs occur, all GPRs are written to simultaneously, thus keeping them consistent. If a GPR parity error is detected in one box, a recovery will be initiated that copies correct data from the equivalent GPR in another box to the failed GPR. Thus the machine can recover from GPR parity errors.

Mean Time to Repair

The number of microsequencers in the VAX 8600 system also adds to its reliability. Ordinary combinatorial control logic is difficult to check without duplication. Using a microsequencer is one method of building control logic that is easily checked. For example, all the microcontrol stores are parity checked. The M Box also checks the parity of the address, stack underflow and overflow, and stack address parity. Microparity errors are recoverable in the E Box, F Box, and I Box. These faults are not recoverable in the M Box since its state is modified in an unrecoverable manner before the parity computation is complete.

Summary

The task of making large machines reliable requires a continuous effort during all phases of the project, from conceptual design to manufacturing. In the future, machines will continue to get larger. Unless some major technology breakthrough that significantly changes the reliability of components occurs–as occurred when transistors replaced tubes–the fault-handling capability designed into large systems must be improved. This improvement is needed to overcome the inherently higher failure rate that comes with having more components. Based on this conclusion, we created many design processes, manufacturing processes, and fault handling features that increased the reliability of the VAX 8600 system. Careful monitoring and simulation were required to ensure that true gains in reliability were actually achieved.

Fault Checkers in the VAX 8600 System

In the E Box

ALU Output Parity Check

Shifter Parity Check

Microcode Parity Check per Board

Other RAM Store Check with Separate Error Flags

AMUX Parity Check

BMUX Parity Check

GPR Copy Write Recovery

Instruction Retry

Diagnostic Fault Insertion

In the M Box

Memory Address Parity Check

ECC on Cache and MOS Memory Data

Writeback on SBE

Microword Parity Check

Microaddress Parity Check

Microstack Parity Check

Microstack Underflow/Overflow Detect

A Bus Parity Check

Array Bus Parity Check

Corrupted Data Tag

CPR Parity Check

In the F Box

FBM Microword Parity Check

FBA Microword Parity Check

FDRAM Parity Check

GPRs Parity Check

Self-test (when not executing instructions)

In the I Box

Microword Parity Check

Ibuffer Parity Check

DRAM Parity Check

GPR Parity Check

OP Bus Parity Check

W Bus Parity Check

IMD Parity Check

References

1. D. P. Siewiorek and R. S. Swarz, *The Theory and Practice of Reliable System Design* (Bedford: Digital Press, 1982).

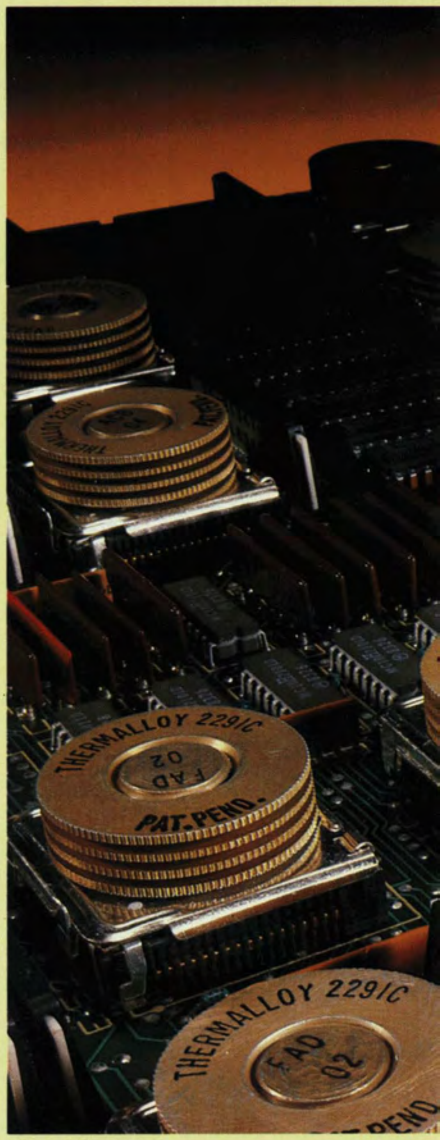
2. L. S. Rosenthal, "Planning and Implementing System Reliability," *IEEE Total Systems Reliability Symposium* (December 12–14, 1983): 112–118.

3. E. J. McCluskey, "Reliable Computing Systems," Technical Note No. 182, Center for Reliable Computing, Stanford University (October 1980).

4. V. A. Cordi, "4381's Error Detection Fault-Isolation Speeds Repairs," *Computer Systems Equipment Design* (November 1984): 23–29.

5. G. H. Maestri, "The Retryable Processor," *IEEE Fall Joint Computer Conference* (1972): 273–277.

Printed in U.S.A. EY-3435E-DP



ISBN 0-932376-83-5

digital™