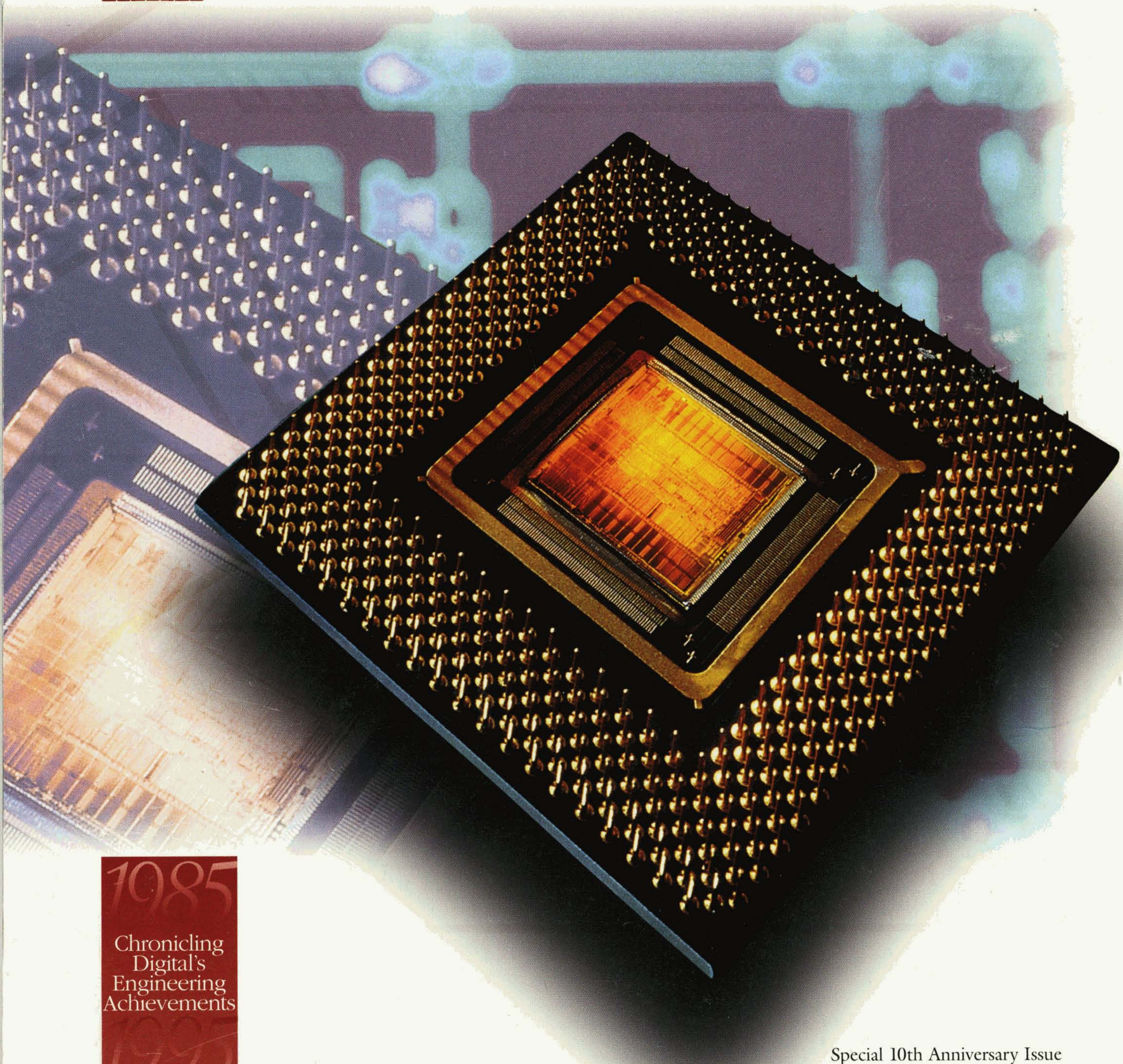
Digital Technical Journal

DATABASE INTEGRATION

ALPHA SERVERS & WORKSTATIONS

ALPHA 21164 CPU

digital™



1985

Chronicling Digital's Engineering Achievements

1995

Special 10th Anniversary Issue
1995

Cover Design
Key to the remarkable performance of the AlphaServer 8400/8200 systems is Digital's new 300-MHz, 64-bit Alpha 21164 microprocessor. Both subjects are featured in this tenth anniversary issue of the *Journal* and are represented on the cover in the forms of a photograph of the microprocessor and an illustration of the AlphaServer system topology.

The cover was designed by Mario Furtado of Furtado Communication Design.

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460. Subscriptions to the *Journal* are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to the *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to dtj@digital.com. Single copies and back issues are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are also available on the Internet at http://www.digital.com/info/DTJ/home.html. Complete Digital Internet listings can be obtained by sending an electronic mail message to info@digital.com.

Digital employees may order subscriptions through Readers Choice by entering VTX PROFILE at the system prompt.

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

ISSN 0898-901X

Documentation Number EY-T135E-TJ

Book production was done by Quantic Communications, Inc.

Contents

In Memoriam

This tenth anniversary issue of the *Digital Technical Journal* is dedicated to the memory of Peter F. Conklin, Corporate Consulting Engineer, who passed away in April 1995. Peter worked at Digital for 26 years. He was one of the pioneers of the DECsystem-10 software group, contributing its first batch and virtual memory subsystems. He was a key contributor in the design of the VAX architecture and its language and run-time environments. He worked in the PDP-11 and Terminals and Printers groups, helping to develop their technical and business strategies in rapidly changing environments. And he was a prime mover of the Alpha program, overseeing first the software development and then all of the engineering, as well as creating its unique cultural and managerial approach to cross-organizational development.

Beyond his outstanding technical and business contributions, Peter espoused and represented all that was good in Digital's culture—respect for and focus on people; doing the right thing for customers and colleagues; driving for concrete results and workable processes. He championed the role of women and minorities as managers and contributors. He helped set up the internal Notes system to further open communications among employees, and between employees and management. And he advocated and modeled the uses of training and development to improve personal and organizational performance at all levels.

Peter's work is woven through the fabric of Digital's history like a bright, unbreakable thread. He will be sorely missed.

Bob Supnik

Editor's Introduction

The *Digital Technical Journal* marks its tenth anniversary with the publication of this issue. Since 1985, the *Journal* has chronicled Digital's engineering achievements from silicon to software: record-breaking microprocessors, standards-setting network technologies, advanced storage architectures, and industry-leading implementations of clusters and distributed systems. More than simply a record, the *Journal* offers readers insights into the how and why of Digital's product designs—in papers written by the design engineers themselves. A look back over the last ten years, however, provides only a partial view to engineering's unique combination of vision and pragmatism, a combination that has spurred industry breakthroughs and established the foundation for the development of today's world-class hardware and software products. To celebrate Digital's outstanding engineering achievements, we have therefore included a special section of historic milestones as part of this anniversary issue. The milestones begin in 1957 with the development of the company's first product, a system module for scientific use that ran at 5 MHz. The milestones continue through the recent introduction of Digital's new high-performance server system based on microprocessors that run at an extraordinary 300 MHz.

The 300-MHz microprocessor and the AlphaServer 8400/8200 system that uses it are in fact featured in this issue. As Dick Sites points out in his Foreword, these second-generation Alpha products truly take advantage of the Alpha 64-bit RISC architecture introduced by Digital in 1992. In addition to discussions of three Alpha hardware designs and the new microprocessor, this issue presents papers on database software technologies. These papers focus on the realities of integrating heterogeneous systems and data sources.

The Database Integrator (DBI) directly addresses the heterogeneity issue by providing a multidatabase management system for data access and integration of distributed data sources. Richard Pledereder, Vishu Krishnamurthy, Mike Gagnon, and Mayank Vadodaria outline the data access issues and compare the DBI approach with others. Their discussion addresses such topics as heterogeneous query optimization, location transparency, global consistency, resolution of semantic differences, and security checks.

Key to solving the problems posed by heterogeneous systems are openness and standards. Both are stressed in the ACMSxp transaction processing monitor design, described by Bob Baafi, Ian Carrie, Bill Drury, and Oren Wiesler. ACMSxp is layered on the OSF's Distributed Computing Environment and uses Transarc's Encina toolkit to support XA-compliant databases. The monitor's application development environment is based on the Structured Transaction Definition Language.

The ACMSxp monitor figures in the next paper, written by Norman Depledge, Bill Turner, and Alexandra Woog, which defines an architecture for improving the effectiveness of heterogeneous environments. The authors first review relevant standards, such as CORBA and DCE, and then describe an open, distributable client-server architecture made up of three tiers: desktop, middleware (founded on the ACMSxp monitor and Digital's ObjectBroker software), and legacy interfaces.

The next set of papers features high-performance systems built on the 300-MHz Alpha 21164 microprocessor. Presented first is the AlphaServer 8000 platform—the basis for the highest performance systems yet developed by Digital. Dave Fenwick, Denis Foley, Bill Gist, Steve VanDoren, and Dan Wissell discuss the principal design issues relative to the aggressive goals set for system data bandwidth and memory read latency. They define their design approach with seven levels of abstraction and review the choices made in each level; the prevailing theme is achieving low memory read latency. As a result, the AlphaServer 8400 and 8200 systems feature a minimum memory read latency of 260 nanoseconds (ns). Moreover, in benchmark tests, the 12-processor AlphaServer 8400 system achieves supercomputer performance levels of 5 billion floating-point operations per second.

Essential for meeting AlphaServer speed requirements was a custom application-specific integrated circuit (ASIC) bus interface. Jean Basmaji, Kay Fisher, Frank Gatulis, Herb Kolk, and Jim Rosencrans describe a timing-driven layout approach for designing and implementing high-performance ASICs. Called CSALT, i.e., CMOS standard-cell alternative technology, the tool suite saved significant project time and provided the customization necessary to support the system's 10-ns bus speed.

Developers of a second-generation processor module for the AlphaServer 2100 multiprocessing system also took advantage of the Alpha 21164 microprocessor performance and at the same time ensured physical compatibility with the first generation. Nitin Godiwala and Barry Maskas highlight the designs that most efficiently used the system bus bandwidth and provided a 1.4 SPEC performance increase over the first-generation module, including a third-level cache, duplicate tag store, and a synchronous clocking scheme.

Also based on the Alpha 21164 microprocessor, the AlphaStation 600 5-series workstation incorporates the 64-bit PCI bus and supports three operating systems. In their paper, John Zurawski, John Murray, and Paul Lemmon focus on the chips that provide high-bandwidth interconnects between the CPU, the main memory, and the PCI bus. They also recount their experiences in the development of a hardware-based verification technique that improved test throughput by five orders of magnitude over the software-based techniques.

The Alpha 21164 microprocessor that is at the heart of the three systems described above delivers an outstanding microprocessor performance (peak) of 1.2 billion instructions per second. Three papers examine the circuit design, the logic functions, and the functional verification of this custom, 64-bit VLSI chip. First, Bill Bowhill et al. examine the circuit design contributions needed to achieve the performance goal of 300-MHz operation. The authors describe the floorplan choices for laying out the 9.3-million transistor chip and the global single-wire clock distribution scheme. They then present a set of significant circuit design challenges—the speed requirement, the complicated microarchitecture, and the large physical size of the chip—and explain circuit implementation decisions for the instruction, execution, and memory units; the system clock; and the three caches.

The paper by John Edmondson et al. describes the functional units of the Alpha 21164 microprocessor: the quad-issue, superscalar instruction unit; the 64-bit integer execution pipelines; two 64-bit floating-point execution pipelines; a high-performance memory unit; and a cache control and bus interface unit. The authors note architectural improvements over the first-generation 21064 microprocessor and provide performance data.

The functional verification of this complex microprocessor is described in our concluding paper. Mike Kantrowitz and Lisa Noack review the many techniques employed to verify the logic design and the PALcode interface, including implementation-directed pseudorandom exercisers used in combination with focused hand-generated tests. The authors relate the lessons learned from the few bugs found in the first prototype of the microprocessor.

An anniversary is a time to look back and ahead. Looking back to the *Journal*'s origins, I want to acknowledge the wisdom of Dick Beane, the *Journal*'s first editor, and Sam Fuller, vice president of Corporate Research. They established the *Journal*'s editorial focus and its structure: to publish technical papers, written by Digital's engineers, that describe the technological foundations of our products, under the guidance of an advisory board responsible for content and editorial philosophy. Because readers responded so well to the working engineer's perspective on product design, the *Journal* has grown from a biannual to a quarterly publication, and was one of the first industry journals to publish electronically on the WorldWide Web. Further, since 1992, papers have been peer reviewed to ensure that readers receive substantive, accurate information on a widening number of topics covered in the *Journal*. Of course this growth would not have happened without the *Journal*'s contributors, the engineers who analyze their unique and informative experiences and share them with their peers. As Digital's engineers add to the timeline of engineering milestones in computer systems, software, networking, storage, semiconductors, and peripherals, the *Journal* will continue to serve its readers by publishing this important work.

The editors thank Bob Supnik, Senior Corporate Consulting Engineer, for his help in bringing together this special issue of the *Journal*.

Upcoming in the *Journal*: systems engineering, Sequoia 2000 research, software environments, scientific computing, and networking.

Jane Blake

Jane C. Blake
Managing Editor

Foreword

Richard L. Sites
Alpha CPU Co-architect
Corporate Consultant Engineer

Welcome to the Tenth Anniversary issue of the *Digital Technical Journal!* As Jane explains in the Editor's Introduction, much has changed since Volume 1 Number 1. In fact, much has changed in only three years, as this issue on Alpha attests.

From Race Cars to Express-Delivery Trucks

What if race cars evolved as quickly as computers? They might stay on the race track and get about three times faster every three years. Or they might move off the track into everyday life.

Just three short years ago, Digital was celebrating in these pages and elsewhere the first-generation Alpha hardware and software announcements. The typical analogy likened Alpha to a race car — blazingly fast, but not seen in your own neighborhood. In this issue, Digital's engineers describe second-generation Alpha hardware and software. Today's analogy is to an express-delivery truck — still faster than the rest of the industry, but now so commonplace in the delivery of business data that it is almost taken for granted.

If we look back at history, most healthy computer architectures did not become firmly established in the marketplace until the introduction of their second generation. The table below shows architecture generations for three popular processor designs, with milestone implementations highlighted. Each architecture goes through many generations, on about a three-year cycle. The second generation in each case has been the point at which the architecture became firmly established. Subsequent generations have represented continuing refinement and performance increases. Alpha fits this industry-wide pattern.

So, with the advent of second-generation Alpha computers, how are we doing?

At the Starting Line

The first-generation Alpha hardware designs focused on performance, especially on high clock rate for the CPU chip. The initial 150-MHz and 200-MHz Alpha systems entered the market when the Intel 486 at 66 MHz and MIPS R4000 at 75 MHz were the fastest chips in their respective lines.

The first-generation Alpha software focused on survival — enough operating system functionality (VMS subset) to support at least some customers, and enough compiler optimization to make at least some (single-user, Fortran, scientific) programs run fast. An important part of the introduction was migration software to help the installed base of VAX/VMS and MIPS/ULTRIX customers move to the Alpha platform. As with any new venture, the approaches were minimalist, with sophistication left to the future.

	IBM System/360	VAX	Alpha
Architecture design	1962	1975	1989
First generation	1965 360/40	1978 VAX-11/780	1992 21064 scientific
Second generation	1968 360/85	1982 VAX-11/750	1995 21164 business
Third generation	1971 370/145	1985 MicroVAX	1998 21264
Fourth generation	1974 Virtual memory 370/168	1988 VAX 6000 Model 200	⋮
Fifth generation	1977	1991 VAX 6000 Model 600	
Sixth generation	1980	1994	
Seventh generation	1983 3090	⋮	
Eighth generation	1986		
Ninth generation	1989 ES/9000		
Tenth generation	1992		
	⋮		

Like race cars, the initial Alpha hardware and software products were criticized for being temperamental — fast indeed for some applications, but not appropriate for others. Some observers assumed that the first generation was a fluke, or that it represented all that Alpha computers could ever be. The reaction to many architectural features, such as 64-bit addressing or relaxed read-write ordering, was "who needs it?" They wrote Alpha off as a niche design.

Delivering on Its Promise

Three years later, the future has arrived with a (muffled) roar:

- The sophisticated second-generation Alpha 21164 chip, described in this issue, is 1.5 to 2 times as efficient as the first-generation 21064 chip in terms of work done per clock cycle on real programs.
- The chip clock rate has been boosted from 200 MHz to a stunning 300 MHz.
- Efficiency of compiled code has improved by 10 to 60 percent on many programs.
- Operating system code has been expanded and tuned.

The performance factors roughly multiply together, producing second-generation systems that are about 2.5 to 3.5 times faster than the equivalent first-generation systems. One example of the higher speeds of these new systems is the AlphaServer 8400, discussed in this issue; system performance approaches the level of supercomputers with Linpack n×n results of 5 GFLOPS.

The second-generation system platforms emphasize industry leadership for a broad range of commercial client-server applications, not just scientific applications. Like express-delivery trucks, much of the second-generation software is focused on enterprise-wide database access. Truly taking advantage of the 64-bit addressing for the first time, Oracle 7 database software can run huge in-memory database queries 200(!) times faster than traditional 32-bit database software. The three database papers in this issue emphasize Digital's focus on commercial applications.

Operating system support is substantially more robust and has been expanded to the fastest UNIX and Windows NT implementations in the industry. Full OpenVMS clustering, including mixed Alpha and VAX clusters, is available. UNIX and NT clustering is announced. All three operating systems now support SMP, symmetric multiprocessing. The 64-bit Digital UNIX implementation has led the rest of the industry in delivering 64-bit software by over 24 months.

Compiled-code improvements have been remarkable. In 1992, I could read the code generated by some of our compilers and redline three out of every four instructions as unneeded unneeded unneededd unneeded. A year ago, I could read compiled code and redline one instruction out of every two as unneeded unneeded. Today, I am hard-pressed to redline even 15 percent of the instructions as unneededd.

Moving beyond the installed base, migration efforts are now focused on bringing in new customers. In addition to VAX and MIPS binary translation, the SPARC-to-Alpha binary translation product is available. Code from x86 PC platforms runs emulated on all Alpha operating systems. A technology demonstration of x86-to-Alpha binary translation has been given at trade shows.

The growing maturity and sophistication of the Alpha products have in turn led to accelerated sales growth. Over 100,000 Alpha systems worth over $3.5 billion (hardware, software, and service) have been shipped, and the ship rate has increased 66 percent in the past year alone. In its first three years, Alpha is off to a much faster start than other RISC architectures, such as HP-PA, in their first three years. Buying patterns have shifted from try-one-out to buy-a-fleet-to-run-the-business.

In three short years, Alpha computers have become established as the fastest in the industry — the yardstick by which others measure computer performance. Competitors have shortened their development cycles and aggressively increased their clock rates. Every single company that described Alpha features as unnecessary in 1992 is now rushing to bring its own 64-bit and relaxed read-write order SMP implementations to market. Alpha has grown from "niche design" to "industry yardstick" in a single generation.

Digital has invested over $1 billion in the development of Alpha. Literally thousands of people have brought a paper design to life. Bleeding-edge, brute-force chip technology has turned into practical engineering, with a balance of sophistication and everyday care: race cars to express-delivery trucks.

Alpha is evolving much as the architects originally envisioned. I believe Peter Conklin, who led the Alpha Program Office and to whom this issue is dedicated in memoriam, would want to also dedicate this issue to all the brilliant and hardworking people who have made it a reality. My thanks and admiration to each of you.

So how are we doing? After you read this issue, I think you will agree, "Quite well, thank you."

DB Integrator: Open Middleware for Data Access

Richard Pledereder
Vishu Krishnamurthy
Michael Gagnon
Mayank Vadodaria

During the last few years, access to heterogeneous data sources and integration of the disparate data has emerged as one of the major areas for growth of database management software. Digital's DB Integrator provides robust data access by supporting heterogeneous query optimization, location transparency, global consistency, resolution of semantic differences, and security checks. A global catalog provides location transparency and operates as an autonomous metadata repository. Global transactions are coordinated through two-phase commit. Highly available horizontal partitioned views support continuous distributed processing in the presence of loss of connectivity. The DB Integrator enables security checks without interfering with the access controls specified in the underlying data sources.

A problem faced by organizations today is how to uniformly access data that is stored in a variety of databases managed by relational and nonrelational data systems and then transform it into an information resource that is manageable, functional, and readily accessible. Digital's DB Integrator (DBI) is a multidatabase management system designed to provide production-quality data access and integration for heterogeneous and distributed data sources.

This paper describes the data integration needs of the enterprise and how the DBI product fulfills those needs. It then presents the DBI approach to multidatabase systems and a technical overview of DBI concepts and terminology. The next section outlines the system architecture of the DBI. The paper concludes with highlights of some of the technologies incorporated in DBI.

Data Integration Needs

Companies often find themselves data rich, but information poor. Propelled by diverse application and end-user requirements, companies have made significant investments in incompatible, fragmented, and geographically distributed database systems that need to be integrated. Companies with centralized information systems are seeking methods to distribute this data to inexpensive, departmental platforms, which would maximize performance, lower cost, and increase availability.

The DB Integrator product family is specifically designed and implemented to address the following data integration needs:

- Data access. The data integration product must provide uniform access to both relational and nonrelational data regardless of location or storage form. Data access must be extensible to allow the user to write special-purpose methods.
- Location and functional transparency. The location of the data and the functional differences of the various database systems must be hidden to provide end users with a single, logical view of the data and a uniformly functional data access system.

- Schema integration and translation. Users of data integration software must be presented with an environment that lets them easily determine what data is available. Such an environment is frequently referred to as a federated database. A data integration product must be flexible enough to help resolve semantic inconsistencies such as variances in field names, data types, and units of measurement.
- Data consistency. Maintaining data consistency is one of the most important aspects of any database system. This is also true for a federated database.
- Performance. Integrating data from multiple data sources can be an expensive operation. The two primary goals are to minimize the amount of data that is transferred across the network and to maximize the amount of rows that are processed within a given unit of time.
- Security. Access to distributed data must not compromise the security of data in the target databases. The security model must provide authorized access to an integrated schema without violating the security of the autonomous data sources that have been integrated.
- Openness. Any data integration product must accommodate tools and applications with standard SQL (structured query language) interfaces, both at the call level (e.g., Open Database Connectivity [ODBC] for personal computer clients) and the language level (e.g., ANSI SQL).[1,2] It must be able to provide and enable access to data over the most commonly deployed transports such as transmission control protocol/internet protocol (TCP/IP), DECnet, or Systems Network Architecture (SNA).[3]
- Administration. The integrated database must provide flexibility in configuration and be easy to set up, maintain, and use.

Figure 1 illustrates the current set of client-server data access supported by the DB Integrator product family.

Multidatabase Management Systems

A multidatabase management system (MDBMS) enables operations across multiple autonomous component databases. Based on the taxonomy for multidatabase systems presented in Reference 4, we can describe DBI as a loosely coupled, heterogeneous, and federated multidatabase system. DBI is loosely coupled compared to the component databases: The database administrator (DBA) that is responsible for DBI and the DBAs that are responsible for the component databases manage their environments independently of one another. DBI is heterogeneous because it supports different types of component database systems. DBI is federated because each component database exists as an independent entity.

Reference Architecture

The MDBMS provides users with a single system view of data distributed over a large number of heterogeneous databases and file systems. The MDBMS interoperates with the individual component databases similar to the way that the SQL query processing engine in a relational DBMS interoperates with the

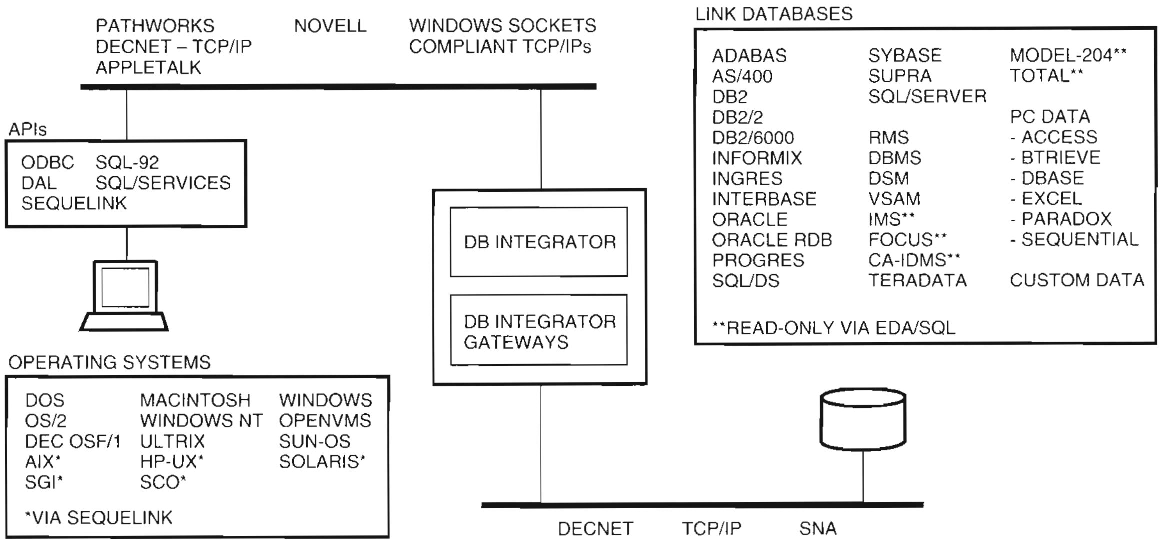


Figure 1
Client-Server Data Access with the DB Integrator

record storage system. Thus, a relational MDBMS, such as DBI, is typically composed of the following processing units:

- Language application programming interface (API) and SQL parser
- Relational data system
  - Global catalog manager
  - Distributed query optimizer and compiler
  - Distributed execution system
  - Distributed transaction management
- Gateways to access data sources

Catalog Management

One of the key differentiators between MDBMS architectures is the way that the metadata catalog is organized. Metadata is defined as the attributes of the data that are accessible (e.g., naming, location, data types, or statistics). The metadata is stored in a catalog. Two common approaches for catalog management are described below:

- Autonomous catalog. The MDBMS maintains its own catalog in a separate database. This catalog describes the data available in the multidatabase. For data that resides in a relational database, the metadata definitions of table objects, index objects, and so forth, are imported (i.e., replicated) into the multidatabase catalog. For data that resides in some other data source such as a record file system (e.g., record management system [RMS]) or a spread sheet, the MDBMS catalog contains a relational description of that data source.
- Integrated catalog. The MDBMS is integrated with a regular database system that is capable of accessing objects (both data and metadata) in remote and foreign databases. A gateway server is responsible for making a foreign database appear as a homogeneous, remote database instance. For data that resides in a relational database, the gateway server stores views of its system relations into that database. For data that resides in a record file system or spread sheet, the gateway server stores the relational metadata description of the data in a separate data store.

DBI Approach

The DBI approach to multidatabase management very closely follows the reference architecture presented earlier. The DBI approach emphasizes the following design directions:

- Global, autonomous catalog for metadata management
- Three-tier integration model (described later in this section)
- Simple, mapped-in gateway drivers to access data sources
- Support of distributed database features for the Oracle Rdb relational database as well as support of existing Oracle Rdb applications in the multidatabase environment

Global Catalog DBI is addressable as a single integration server. Integration clients such as tools and applications do not need to deal with the complexities of the distributed data. The DBI global catalog is a repository in which DBI maintains the description of the distributed data. It enables DBI to provide tools and applications with a single access point to the federated database environment. The global catalog enables DBI to tell users what data is available without requiring immediate connectivity to the data or its data source. It can be managed and maintained as an independent database. The maintenance of the DBI global catalog is not inherently tied to a specific data manager; currently, the DBI catalog may reside in ORACLE, SYBASE, or Oracle Rdb databases.

The use of a global catalog may result in a system with a single point of failure. To eliminate its potential failure within a node, a disk, or a network, standard high-availability mechanisms may be employed. These include shadowed disks with shared access (e.g., clustered nodes) and data replication of the DBI catalog tables with products such as the Digital Data Distributor.[5]

Three-tier versus Two-tier Architecture With a two-tier data integration model, once the data has been retrieved from the server tier, the actual integration occurs on the client tier. This may result in massive integration operations at the client site. In contrast, the DBI is based on a three-tier architecture that performs most integration functions on a middle tier between the client and the various database servers. The three-tier approach avoids unnecessary transfer of data to the client and is essential to providing production-quality data integration. In another comparison, all clients in the two-tier approach need to be configured to access the various data sources; however, the three-tier approach significantly reduces such management complexities.

Gateway Driver Model DBI deploys a set of gateway drivers to access specific data sources, including other DBI databases. These drivers share a single operating system process space with DBI to avoid unnecessary interprocess communications. When DBI performs parallel query processing, however, gateway drivers may reside in a separate process space. The core of DBI interacts with the actual gateway drivers (e.g., a SYBASE gateway driver) through the Strategic Data

Interface (SDI), an architected interface that is used within the DBI product family as a design center.[6] A gateway driver is implemented as a relatively thin software layer that is SDI compliant and that is responsible for handling impedance mismatches in data models (e.g., RMS versus relational), query language (e.g., different dialects of SQL), and run-time capabilities (e.g., SQL statement atomicity).

Distributed Rdb One of the design goals for DBI was to enable distributed database processing for DEC Rdb (now Oracle Rdb).[7] From the perspective of an application, DBI therefore looks like a distributed Rdb database system.

DBI Concepts and Terminology

In this section, we present a brief overview of the concepts and terminology relevant to DBI.

DBI Database

A DBI database consists of (1) a set of tables that DBI creates to maintain the DBI metadata (also referred to as the catalog) and (2) the distributed data that is available to the user when connected to the DBI catalog.

A DBA creates a DBI database using the DEC SQL CREATE DATABASE statement. This statement has been extended for DBI to allow the user to indicate the physical database (e.g., a SYBASE database) that will be used to hold the DBI metadata tables.

The creator of a DBI database automatically becomes the owner and system administrator of that database. A DBI system administrator may grant access privileges on the DBI database to other users. Depending on the level of privilege, a user may then perform system administration functions, execute data definition language (DDL) operations, and/or query the tables in the virtual database.

DBI Objects

In addition to regular SQL objects such as tables or columns, DBI uses objects, links, and proxies that are outside the scope of the SQL language standard.

Links and Proxies The link object tells DBI how to connect to an underlying data source (referred to as the link database). A link object has three components: a link name, the access string used to attach to the link database, and, optionally, security information used by the DBI gateway driver to provide authentication information to the link database system. The proxy object is associated with a link object. It can be used to specify user-specific authentication information for individual links. When users do not want to use proxies for their links, they must specify the authentication information for a specific database at the time they connect to DBI.

Tables With link and proxy objects in place, the user can import metadata definitions of underlying tables into the DBI catalog. The metadata imported for a table includes statistics, and constraint and index information, all of which are used by the DBI optimizer. The import step is performed with a CREATE TABLE statement that has been extended to allow for a link reference. For example:

```
-- Import "rdb_emp" table into DBI
-- database as "emp" from the link
-- database represented by the link
-- named "link_rdb".
--
CREATE TABLE emp LINK TO rdb_emp
  USING link_rdb;
```

Views View objects are useful for making multiple tables from different link databases appear as a single table. In DBI, views serve as powerful mechanisms to resolve semantic differences in tables from disparate databases. DBI supports two types of views: regular SQL views and horizontally partitioned views (HPVs). Regular views are compliant with ANSI SQL92 Level 1; they support full query expression capabilities and updatability.[2] HPVs consist of a view name, a partitioning column, and partition specifications. Figure 2 is an example of an HPV definition.

HPVs provide a very powerful construct for defining a logical table composed of horizontal partitions

```
CREATE VIEW emp (emp_id, first_name, last_name, country)
       USING HORIZONTAL PARTITIONING ON (country)

       PARTITION us WHERE country = 'US' COMPOSE AS
                 SELECT employeeid, firstname, lastname, 'US'
                 FROM emp_us

       PARTITION europe WHERE OTHERWISE COMPOSE AS
                 SELECT emp_id, first_name, last_name, country_code
                 FROM emp_eur;
```

Figure 2
Example of an HPV Definition

that may span tables from disparate data sources. Both retrieval and update operations on HPVs are optimized such that unnecessary partition access is eliminated. In addition, HPVs may be used to implement a shared-nothing computing model on top of both homogeneous and heterogeneous databases.[8]

Stored Procedures DBI supports stored procedure objects. Stored procedures allow the user to embed application logic in the database. They make application code easily shareable and facilitate DBI to maintain dependencies between the application code and database objects. Furthermore, stored procedures reduce message traffic between the client and the server. Figure 3 is an example of a stored procedure.

DBI Database Administration

DBI supports statements that keep the imported metadata consistent with the link database. The extended ALTER TABLE statement may be used to regularly refresh the table metadata information or update the table's statistics. The ALTER LINK statement may be used to modify the link database specification or a proxy for a given link object.

DBI Configuration Capabilities

Figure 4 shows the power of configuration options supported by DBI. Following the three-tier model for data integration, the DBI server may access a very large number of databases, including other DBI databases.

The DBI server is accessible through SQL APIs that are available on popular client platforms. DBI's client-server protocol is supported on all common transports such as TCP/IP, Novell's sequenced packet exchange/internetwork packet exchange (SPX/IPX), DECnet, or Windows Sockets. DBI itself may be deployed on Digital UNIX (formerly DEC OSF/1) and OpenVMS platforms today. Support for additional platforms is being added.

DBI System Architecture

In this section, we describe the system architecture of the DBI product family and present some of its specific designs.

Interfaces

As shown in Figure 5, the DBI system architecture is anchored by two external interfaces, SQL and metadata driver interfaces/data driver interfaces (MDI/DDI), and two internal interfaces, Digital Standard Relational Interface (DSRI) and SDI.

The SQL interface is used by DBI clients to issue requests to the integration server. The MDI/DDI interface is used by DBI to call gateway drivers that are provided by a user. The MDI/DDI interface specifies a simple, record-oriented data access interface provided by Digital to assist users in the access and integration of data sources for which no Digital-supplied gateway drivers are available.

DSRI is the interface between DBI's SQL parser and the DBI processing engine.[9] The SDI interface specifies a canonical data interface that shields the DBI core from data-source-specific interfaces and facilitates modular development.[6]

```
procedure maintain_salaries(:state char(2) in,
                            :n_decreased integer out);
begin
    set :n_decreased = 0;
    for :empfor as each row of
         select * from employees emp where state = :state;
     do
          set :last_salary = 0;
          history_loop:
          for :salfor as for each row of
              select salary_amount from salary history s
              where s.employee_id = :empfor.employee_id
       do
              if :salfor.salary_amount {lt} :last_salary then
                  set :n_decreased = :n_decreased + 1;
                  leave history_loop;
              end if;
               set :last_salary = :salfor.salary_amount
          end for;
    end for;
end;
```

Figure 3
Example of a Stored Procedure

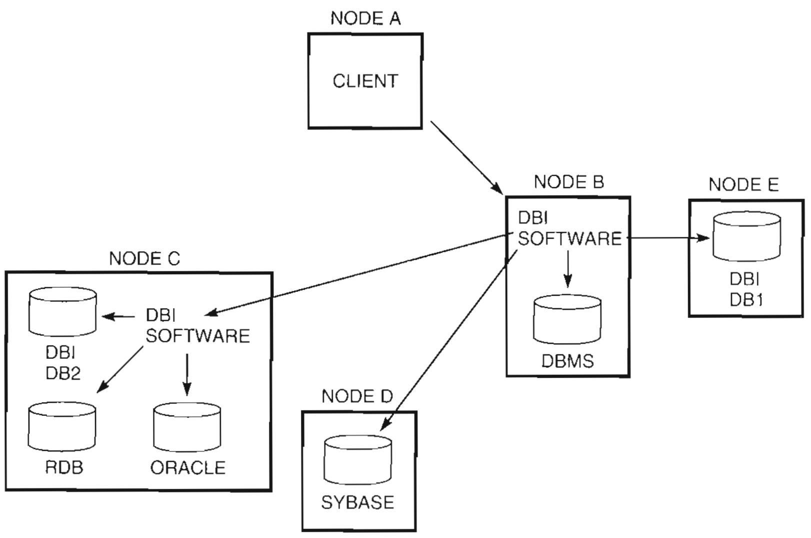


Figure 4
DBI Configuration Capabilities

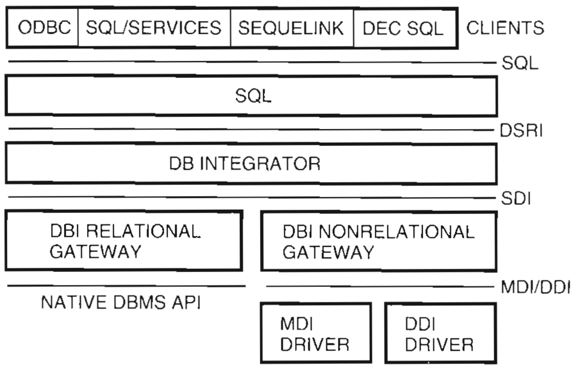


Figure 5
DB Integrator Architecture

Components

The component architecture of DBI in Figure 6 closely resembles the multidatabase reference architecture presented earlier:

- The SQL and ODBC client-server environment provides language API and SQL parser functions.
- The API driver and context manager support distributed transaction management and part of the distributed execution system.
- The metadata manager provides global catalog management.
- The compiler supports the distributed query optimizer and compilation.
- The executor supports the remaining part of the distributed execution system.
- The SDI dispatcher and gateway drivers provide the access to data sources.

SQL Environment and Server Infrastructure The SQL parser supports DEC SQL, an ANSI/National Institute for Science and Technology (NIST)-compliant SQL implementation by mapping DEC SQL syntax into an internal query graph representation.[9] In a client-server environment, the DBI server infrastructure is used to manage, monitor, and maintain a DBI server configuration that supports workstation and desktop clients.

API Driver and Context Manager The API driver is responsible for the top-level control flow of client requests within the DBI core. It currently accepts DSRI calls from applications such as DEC SQL and dispatches them within DBI. The context manager performs demand-driven propagation of execution context to the gateway drivers and maintains the distributed context of active sessions, transactions, and requests.

Metadata Manager The metadata manager is responsible for the overall management and access to metadata. The services provided fall into the categories of catalog management, data definition, metadata cache management, and query access to DBI system relations. The metadata catalog manager maintains the DBI catalog in the form of DBI-created tables in an underlying database (e.g., SYBASE or ORACLE). The DDL processor executes the data definition statements. The metadata cache manager is responsible for maintaining metadata in a volatile cache that provides high-speed access to metadata objects.

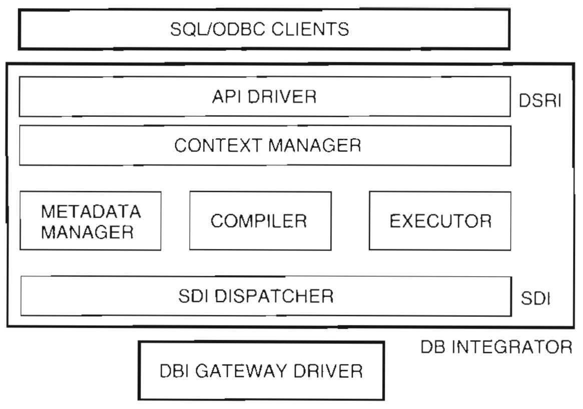


Figure 6
DB Integrator Components

Compiler The compiler provides services for translating SQL statements and stored procedures into DBI execution plans. A rule-based query optimizer performs query rewrite operations, enumerates different execution strategies, and factors in functional capabilities of the underlying data sources. Each execution strategy is associated with a cost that is based on predicate selectivity estimates, table cardinalities, availability of indices, network bandwidth, and so forth. The lowest cost strategy is chosen as the final execution plan. Above a certain threshold of query complexity, the optimizer switches from an exhaustive search method to a greedy search method to limit the computational complexity of the optimization phase. The compiler generates code that can be processed by the executor component and the gateway drivers.

Executor The executor component is responsible for processing the execution plan that the compiler produces. These activities include

- Exchanging data between the DBI and the client
- Streaming data between the DBI core and the link databases
- Performing intermediate data manipulation steps such as joins or aggregates
- Managing workspace and buffer pool to efficiently handle large amounts of transient and intermediate data
- Supporting parallel processing

SDI Dispatcher and Gateway Drivers The SDI dispatcher separates the core of DBI from the gateway driver space. It locates and loads shareable images that represent gateway drivers and routes SDI calls to the corresponding entries in the gateway driver image.

Technical Considerations

The DBI development team selected several designs and technologies that it believes to be crucial for distributed and heterogeneous data processing. This section summarizes those designs within the following functional units: distributed execution; distributed metadata handling; distributed, heterogeneous query processing; high availability; performance; and DBI server configuration.

Distributed Execution

To support transparent distributed query processing, DBI propagates execution context such as connection context or transaction context to the target data sources. Tools and applications see only the simple user session and transaction that they establish with the DBI integration server.

DBI uses a tree organization to track the distributed execution context. When a user connects to a DBI database, a DBI user session context is created. This session context is subsequently used to anchor active transactions, compiled SQL statements, as well as the metadata cache that is created for every user attaching to DBI. When DBI passes control to a gateway driver, both session and transaction context are established at the target data source.

Distributed transactions must support consistency and concurrency across autonomous database managers. Consistency requires that a distributed transaction manager with two-phase commit logic is available. DBI uses the Digital Distributed Transaction Manager (DDTM) for that purpose and is adding support for the distributed transaction processing (DTP) XA standard integration.[10,11]

Concurrency requires that distributed deadlocks are detected. In a multidatabase system, distributed deadlock prevention is not feasible because no database manager exposes external interfaces to its lock management services--a procedure required to perform deadlock detection. DBI therefore relies on the simple technique of transaction time-out to detect deadlocks. In addition, a DBI application may choose to specify isolation levels lower than serializability or repeatable read. This is done with the SQL SET TRANSACTION statement. The DBI context manager records the transaction attributes specified and forwards them to the underlying data sources as part of propagating transaction context. Lower isolation levels will, in general, result in fewer lock requests and thus fewer deadlock situations.

Distributed Request Activation DBI supports SQL statement atomicity. This requires either that a single SQL statement executes in its entirety or, in the case of a failure, that the database is reset to its state prior to the execution of the statement. With DBI, the SQL statement may be executed as a series of database requests at multiple data sources. DBI internally uses the concept of markpoints to track SQL statement boundaries. Gateway drivers are informed of markpoint boundaries, and the driver attempts to map the markpoint SDI operations into semantically equivalent constructs (e.g., savepoints) at the target data source. Some databases support SQL3-style savepoints, which are atomic units of work within a transaction. When DBI decides to roll back a markpoint, the gateway driver may then inform such a data source to roll back to the last savepoint. In the absence of markpoint primitives in the target data source, the gateway driver may elect to roll back the entire transaction to meet the roll-back markpoint semantics.

Gateway Drivers In contrast with other data integration architectures, the DBI gateway drivers are designed to be simple protocol and data translators. Their primary task is to report the capabilities of the data-source interface (API and SQL language) to the DBI core and subsequently map between the SDI interface protocol and the data-source interface. The gateway drivers typically share process context with the DBI server process, thus avoiding the need for an intermediate gateway server process that would otherwise reside between the DBI server and the data-source server (e.g., SYBASE SQL Server). This reduces the amount of context switching and interprocess message transfer.

The gateway drivers are responsible for mapping the SDI semantics to the interface primitives provided at the target data source. For relational databases such as Oracle Rdb, ORACLE, INFORMIX, SYBASE, or DB2, this requires primarily a mapping to the product-specific SQL dialect and the product-specific data types. For file systems such as RMS, the gateway driver maps the SDI semantics to calls to the RMS run-time library.

Distributed Metadata Handling

In this section, we discuss three areas of importance to the handling of metadata in DBI: catalog management, security, and metadata caching.

Catalog Management The DBI requirement of database independence implies that DBI cannot require the presence of a particular DBMS for its persistence metadata storage. Rather than devising a private storage and retrieval system, DBI was designed to layer on top of common relational DBMSs.

Static, precompiled native applications are used to access metadata from a given catalog DBMS for two reasons: (1) The pattern of metadata access for the catalog database is known, and (2) The tables housing the DBI metadata in the catalog database are predetermined. Although this approach does not take advantage of the existing gateway drivers, it results in high-performance access to the metadata store.

To simplify the development of a catalog application, the set of primitive operations on the catalog database was isolated, and a catalog application interface (CI) was defined. Catalog applications are developed according to the CI specification and implemented as shareable images. DBI dynamically loads the appropriate catalog application image based on the catalog type specified by a user attaching to a DBI database.

Security The security support in the currently released version 3.1 of DBI is simple but effective. It uses the security mechanisms of the underlying link database systems in the following areas:

- Authorization to connect to an underlying database through DBI and access data from it.

  Access to the data that is manipulated through DBI is controlled by the underlying DBMS. Typically, underlying database systems control access to data based on the identity of the user attached to its database. DBI supports objects called proxies that enable the client to specify its user identity (i.e., username/password), which is then used to attach to the underlying database.

- Authorization to perform various DBI operations.

  All privileges for a DBI database are for the database itself, rather than for tables or columns. The privileges are based on hierarchically organized categories of users:

  - The DBADM privilege is given to users responsible for setting up and maintaining a DBI database.
  - The CREATE, DROP privilege is granted to interactive users and application developers with database design responsibility who must perform data definition operations.
  - The SELECT privilege is reserved for interactive users and application developers who perform data manipulation operations but do not perform any data definition operations.

When a DBI administrator grants or revokes privileges for a DBI database, DBI, in turn, grants or revokes the appropriate set of privileges on the DBI tables in the database system that manages the DBI catalog. The enforcement of privileges is therefore carried out by that database system. For example, when the SELECT privilege is granted on the logical database, DBI grants the SELECT privilege on the tables that represent the DBI catalog. This ensures that the user has access to the metadata for processing queries.

Similarly, when a user is granted the CREATE, DROP privilege on the DBI database, DBI grants SELECT, INSERT, UPDATE, and DELETE on the appropriate tables in the catalog database to the user. This ensures that any DDL actions executed by the user will enable DBI to modify the tables in the catalog database.

Metadata Manager Cache The in-memory metadata cache serves a dual purpose. First, it facilitates rapid access to the metadata by the DBI compiler. Second, it serves as a data store for the DBI system relations that can be queried by tools and applications. For example, DEC SQL obtains metadata for semantic analysis of SQL statements by querying the DBI system relations.

The metadata cache is structured as a single hash table representing a flat namespace across all DBI objects. An open hashing scheme is employed in which the hash-table entries hang off the buckets in the hash table in a linked list.

To optimize the use of the cache as well as to accelerate the attach operation, the metadata manager initially obtains only minimal, high-level metadata information from the catalog database; for example, only names of tables are fetched into the cache during the DBI database attach operation. Subsequently, the metadata manager obtains further metadata information from the catalog database on a demand basis.

DBI allows the creation of new metadata objects. These operations are typically performed within markpoint and transaction boundaries to enforce proper statement and transaction demarcation. The metadata manager maintains a physical log in cache to denote transaction and markpoint boundaries. The log is an ordered list of structures, each representing a DDL action, a pointer to the cache structure that was changed, and either the previous values of fields that were updated or a pointer to a previous image of an entire structure. When a markpoint or transaction is committed, the corresponding log part is reset; when a markpoint or transaction is rolled back, the log is used to restore the cache to its state prior to the start of the markpoint or transaction.

An object in cache can become stale when another user attaches to the DBI database and causes an object's metadata to be changed in the catalog database. To ensure consistency of the cached version of an object's metadata with the actual version in the catalog database, the metadata manager uses a time stamp to check the currency of the cached object when performing incremental fetching of the object's metadata. If the object in cache is stale, the object is not accessible in the session, and an error message is issued to the user indicating that the object in cache is inconsistent with the catalog database. In a production environment, this would be a rare event, given the low frequency of data definition operations.

The metadata cache is also the data source for the DBI system relation queries. The metadata manager navigates the cache structures to obtain data for the system relations, making use of the hash table for efficient access and using DBI's execution component for evaluating search conditions and expressions.

Distributed, Heterogeneous Query Processing

Distributed query processing in a heterogeneous database environment poses certain unique problems. Data sources behave differently in terms of data transfer cost, and they support different language constructs. Many systems employ rudimentary techniques for decomposing a query, frequently pulling in all the data from underlying tables to the processing node, and then performing all the operations in the integration engine. Others simply use syntactic transformations, thereby providing the least common denominator in language functionality. DBI, on the other hand, provides a robust query optimizer that includes decomposition algorithms to reduce the data flow and provide high-performance query execution.

Cost-based Plan Generation When a query has several equivalent means of producing the result, the plan that has the least estimated cost is chosen. Statistics for table, column, and index objects are used for estimating result size after various relational operations.[12,13] Data transmission costs from the underlying link database to DBI are taken into account when estimating how much of the query is to be sent to the gateway database. The network transmission cost is measured dynamically for each user session, once per gateway connection. The cost associated with performing a relational operation is also aggregated into the overall cost. This crucial step ensures that the plan is not skewed toward one database engine, which would be the case if only the network transmission costs were taken into account.

Rule-based Transformations A query result may be produced with different sequences of relational operations. These sequences are generated using rule-based transformations. The starting point is the original operation set in which the query was syntactically represented. From this, permutations are generated to form equivalence sets, which then lead to the various combinations of execution plans that need to be examined for cost. Finally, the least costly plan is chosen for the query. Heuristics are applied to limit the amount of search space.

Capability-based Decomposition The critical characteristic of a heterogeneous environment is that the data sources are nonuniform in their ability to perform certain operations and in their support of various language constructs. For example, most databases cannot support derived table expressions (i.e., select expressions in the FROM clause of another SELECT statement).

The plan generation and decomposition phases of the optimizer must recognize the underlying databases' capabilities. Consider the query example shown in Figure 7 and the indicated locations of the tables.

First, with T1 and T3 located in the same database, the optimizer can generate a subplan in which the join between these two tables can be executed in the ORACLE database. An examination of the last (third) AND predicate indicates that all the tables involved in that predicate are located in the same ORACLE database. Due to the limitations in ORACLE's SQL language support, however, it cannot evaluate the combined expression between two subqueries in the WHERE clause, where the arithmetic result is to be compared to the column T1.c5.

The DBI optimizer employs a more sophisticated alternative. It evaluates the two subqueries separately and then substitutes them in the predicate in the subplan for ORACLE as run-time parameter values. This technique leads to the most efficient plan:

1. Retrieve value for (select avg(T4.c5) from T4) from ORACLE.
2. Assign value to variable *X*.
3. Retrieve value for (select T5.c7 from T5 where T5.c8 = 'a') from ORACLE.
4. Assign value to variable *Y*.
5. Assign param_1 := variable *X*.
6. Assign param_2 := variable *Y*.
7. Execute the SELECT statement below in ORACLE and fetch the result rows.

```
select *
  from T1, T3
 where (T1.c3 = T3.c3)
   and (T1.c5 = param_1 + param_2);
```

8. Fetch the rows of T2 from DB2 into DBI.
9. Perform the join in DBI between the results of steps 7 and 8.

Query Unnesting A nested SQL query, in its simplest form, is a SELECT query with the WHERE clause predicate containing a subquery (i.e., another SELECT query). The following are examples of nested SQL queries:

Example 1, Table Subquery

```
select *
  from A
 where A.c1 IN (select (B.c2 + 5)
                  from B
                 where B.c3 = A.c3);
```

Example 2, Scalar Subquery

```
select *
  from A
 where A.c1 = (select max(B.c2)
                 from B
                where B.c3 = A.c3);
```

Using strict SQL semantics, we can evaluate this nested query by computing the results of the inner subquery for every tuple in the outer (containing) query block. The value for the column A.c3 is substituted in the inner subquery, and the resulting value (or values) are computed for the select list and used to evaluate the Boolean condition on column A.c1: this is repeated for every tuple of A. This method of evaluating the results is very expensive, especially in a distributed environment.

Query unnesting algorithms provide other methods of evaluation that are semantically equivalent but much more efficient in both time and space. Unnesting deals with the transformation of nested SQL queries into an equivalent sequence of relational operations. These relational operations are performed as set operations, thereby avoiding the expensive tuple iteration operators during execution and providing large performance gains in most cases. The background and motivation behind the use of unnesting has been presented in several research papers.[14,15]

```
select *
  from T1, T2, T3
 where (T1.c1 = T2.c2)
   and (T1.c3 = T3.c3)
   and (T1.c5 = (select avg(T4.c5) from T4)
                + (select T5.c7 from T5 where T5.c8 = 'a') );

T1, T3, T4 and T5 are located in a Oracle database.
Table T2 is located in a DB2 database.
```

Figure 7
Example of an SQL Query

Depending on the type of operations and constructs found in the nested select block and its parent select block, several different algorithms can be used. Some of these require no special operators over and above the regular join operator. Other transformations require a special semijoin operator. Consider the examples shown in Figure 8.

In the example shown in Figure 9, a special operator called semijoin is necessary. The semijoin of table *R* with *S* on condition *J* is defined as the subset of *R*-tuples for which there is at least one matching *S*-tuple satisfying *J*. Note that this makes the operator asymmetric, in that (*R* semijoin *S*) is not the same as (*S* semijoin *R*), whereas the regular join is symmetric. By implementing the special semantics required for this semijoin operator, we can transform the nested query into this join operator that can again make use of high-performance techniques like hash joins within the DBI execution engine.

Predicate Analysis When a query against an HPV can be satisfied by simply accessing a single logical partition, then the rest of the partitions can be eliminated from the execution plan. Partition elimination algorithms in DBI are used both at compile time, when the predicates on the HPV query involve comparison of the partitioning column with literals, as well as at query execution time (run time), when the partitioning column is compared with run-time parameters.

During affinity analysis, predicates are situated as close to the inner table operation as feasible. For example, consider the following view definition, and the subsequent select statement on that view:

```
create view V1 (a, b) as
  select T1.c1, avg(T2.c2)
    from T1, T2
   where (T1.c4 = T2.c4)
   group by T1.c1;

select * from V1 where (a = 5 and b > 10);
```

The predicate a = 5 (upon further conjunctive normal form [CNF] analysis) can be applied on the base table scan itself as T1.c1 = 5.

Index join is one of the efficient join techniques used in DBI. This join technique minimizes the movement of data from the link databases by taking advantage of the indexing schemes in the link database to facilitate the join process. Consider the following query:

```
select *
  from T1, T2
 where T1.c1 = T2.c2 + 5
   and (...some restrict predicate(s)
       on T2...)
```

```
--
-- Q1  - query that will not require a special join after transformation
--
select snum, city, status
  from S
 where status = (select avg(weight) + 5     -- nesting predicate
                   from P
                  where P.city = S.city);   -- correlation predicate

--
-- Q1-U - the unnested version
--
select snum, city, status
  from S, (select city, avg(weight) + 5
             from P
            group by city) as T1(c1,c2)
 where T1.c1 = S.city
   and S.status = T1.c2;

--  Algorithm:
--
--  1) Take the inner block's FROM table that has a correlation predicate.
--  2) Add a Group-By to the inner block containing all attributes of this
--     table that appear in one or more correlation predicates. The order of
--     the attributes in the Group-By does not matter.
--  3) Also, add these elements to the select list of the inner block; at the
--     beginning or at the end, whatever is convenient.
--  4) Next, add this block to the FROM list of the outer block - effectively
--     doing a regular join with the tables in the outer FROM list.
--  5) Lastly, rewrite the correlation and nesting predicates as shown.
```

Figure 8
Query Unnesting Algorithm

```
-
- Q2 - query requiring a semi-join
-
select snum
  from S
 where city IN (select city
                  from P
                 where P.weight = S.status);

-
- Q2-U - the unnested version
-
select snum
  from (S  semi-join P
           on (P.weight = S.status AND S.city = P.city)
       );

--Algorithm:
--
--1) Do a semi-join between S and P using the following (combined) condition:
--      "(P.weight = S.status) AND (S.city = P.city)"
--    In reality, this is actually specified as 2 separate semi-joins between
--    S and P, one with the correlation predicate and one with the form of
--    the nesting predicate. But these get combined using rules.
-- 2) Project out S.snum from the result
```

Figure 9
Algorithm with Semijoin Operator

Given an index on column c1 of table T1, and with cardinality and cost estimates permitting, the query optimizer can generate an alternate plan. This plan allows the join to be performed by using efficiently indexed access retrieval for table T1.

High Availability

High availability in DBI results from the use of horizontal partitioned views and catalog replication.

Horizontal Partitioned Views An HPV is a special kind of view in which DBI is provided with information about how data is distributed among tables in link databases. HPVs offer many advantages over normal views, one of them being improved performance through partition elimination and use of parallelism. The other advantage is high availability.

If a partitioned view has multiple partitions and if some partitions are unavailable when the view is queried, then DBI does not fail the query but returns data from the available partitions. An example is shown in Figure 10. The example creates a partitioned view named ALL_EMPLOYEES, with four columns and three partitions, each of which obtains rows from three different tables. The partitioning is based on a specific column, in this case the CITY column, as specified in the USING HORIZONTAL PARTITIONING ON clause.

Suppose the following query is submitted

```
SQL> SELECT * FROM ALL_EMPLOYEES
              WHERE (CITY = 'MUNICH')
                 OR (CITY = 'NASHUA');
```

First, partition P2 is eliminated at compile time. Now suppose partition P3 is presently not available due to network connectivity problems (Figure 11). For each partition that is unavailable, a message is returned indicating that some rows are missing from the result table: %DBI-W-HAHPV_UNAVAILABLE Partition P3 is currently unavailable. However, DBI still attempts to return as much data as is accessible.

Catalog Replication To prevent the DBI global catalog from becoming a single point of failure, multiple copies of a catalog table can be maintained by using replication techniques. Catalog table copies can be created easily and maintained using replication tools such as the DEC Data Distributor.[5]

Performance

In addition to its distributed query optimizer, DBI uses a series of techniques to increase the speed of query processing, most notably in the areas of data transfer, memory management, join processing, parallelism, and stored procedures.

Data Transfer The DBI execution engine performs bulk data transfer using the bulk fetch mechanisms provided by the SDI interface. With bulk data transfer, a single request message to a local or remote data source returns many tuples with a single response message. Bulk transfer techniques are mandatory in a distributed environment; they reduce both message traffic and stall waits due to message delays. The data

```
CREATE VIEW ALL_EMPLOYEES(ID, NAME, ADDRESS, CITY)
USING HORIZONTAL PARTITIONING ON CITY
PARTITION P1 WHERE CITY = 'MUNICH'
          COMPOSE AS SELECT ID, LAST_NAME, ADDRESS, 'MUNICH'
                     FROM   MUNICH_EMPLOYEES
                     WHERE  STATUS = 'Y'

PARTITION P2 WHERE CITY = 'PARIS'
          COMPOSE AS SELECT ID, FULL_NAME, ADDRESS, 'PARIS'
                     FROM   PARIS_EMPLOYEES
                     WHERE  STATUS = 'Y';

PARTITION P3 WHERE CITY = 'NASHUA'
          COMPOSE AS SELECT ID, FULL_NAME, ADDRESS, LOCATION
                     FROM   NH_EMPLOYEES
                     WHERE STATUS = 'Y';
```

Figure 10
Example of a Partitioned View

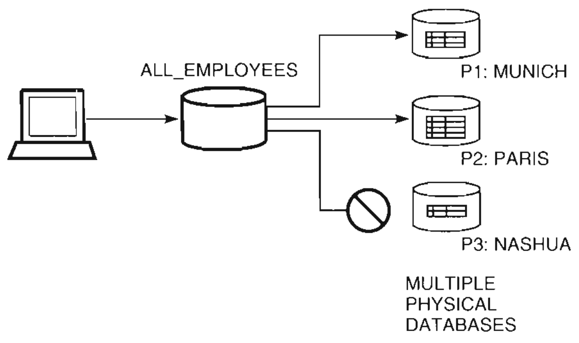


Figure 11
High Availability with Partitioned Views

transfer bandwidth between the DBI engine and the gateway drivers is further increased through the use of asynchronous SDI operations.

Memory Management An MDBMS needs to be able to process large amounts of data efficiently without exceeding platform- or user-specific operational quotas such as the page file size or the working set limit. In addition, standard operating system paging techniques may easily result in heavy I/O thrashing for database-centric work loads.

The DBI executor places data streams, intermediate query results, or hash buckets into individual workspaces. A workspace is organized as a linear sequence of fixed-size pages. A standard page-table mechanism identifies the allocated pages and records status such as whether a page is present in memory or whether it is paged out to secondary storage. The workspace manager operates as an intelligent buffer manager and paging system that controls fair access to memory across all active workspaces of a given DBI user. A buffer pool manager holds the workspace pages that reside in memory.

The buffer pool manager supports multiple buffer replacement policies, which is important for database workloads that involve sequential access to data that is subsequently no longer needed. The two supported strategies are least recently used (LRU) and most recently used (MRU).[16] Finally, the workspace manager supports write-behind for newly allocated pages. This allows newly allocated pages that have been filled to be written asynchronously.

Join Processing Highly efficient processing of joins and unions is important in any commercial database; it is crucial for a multidatabase system. DBI supports nested loop join, index join, and hash join. In fact, DBI supports both a regular hash-join mechanism and a hybrid, hash-partitioned variant that is augmented with Bloom filtering.[17,18,19]

For both hash-join variants, the inner table rows are read asynchronously into a DBI workspace. This first pass is used to estimate whether or not to use the hash-partitioned variant. An exact estimate for the number of partitions to use is well worth the overhead of this initial pass.[20] In addition, a Bloom filter with 64 kilobits is populated as part of this pass. The inner table cardinality, an estimate for the outer table cardinality, and an estimate of the presently available memory are used to determine whether the simple hash-join technique is sufficient, or whether the use of the hybrid hash-partitioned join technique is warranted.

In general, hash-partitioned join processing is indicated when the inner table and its hash-table buckets do not fit in memory. In this case, both the build phase for the inner-table hash buckets as well as the probe phase of outer-table tuples against the inner-table hash buckets may incur massive amounts of random I/O. When the hash-partitioned variant is selected, the following steps are performed.

- Each partition receives a separate workspace.
- The inner table is partitioned first. During this partitioning step, a Bloom filter is generated from the join column of inner-table tuples and is applied when the outer table rows are partitioned. This results in a potentially massive reduction of the number of rows that are placed into the outer partitions, thus eliminating expensive I/O operations.
- The workspaces that hold the inner-table partition 1 and the hash-table buckets for that partition are aged LRU, which keeps them in memory for the join operation on the first partition pair.
- The workspaces that hold the remaining inner-table partitions 2 through (n) are aged MRU; these pages become immediately available for buffer replacement selection once they have been filled and their frames unpinned.
- Once the partitioning phase is complete, each pair of inner and outer partitions is joined starting with partition pair 1. The inner partitions are aged LRU, and the outer partitions are aged MRU to keep the inner partition tuples in memory.

The use of flexible buffer replacement strategies is crucial for good buffer cache behavior.

Parallelism DBI employs two types of parallelism: pipelined parallelism and independent parallelism.[8]

With hash-join processing, for instance, the outer table rows are read by separate DBI execution threads from the underlying database. This means that the outer table tuple stream is effectively generated in parallel with the probe phase processing of the hash-join operator on the inner table rows. The outer-table tuple stream is directed into the hash-join probe phase.

For UNION processing on partitioned views, the individual input streams to the UNION operator are generated by separate DBI execution threads. The streams are provided in parallel and independently to the UNION operator.

Stored Procedures Stored procedures provide a critical performance enhancement for client-server processing. They allow the DBA to encapsulate a set of SQL statements plus control logic. The client sends one message containing a stored procedure rather than several messages, each containing one SQL statement. This reduces processing delays that otherwise would be incurred due to network traffic.

DBI Server Configuration

In a standard DBI configuration, one execution process is created for each DBI client. As the number of clients increases, more and more operating system resources are consumed. The DBI server configuration addresses this problem.

Server Components A DBI server configuration includes minimally a monitor process, a dispatcher process, and a set of DBI executor processes. The monitor process supports on-line system management of the server configuration. One or more dispatcher processes manage all client communications context. Dispatchers route client messages to an appropriate DBI executor process through high-speed shared memory queues. Figure 12 shows a typical DBI server configuration.

Server Infrastructure In the DBI server environment, an ODBC client logically connects to a service object that provides access to a specific DBI database.[1] A service is instantiated by a pool of DBI executor processes that contain the DBI image. The amount of processes of the pool is configurable, both off-line and on-line. This allows the administrator to match the throughput requirements for a given DBI database with the appropriate amount of executor processes.

Multithreading DBI executor processes may presently be configured as session-reusable or transaction-reusable. Session-reusable means that a client is bound to an executor process for the duration of the entire database session. Transaction-reusable means that multiple clients may share the same executor process; a client is scheduled to a DBI executor for one transaction at a time.

Summary

The DB Integrator product contains many features that enable it to provide open, robust, and high-performance data access. DBI guarantees open data access by supporting de facto and de jure interface standards such as SQL92 and ODBC. Client-server connectivity is available over the DECnet, TCP/IP, and SPX/IPX transports. The MDI/DDI interface allows users to extend the use of DBI to gain access to any number of data sources.

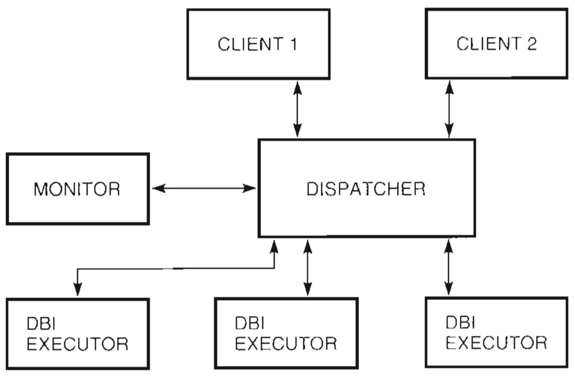


Figure 12
DBI Server Configuration

DBI provides robust data access by supporting heterogeneous query optimization, location transparency, global consistency, resolution of semantic differences, and security checks. The DBI query optimizer takes cost factors and capabilities into account to determine the optimal plan. A global catalog provides location transparency and operates as an autonomous metadata repository. Global transactions are coordinated through two-phase commit. Highly available horizontal partitioned views support continuous distributed processing in the presence of loss of connectivity. Definitions of views and stored procedures allow the user to hide semantic differences among the underlying databases. Finally, DBI enables security checks without interfering with the access controls specified in the underlying data sources.

DBI offers high-performance data access through a combination of sophisticated query optimization, advanced query execution algorithms, and efficient use of network resources. The query optimizer decomposes a distributed query by using as many features of the underlying database as possible and by employing state-of-the-art techniques such as query unnesting and partition elimination. The DBI query processor is capable of driving index joins and hybrid hash-partitioned joins. All intermediate data is cached I/O optimized. Connections to remote data sources are established solely on demand. Finally, parallel query execution is supported.

In the future, performance will continue to be an important factor for any data access product as will support for object-oriented data models. By combining data-integration technologies such as DBI with application-integration standards such as Object Request Brokers, a merger of data integration and application integration will be feasible.

Acknowledgments

The authors would like to recognize everyone who contributed to the DBI project. Jim Gray, Hal Berenson, Dave Lomet, and Gayn Winters helped to establish the product vision. Russ Holden and Dan Dietterich lent their technical guidance and project leadership. The DBI engineering team designed, implemented, and delivered the product on schedule. The DBI management team of Steve Serra, Rich Bourdeau, Arlene Lacharite, and Trish Pendleton contributed their commitment to delivering the vision. We would also like to thank the anonymous referees for their invaluable comments on the content and presentation of this paper.

References

1. *Microsoft Open Database Connectivity, Programmer's Reference, Version 2.0* (Redmond, Wash.: Microsoft Corporation, 1993).

2. *Information Technology—Database Language SQL, ANSI X3H2-92-154/DBL CBR-002* (New York: American National Standards Institute, 1992).

3. "Middleware: Panacea or Boondoggle?," *Strategic Analysis Report* (Gartner Group, July 5, 1994).

4. A. Sheth and J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, vol. 22, no. 3 (1990).

5. *Digital Data Distributor Handbook* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-HZ65FI-TE, 1994).

6. *Strategic Data Interface*, Version 3.1 (Maynard, Mass.: Digital Equipment Corporation, 1994). This internal DB integrator specification is not available to external readers.

7. *DEC Rdb Documentation Set for DEC Rdb Version 6.0* (Maynard, Mass.: Digital Equipment Corporation, 1994).

8. D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," *Communications of the ACM*, vol. 35, no. 6 (1992): 85–98.

9. *Digital DSRI Handbook, Version 5.1* (Maynard, Mass.: Digital Equipment Corporation, 1994). This internal document is not available to external readers.

10. *Digital Distributed Transaction Manager, OpenVMS Documentation, Version 5.5* (Maynard, Mass.: Digital Equipment Corporation, 1992).

11. *Distributed Transaction Processing: The XA Specification*, X/Open CAE Specification: C193, ISBN 1-872630-24-3 (1992).

12. P. Selinger et al., "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference* (1979).

13. P. Selinger and M. Adiba, "Access Path Selection in Distributed Database Management Systems," *IBM Research Report* (1980).

14. W. Kim, "On Optimizing an SQL-like Nested Query," *ACM Transactions on Database Systems*, vol. 7, no. 3 (1982).

15. U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers," *Proceedings of the 13th Conference on Very Large Databases* (VLDB), Brighton (1987).

16. M. Stonebraker, "Operating System Support for Database Management Systems," *Communications of the ACM*, vol. 24, no. 7 (1981): 412.

17. G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, no. 2 (1993).

18. B. H. Bloom, "Space/time Tradeoffs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7 (1970): 422–426.

19. M. Ramakrishna, "Practical Performance of Bloom Filters and Parallel Free-text Searching," *Communications of the ACM*, vol. 32, no. 10 (1989): 1237.

20. S. Christodoulakis, "Estimating Block Transfers and Join Sizes," *Proceedings of the ACM SIGMOD Conference* (1983).

Biographies

Richard Pledereder
Formerly a consulting software engineer in Digital's Software Products Group, Richard Pledereder was the system architect on the DB Integrator product family and contributed to the architecture and implementation of common DBI and Rdb features such as SQL stored procedures. Richard also initiated the architecture, design, and development effort of a multithreaded database server environment, which is now part of the DBI/OSF and Rdb/OSF products. He is now a software architect in the Distributed Products Group at Sybase, Inc. He received a B.S. and an M.S. in computer science from the Technical University Munich, Bavaria. Richard also collects tapes of operas by the Bavarian composer Richard Wagner.

Vishu Krishnamurthy
Vishu Krishnamurthy is a principal engineer in Digital's Database Integration and Interoperability Group, where he is currently the project leader for the DB Integrator product. Vishu was the technical leader for the metadata and catalog management components of DBI. Since joining Digital in 1988, he has held senior development positions in the Distributed Compiler Group, in the RdbStar project, and in the DEC Data Distributor project. Vishu holds a B.E. (honors) in mechanical engineering from the University of Madras and M.S. degrees in computer and information sciences and in mechanical engineering (robotics) from the University of Florida.

Michael Gagnon
Mike Gagnon joined Digital in 1981 and worked on the design and development of Digital's transaction processing and database systems. Mike contributed to the development of ACMS, Digital's transaction processing monitor for VMS systems, and more recently he contributed to the development of a distributed heterogeneous database system. When that system was refocused as the DB Integrator product, Mike led the team that produced the execution engine for all relational processing. Mike assumed project leadership responsibility for DBI version 1.0 and led the project through version 3.1. Mike is currently employed by Iris Associates, a subsidiary of Lotus Software.

Mayank Vadodaria
Formerly a principal software engineer in Digital's Database Integration and Interoperability Group, Mayank Vadodaria was the technical group leader for the query processing components of Digital's DB Integrator product family. He was also responsible for Digital's SQL development environment products. He has been instrumental in the design of many key features in the compilation and query optimization within DBI. Mayank holds a B. Tech. from the Indian Institute of Technology, Madras, and an M.S. in computer science from the University of Illinois at Urbana-Champaign. He is currently with Gupta Technologies.

ACMSxp Open Distributed Transaction Processing

Robert K. Baafi
J. Ian Carrie
William B. Drury
Oren L. Wiesler

Digital's ACMSxp portable transaction processing (TP) monitor supports open TP standards and provides an environment for the development, execution, and administration of robust, distributed, client-server applications. The ACMSxp TP monitor supports the Structured Transaction Definition Language, a modular language that simplifies the development of transactional applications. ACMSxp software is layered on the Open Software Foundation's Distributed Computing Environment (DCE) and supports XA-compliant databases and other resource managers by using the Encina toolkit from Transarc Corporation or Digital's distributed transaction manager (DECdtm) software. As a framework for DCE-based applications, the ACMSxp TP monitor simplifies application development, integrates system administration, and provides the additional capabilities of high availability, high performance, fault tolerance, and data integrity.

Transaction processing (TP) is a style of computing that guarantees robustness and high availability for critical business applications. TP typically involves a large number of users using display devices to issue similar and repetitive requests. The requests result in the accessing and updating of one or more databases to reflect the current state of the business.

The basic building block in a TP system is a transaction. A transaction is an indivisible unit of work that represents the fundamental construct of recovery, consistency, and concurrency. Each transaction has the properties of atomicity, consistency, isolation, and durability (ACID). These properties are defined as follows:

- Atomicity. Either all the actions of a transaction succeed or all fail. In case of failure, the actions are rolled back.
- Consistency. After a transaction executes, it must either leave the system in a correct state or abort and return the system to its initial state.
- Isolation. The actions carried out by a transaction against a shared database cannot become visible to other transactions until the transaction commits.
- Durability. The effects of a committed transaction are permanent.

A TP monitor manages and coordinates the flow of transactions through the system. Transaction requests typically originate from clients, are processed by one or more servers, and end at the originating client. When a transaction ends, the TP monitor ensures that all systems involved in the transaction are left in a consistent state.

The development of powerful desktop systems and advances in communications technology have fueled the growth of distributed client-server computing. The systems in a distributed environment may run different operating systems, possibly from different vendors. Business-critical applications may run under the control of different TP monitors. To coordinate their activities, TP monitors on heterogeneous systems need to conform to standards for open transaction processing.

Open standards for transaction processing have been adopted by the International Organization for

Standardization/Open Systems Interconnection (ISO/OSI), the X/Open initiative, and the Service Providers' Integrated Requirements for Information Technology (SPIRIT) consortium.[1,2] The X/Open initiative is a consortium of vendors whose purpose is to define standards for application portability. SPIRIT is a consortium of telecommunications service providers from the U.S., Europe, and Japan working under the general sponsorship of the Network Management Forum (NMF).[3-5] The goal of the NMF's SPIRIT consortium is to define standards for portability and interoperability across heterogeneous systems to be used as the basis for procurement.

The main standards for open transaction processing are

- X/Open distributed transaction processing (DTP), which is an architecture that allows multiple programs to share resources (e.g., databases and files) provided by multiple resource managers and allows their work to be coordinated. The architecture defines application programming interfaces and interactions among transactional applications, transaction managers, resource managers, and communications resource managers. The transaction manager and the resource manager communicate by means of the XA interface.[6]
- X/Open transactional remote procedure call (TxRPC), which allows an application to invoke local and remote resource managers as if they were all local. TxRPC also allows an application to be decomposed into client and server components on different computers interconnected by means of remote procedure calls (RPCs).
- SPIRIT Structured Transaction Definition Language (STDL), which is a block-structured language for transaction processing.[4,5,7] STDL provides transactional features including demarcation of transaction boundaries, transaction recovery, exception handling, transactional communications, access to data queues, submission of queued work requests, and invocation of presentation services.

Digital's Application Control and Management System/Cross-platform (ACMSxp) software product is a portable TP monitor that supports the open TP standards. It provides an environment for the development, execution, and administration of STDL applications. ACMSxp software is layered on the Open Software Foundation's (OSF's) Distributed Computing Environment (DCE) and supports multiple resource managers through Transarc Corporation's Encina toolkit on the UNIX operating system and Digital's distributed transaction manager (DECdtm) services on the OpenVMS operating system.[8] This paper describes the design of the ACMSxp TP monitor.

Application Development

ACMSxp applications are written using a combination of the STDL and traditional languages such as C and COBOL. STDL is a modular, block-structured language developed specifically for transaction processing. It is based on the ACMS Task Definition Language (TDL) and was developed as part of Nippon Telegraph and Telephone's (NTT's) Multivendor Integration Architecture (MIA) initiative.[9-11] The NMF's SPIRIT consortium subsequently adopted STDL.

STDL Language Overview

STDL provides transactional features including transaction demarcation, transactional remote procedure call, transactional task and data record queuing, transactional display management, transactional exception handling, and transactional working storage called workspaces.

STDL divides an application into three parts: presentation, transaction flow control, and processing, as illustrated in Figure 1. The presentation part interfaces with display devices using a presentation manager, such as Motif, Windows, or forms manager software. The transaction flow control part is written in STDL and controls the flow of execution, including transaction demarcation, exception handling, and access to queues. The processing part is written in traditional languages, such as C, COBOL, and SQL, and provides computation and access to resource managers such as databases and files.

The application functions in the three parts of the STDL application model are referred to respectively as presentation procedures, tasks, and processing procedures. The application functions are packaged into groups for the purposes of compilation and execution. The groups are referred to as presentation groups, task groups, and processing groups.

A group specification describes the functions in the group and their interfaces. The interface specification includes the arguments that are passed to the function and an indication of whether an argument is input only, output only, or both input and output. For a task, the interface specification also indicates whether the task begins a new transaction (NONCOMPOSABLE) or joins the caller's transaction (COMPOSABLE).

STDL variables are defined in constructs called workspaces. Workspaces may have the transactional attribute, thus allowing an application to coordinate internal data with the outcome of the transaction along with other resource manager participants. Workspaces have the scope of either PRIVATE or SHARED. A PRIVATE workspace is accessible to only a single task; a SHARED workspace is accessible to all tasks in a task group.

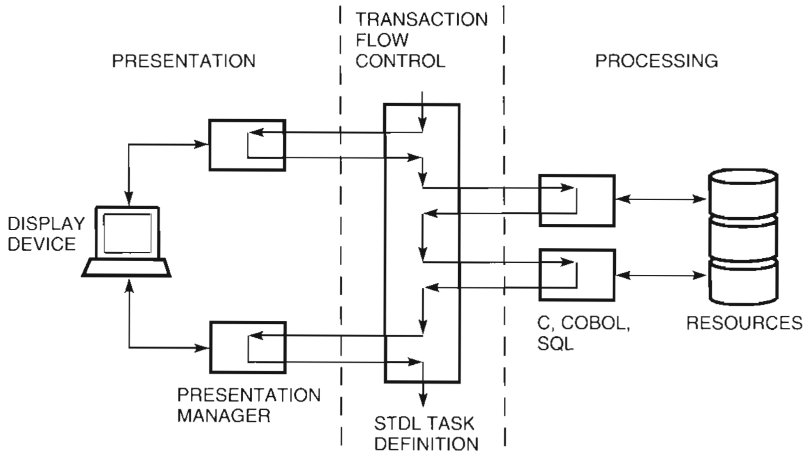


Figure 1
STDL Application Model

STDL supports two types of queues: record and task. Record queues provide a transactional, durable scratch pad facility for applications to store and retrieve intermediate results. Task queues provide a way of executing tasks independently of the currently executing task in both time and location. Storage of the task queue element on the task queue may or may not be conditional on the outcome of the currently executing task.

Sample STDL Application

Figure 2 shows a sample STDL application program. The sample program accepts an integer, increments it, and displays it. In addition, shared workspaces are defined in the task group to track the number of successful executions (*successes*) and the number of failed executions (*failures*). These operations all take place within the context of a transaction defined by task add1. If the transaction succeeds, the program increments *number* and the shared workspace *successes*. If the transaction fails, the program restores *number* to its initial state and invokes the exception handler. The exception handler then updates the shared workspace *failures*.

STDL Compiler

The STDL compiler simplifies the process of developing distributed client-server applications. It generates

```
RECORD arg1
      number INTEGER;
END RECORD;

TASK GROUP example1
      TASK add1
            TASK ARGUMENT IS arg1 PASSED AS INOUT;
END TASK GROUP;

TASK add1 ARGUMENT IS arg1 PASSED AS INOUT;
      WORKSPACES ARE        successes SHARED UPDATE RECOVERABLE,
                            failures SHARED UPDATE,
                            arg1 PRIVATE RECOVERABLE;
BLOCK WITH TRANSACTION
      PROCESSING
            COMPUTE successes = successes + 1
      PROCESSING
            COMPUTE number = number + 1
      EXCHANGE
            SEND RECORD number TO inscreen
END BLOCK
EXCEPTION HANDLER IS
      PROCESSING
            COMPUTE failures = failures + 1
END EXCEPTION HANDLER;
END TASK;
```

Figure 2
Sample STDL Application

all the code necessary for supporting the application in the distributed environment, including server initialization, namespace registration, namespace lookup, and application context propagation. This allows the application programmer to focus on the application problem at hand.

The ACMSxp STDL compiler translates STDL specifications into executable code. The compiler itself is written in the ANSI C programming language using POSIX 1003.1 library interfaces for platform portability; the generated code consists of only ACMSxp run-time service calls and DCE service calls.[12] To the application programmer, the ACMSxp STDL compiler looks much like a classical compiler. The STDL compiler reads source code, converts it to object code, and then links it to create an executable program. Figure 3 shows the elements written by the application programmer and the transformations required to create an executable program.

Internally, the STDL compiler consists of a series of steps that run under the control of a driver program. This processing takes place in the steps shown inside the dashed-line box of Figure 3. The STDL driver first reads STDL specifications in one pass and constructs internal structures that represent each STDL entity in the source file. Once an entity has been completely parsed and the syntax has been checked for errors, the driver generates intermediate files by translating

- STDL groups into ACMSxp client and server stubs and a DCE RPC Interface Definition Language (IDL) file
- STDL tasks into C code and ACMSxp run-time service calls
- STDL record definitions into C structures contained in C header files or COBOL copy files

After the STDL driver has generated all the intermediate files, it invokes the appropriate language processor to convert the files into object files. The DCE IDL compiler processes the IDL files, and the C compiler processes the tasks and the ACMSxp stubs. To keep the number of pieces visible to the application programmer within reason, the ACMSxp client and server stubs are combined with the DCE client and server stubs. The result is a collection of object files similar to those found in a conventional DCE application. The platform linker then combines the resulting files into an executable program.

The ACMSxp client and server stubs are similar in concept to the DCE RPC client and server stubs. The client stub is linked with other applications that invoke

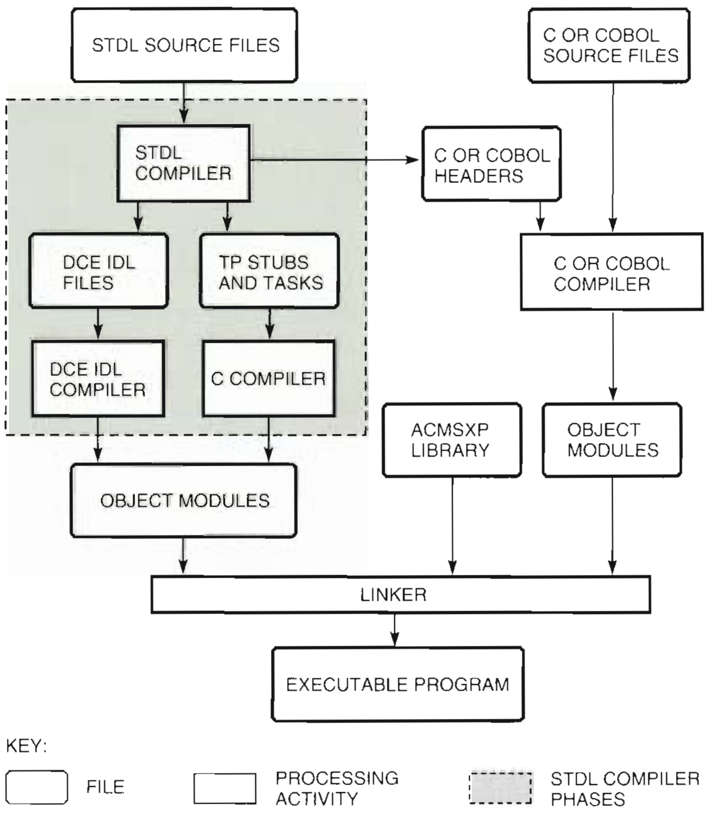


Figure 3
STDL Compiler Flows

this group's tasks or procedures. The server stub is combined with application code to create the application server image. The ACMSxp stubs call ACMSxp run-time services to add to the base DCE RPC services features such as transactions, failover and failback, and time-out.

Execution Environment

The ACMSxp run-time system provides an environment for executing and invoking STDL applications. It also provides services that allow components in the execution environment to be managed. The execution environment provides many services typically needed in TP environments, such as resource scheduling, fault tolerance, and queuing.

Process Model

The ACMSxp environment consists of client and server components. A TPsystem comprises multiple server components on a node that are managed as a unit. A given TPsystem has a globally unique name and is associated with only one node, but a node can have multiple TPsystems associated with it. A TPsystem contains a central process called the TPcontroller, which controls the components within the TPsystem. The processes in the execution environment are illustrated in Figure 4.

As the central point of control for the components within a TPsystem, the TPcontroller performs many functions, including license checking, starting and stopping servers, and monitoring server processes and restarting them when they terminate abnormally. It also receives administration requests and performs the requested operations, maintains information in shared memory for communication with server processes, and maintains key files for server authentication.

A task server executes STDL task group code and uses multiple threads in a single process to achieve concurrent execution (multithreaded). A processing server executes STDL processing group code and uses a pool of single-threaded processes to achieve concurrent execution (multiprocess).

System servers provide specific run-time services to the TPcontroller, task servers, and processing servers. The system servers include the event log server, the request queue server, and the record queue server. System servers are multithreaded.

Client processes invoke services provided by a TPsystem and its servers. An administration client (also referred to as the director) invokes administration services provided by the TPcontroller and system servers. An application client invokes application services provided by task servers. An application client can be a customer-written client or an ACMS Desktop client. A customer-written client can consist of code necessary to support a forms manager or device control such as an automatic teller machine or a gas pump. An ACMS Desktop client allows popular desktop systems such as the Macintosh, SCO's UNIX, Microsoft Windows, and Windows NT operating systems to be used to access services provided by ACMSxp application servers.

Run-time Services

The ACMSxp run-time system provides services required for the execution of client-server TP applications. The run-time services are highly modular and are layered on the services provided by the underlying transaction manager, DCE, operating system, network, and other services, as shown in Figure 5.

The run-time services integrate the services of the underlying platform and provide additional functionality. They export an application programming interface (API) called the transaction processing service interface (TPSI). The run-time services include

- Communication, which provides services for transactional and nontransactional communication

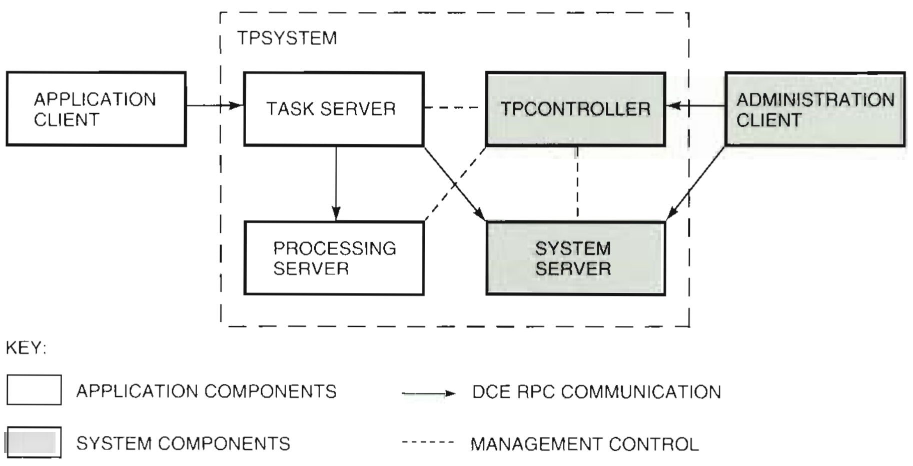


Figure 4
Processes in Execution Environment

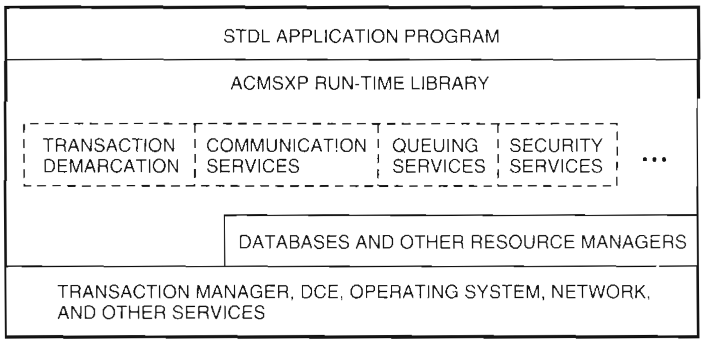


Figure 5
Modular Run-time Architecture

between clients and servers using DCE RPC. The supported transports are transmission control protocol/internet protocol (TCP/IP), DECnet OSI, and Fast Local Transport.

- Process management, which provides services for starting and stopping server processes, monitoring server processes for abnormal termination, and restarting new ones to maintain the specified number of processes.
- Thread context management, which provides services for creating, setting, and propagating thread context. Thread context includes request context, exception context, transaction context, and procedure context.
- Timer alert, which provides services for accumulating CPU time and transaction (elapsed) time.
- Transaction demarcation, which integrates with the Encina toolkit on the OSF/1 platform or the DECdtm software on the OpenVMS platform to provide distributed transaction support.
- Queuing, which provides services for request queuing and record queuing. Request queuing allows task requests to be queued for deferred invocation. Record queuing allows data records to be enqueued and dequeued.
- File management, which provides file management services for COBOL and C programs. It provides thread-based transaction semantics for STDL file access and handles opening and closing of files, file positioning, and file locking.
- Workspace management, which provides services for managing private and shared workspaces. A workspace is an STDL construct and represents an area of memory used for data storage and for arguments passed in a procedure call. A workspace can be recoverable or nonrecoverable.
- Security, which authenticates users and servers and provides access control, based on the DCE security service, for application invocation as well as management operations.
- Event posting, which provides services for writing events into a log. Logged events include error, security, status, audit, and trace events.
- Performance monitoring, which provides services for capturing performance measurement data.

Client-Server Communication

The ACMSxp communications services use OSF's DCE services for locating servers, invoking servers, and ensuring secure communications. The communications services maximize the efficiency of DCE service usage, provide robustness in the event of failure, and add distribution of transaction semantics to DCE RPC communications.

Figure 6 shows the elements and steps involved in the communication between a client and a server. The numeric annotations in the following discussion refer to the numbers that appear in the figure.

The STDL client application calls the server (1). The ACMSxp client stub issues run-time service calls (2) to initialize context blocks and to obtain a binding handle (i.e., server addressing information), and calls the DCE RPC client stub, passing context blocks and application data (3). The DCE RPC client stub marshals data and calls the server (4).

The DCE RPC server stub receives the call, unmarshals data, and calls the ACMSxp server stub (5). The ACMSxp server stub issues run-time service calls (6) to establish local context and to check security authorization, and calls the server application (7). The server application executes and returns the results to the ACMSxp server stub, which propagates any error information.

Transaction Processing Characteristics

The run-time system provides the TP monitor with characteristics such as high availability, load balancing, and high performance. Some of the mechanisms used to achieve these characteristics are discussed below.

Availability The run-time system provides failover and failback capabilities to enhance the availability of

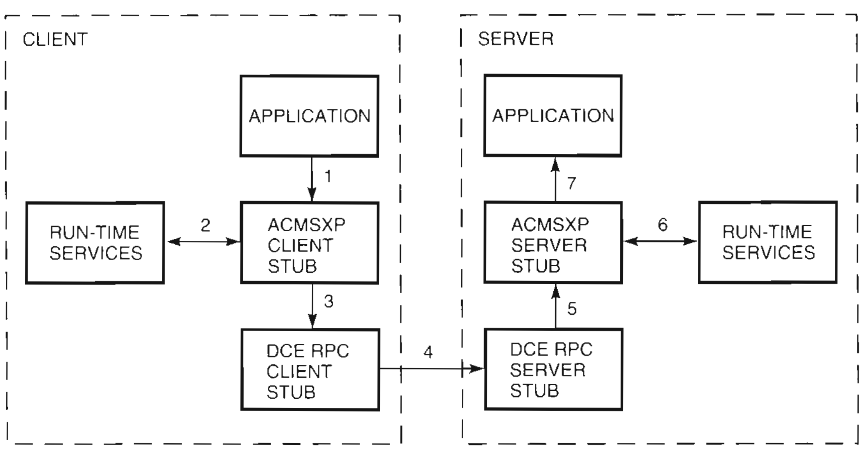


Figure 6
Client-Server Communication Flow

applications. Failover is the redirection of an RPC to an alternate server if the intended server is not reachable. The target server can be unreachable for many reasons, including loss of connectivity, application failures, and machine failures. Failback is the redirection of calls to the original server when it becomes available.

Failover and failback capabilities are supported for task servers but not for processing servers. The DCE cell directory service (CDS) namespace profile mechanism supports failover and failback. The system administrator configures the primary and alternate servers by placing them in the same namespace profile with different priorities. The server with the lower priority number is the primary server.

Run-time support for failover and failback is implemented in the client stub. Failover is attempted if an RPC fails and the returned error indicates that no work had been done by the called server in the current transaction. Failover is always attempted for a nontransactional RPC but is attempted for a transactional RPC only if this is the first call to the intended server in the transaction. The failover mechanism is optimized in three ways: by reconnecting, by pinging, and by checking the failed servers table. When a failure is detected, the failover mechanism attempts to reconnect to the server in case the failure was caused by intermittent communications problems. If the reconnect fails, the failover mechanism attempts to find an alternate server. When an alternate server is selected, it is pinged to ensure that it is reachable before being called with application work. If a server cannot be reached, it is recorded in a "failed servers" table and skipped on subsequent failover attempts.

Failback is attempted if the binding found is for an alternate server. Failback to the primary server is attempted even if the binding for the alternate server is good, as long as the failback timer has expired. The failback timer defaults to 300 seconds and can be set by an environment variable.

Load Balancing The ACMSxp run-time system can achieve load balancing for task servers through the DCE CDS. The DCE CDS group entry contains multiple server entries that provide the same interface. Locating a server by means of a group entry results in the random selection of one server in the entry. A combination of static load balancing and failover can also be implemented using DCE CDS functionality.

Performance Many parts of the ACMSxp system contain mechanisms that are designed to improve performance. A discussion of some of these mechanisms follows.

The server stub caches server bindings to improve performance. Server bindings are the addressing information that allows a client process to call a server process. Binding caching is a means of retaining the server addressing information for reuse. Reading the binding from the namespace can be time-consuming. For example, a DCE CDS namespace lookup requires a network connection to fetch the data from another process, which may be on a separate node. The cache of server bindings is shared among all the threads in the client process. This sharing provides a second order of performance improvement in that work previously performed on behalf of other threads can improve the performance of all threads by preloading the cache.

The scheduler subcomponent of the communications services allocates and deallocates server processes. It maintains a local namespace (also referred to as scheduler database) in shared memory to keep track of server process allocation. The use of the local namespace instead of DCE CDS improves the performance of RPC calls between task servers and processing servers, which are required to be in the same TPsystem.

The security service caches access control lists (ACLs) to improve performance. The TPcontroller maintains in shared memory the ACLs for managed objects that the ACMSxp TP monitor accesses at run time (e.g., procedures). The security service caches each object's ACL into the server process memory when the object is first accessed. The server process refreshes its cache if the entry in shared memory is updated.

System Administration

The distributed TP environment is inherently complex and requires effective system administration. The ACMSxp TP monitor provides the following system administration facilities for configuring, monitoring, and controlling components and resources within the ACMSxp run-time environment:

- Integrated user interface. The director (see the discussion of Figure 7, which follows) provides a consistent user interface for invoking management operations on all managed objects. The command line interface provides features such as command scripts, symbol substitution, session logging, default session parameters, and on-line help.
- Centralized distributed management. A single director can manage multiple TPsystems on the local or remote nodes using DCE RPC for communication.
- Extensibility. The object-oriented approach allows the ACMSxp TP monitor to represent managed resources in a consistent manner and to add new objects gracefully.

Management Model

The ACMSxp management model is object oriented and is based on the ISO/OSI standard for network and system management.[2,13] Figure 7 illustrates the elements of the model.

A director initiates management requests on behalf of the system administrator and serves as the interface between a system administrator and the objects being monitored and controlled.[14] A director consists of two parts: the user interface and the management service interface. The user interface interacts with the user and is either command line or graphical. The management service interface interacts with management agents. This interface provides services for creating an association for communication between a director and management agents, for initiating management requests, for returning results to the director, for canceling an outstanding request without waiting for completion, and for terminating an association normally.

The management protocol specifies both the mechanism for communication between a director and management agents and the model of interaction between them. The model specifies how requests and responses are passed between the director and the management agents, the processing of requests that involve wild card object instances, and the buffering of multiple responses to optimize performance. The ACMSxp TP monitor uses DCE RPC for communication between a director and management agents.

A management agent performs operations for a managed object. Each object class has a management agent that performs management operations for instances of that object class. The management agent receives a management request from the director, performs the requested operation, and returns the results.

Management Functions

Management operations that can be performed on managed objects are grouped into the following functional categories, as defined by the OSI management framework:

- Configuration management. Managed objects are instantiated, observed, and controlled. Persistent information about managed objects is stored in a configuration database.
- Fault management. Events generated by system operation are recorded in a log file. The contents of the log can be examined using a variety of search criteria.
- Performance management. Performance metrics are collected as the run-time system executes. The performance data is captured as attributes of managed objects and can be examined using the director.
- Security management. Principals are authenticated using the DCE. Access to system administration operations and application procedures is controlled using an ACL mechanism based on the DCE model.

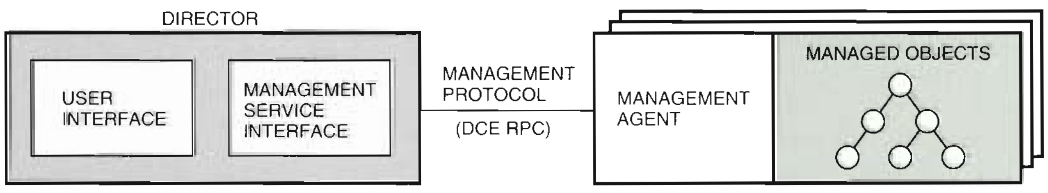


Figure 7
Management Model

Managed Objects

The resources in the ACMSxp environment that need to be managed are represented as objects. A managed object encapsulates the functionality of a real resource and specifies as visible only those aspects that need to be accessed by the manager. A managed object has the following properties:

- Attributes. Attributes are pieces of information that describe an object and represent internal state variables. Each attribute has a name and a value, which can be examined or modified as a result of a management operation. Examples of attributes are executable file name and processing state (for a server).
- Operations. Operations are activities that the manager can perform on the managed object. Operations allow the manager to examine attributes, modify attributes, and perform actions specific to the object. Examples of operations are create, delete, enable, disable, set, and show.
- Events. Events indicate the occurrence of normal and abnormal conditions. Examples of events are the detection of an error and the arrival at a threshold.
- Behavior. Behavior defines how attributes, operations, and events work together and how they affect the managed resource.

For naming purposes, managed objects are organized into a containment hierarchy. This hierarchy specifies which managed object is contained within another and reflects the containment relationship of all their corresponding managed resources. The top-level object in the structure, referred to as a global object, has a globally unique name. Objects contained within the global object are referred to as local objects and have names that are unique only within the context of their level in the structure. Table 1 describes the managed objects in the ACMSxp system.

Conclusion

The ACMSxp transaction processing monitor employs modular design techniques and a proven transaction processing architecture to provide a truly open, distributed transaction processing system. The STDL application development language, which the ACMSxp TP monitor supports, has been endorsed by an international standards consortium and has been implemented on other vendors' platforms. The layering on both the Open Software Foundation's Distributed Computing Environment software and the Encina toolkit provides a foundation of open distributed processing that has been accepted by the world's largest computer systems providers. The ACMSxp TP monitor provides a comprehensive set of facilities for managing the run-time environment. The object-oriented management approach results in a consistent representation of managed objects, a consistent user interface, a modular implementation, and extensibility.

Table 1
ACMSxp Managed Objects

Object Class	Description
TPsystem	A collection of system and application components and resources on a given node that is managed as a unit. A TPsystem is referred to as a global entity because it contains other managed objects and is not contained in any other managed object.
Server	A managed object that executes procedures. It encapsulates a collection of one or more operating system processes that execute the same program image.
Process	The basic unit scheduled by the operating system that provides the context within which a program image executes. It represents an operating system process.
Interface	A set of procedures that is provided by a server. It represents a DCE RPC interface and has a universally unique identifier (UUID) that distinguishes it from other instances.
Procedure	A structured sequence of instructions executed to achieve a particular result. It represents a DCE RPC operation.
Queue	A repository for storing an ordered collection of elements. The supported queues include a request queue, which contains submit requests, and a record queue, which contains data records.
Element	A single entry in a queue.
Log	A named repository where event records are stored.
Request session	The occurrence of a request at a particular TPsystem. A request is a series of operations invoked by a client program on behalf of a user and executed by one or more servers.

Acknowledgments

Throughout the course of this project, many people have participated in the design, implementation, and documentation of the product. The authors would like to thank all these people for their dedication and their contributions.

References

1. *X/Open Distributed Transaction Processing Reference Model*, ISBN 1-872630-16-2 (Reading, U.K.: X/Open Company Ltd., 1991).

2. *Information Processing Systems—Open Systems Interconnection—Basic Reference Model—Part 4: Management Framework*, ISO/IEC 7498-4:1989 (Geneva: International Organization for Standardization, 1989).

3. *SPIRIT Platform Blueprint (SPIRIT 2.0, vol. 1)*, ISBN 1-85912-059-8, Document No. J401 (Reading, U.K.: X/Open Company Ltd., 1994).

4. *SPIRIT STDL Language Specification (SPIRIT 2.0, vol. 3)*, ISBN 1-85912-063-6, Document No. J403 (Reading, U.K.: X/Open Company Ltd., 1994).

5. *SPIRIT STDL Environment, Execution and Protocol Mapping (SPIRIT 2.0, vol. 4)*, ISBN 1-85912-064-4, Document No. J404 (Reading, U.K.: X/Open Company Ltd., 1994).

6. *X/Open CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification*, ISBN 1-872630-24-3, Document No. C193 or XO/CAE/91/300 (Reading, U.K.: X/Open Company Ltd., 1994).

7. P. Bernstein, P. Gyllstrom, and T. Wimberg, "STDL—Portable Language for Transaction Processing," *Proceedings of the Nineteenth International Conference on Very Large Databases*, Dublin, Ireland, 1993.

8. W. Laing, J. Johnson, and R. Landau, "Transaction Management Support in the VMS Operating System Kernel," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1991): 33–44.

9. T. Speer and M. Storm, "Digital's Transaction Processing Monitors," *Digital Technical Journal*, vol. 3, no. 1 (Winter 1994): 18–32.

10. *Multivendor Integration Architecture, Division 1, Overview, Technical Requirements*, TR550001 (Tokyo, Japan: Nippon Telegraph and Telephone Corporation, 1991).

11. E. Newcomer, "Pioneering Distributed Transaction Management," *Bulletin of the Technical Committee on Data Engineering*, vol. 17, no. 1 (March 1994).

12. *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Interface (API) [C Language]*, IEEE 1003.1-1990 (New York: The Institute of Electrical and Electronics Engineers, 1990).

13. M. Sylor, F. Dolan, and D. Shurtleff, "Network Management," *Digital Technical Journal*, vol. 5, no. 1 (Winter 1993): 117–129.

14. C. Strutt and J. Swist, "Design of the DECmcc Management Director," *Digital Technical Journal*, vol. 5, no. 1 (Winter 1993): 130–142.

Biographies

Robert K. Baafi
A principal software engineer in the Transaction Processing Engineering Group, Robert Baafi is the primary architect for the system management component of the ACMSxp transaction processing monitor. Prior to joining Digital in 1989, he was the project leader for Cullinet Software's IDMS-DC transaction processing monitor. Bob received a B.S. in electrical engineering from the University of Connecticut in 1971 and an M.S. in information systems from Lehigh University in 1973. He is a member of ACM, Tau Beta Pi, and Eta Kappa Nu.

J. Ian Carrie
Ian Carrie is the project leader for the ACMS Desktop product. He is a member of the Transaction Processing Engineering Group. Since joining Digital in 1989, Ian has also contributed to the ACMSxp transaction processing monitor. He worked on the STDL compiler code generator, language run-time support, file support, and Encina transaction manager integration. In earlier work, he was employed by Cullinet Software as a project leader in the IDMS/R database product's Communications Group. Ian holds a B.A. (1980) in computer science/managerial studies from Rice University. He is a member of ACM.

William B. Drury
Bill Drury is currently employed by Stratus Computer as Engineering Manager, Transaction Processing and System Performance. While a consulting engineer at Digital, he led the design and development of the ACMSxp transaction processing monitor. He presented the product at numerous technical forums, including the STDL Implementors' Workshop, DECUS, DECORUM '94 (Transarc Corporation's user group), and the OSF DCE Developers' Conference. He also contributed to the specification of the Multivendor Integration Architecture (MIA) on which the ACMSxp product is based. Bill received B.S.E.E. (1982) and M.S.E.E. (1986) degrees from Ohio University.

Oren L. Wiesler
Presently, Oren Wiesler is a Factory Integration Manager at PRI Automation, a manufacturer of automation equipment used in semiconductor manufacturing. At Digital, he was a principal software engineer in the Transaction Processing Engineering Group. He led the ACMSxp for OSF/1 AXP version 2.0 effort and the support team for ACMSxp for OpenVMS VAX version 1.0, and contributed to the ACMSxp run-time system. Earlier, Oren worked in a processor hardware group. He received a B.S.E.E. from Worcester Polytechnic Institute in 1984 and holds two patents: one related to dynamic control of simultaneously switching outputs, the other on interleaved control store addressing.

Norman G. Depledge
William A. Turner
Alexandra Woog

An Open, Distributable, Three-tier Client-Server Architecture with Transaction Semantics

This paper describes a distributable, three-tier client-server architecture for heterogeneous, multivendor environments based on the integration of Digital's ObjectBroker and ACMSxp transaction processing monitor products. ObjectBroker integration software provides the flexibility to decouple the tight association between desktop devices and specific legacy systems. The ACMSxp transaction processing monitor provides the transaction semantics, system management, scalabilty, and high availability that mission-critical production systems require. Combining these technologies and products in a three-tier architecture provides a strategic direction for the development of new applications and allows for optimal integration of legacy systems. The architecture complies with industry standards, which facilitates vendor independence and ensures the longevity of the solution.

Almost all large global enterprises have developed separate systems to address specific business needs. Frequently, these systems are on disparate platforms from different vendors. Users may have to log in to several systems in order to process a single service request from a customer. To improve customer service and develop new products, new applications must integrate existing environments and must be capable of accessing and integrating data from existing platforms.

End users may be faced with an array of inconsistent and incompatible user interfaces that are difficult to learn to use. This source of inefficiency directly impacts the level and cost of service provided to customers and the time-to-market for new products and services.

An analysis of the above problems leads to some fundamental conclusions about existing business systems in large enterprises. Generally, the in-place applications are mission-critical legacy systems that record transactions performed by the businesses. These systems demand superior transactional integrity and operational reliability. They serve hundreds to thousands of users and yet provide good response at high levels of performance. Systems designers will not introduce changes to them that would compromise these exacting requirements. Consequently, enterprises do not readily replace their legacy systems but, instead, look for other solutions that integrate them with new systems.

To improve the effectiveness of existing legacy systems, major enterprises are seeking to reengineer the user interface. The goal is to reface the applications with a modern, consistent, easy-to-use interface that directly reflects the users' and customers' needs. The new interface must be fully articulated; that is, it should allow any desktop to access any permitted application, regardless of its location or the platform on which it is running. The solution should allow the composition of new compound business functions by combining existing application transactions from multiple legacy systems and possibly new or downsized applications. The new user interface should accomplish this without disrupting the level of service provided to the users.

All these requirements indicate the need for an intermediate architectural layer that provides for isolation, switching, transaction semantics, composition of function, and location transparency. The resultant architecture has three tiers: the clients, the intermediate layer, and the existing legacy systems and new servers.

Such an architecture is expected to last a considerable number of years. It is, therefore, essential that the architecture be based on modern but stable technologies and be flexible enough to accommodate technology evolution.

The Three-tier Architecture

The three-tier architecture consists of the following separate layers of systems and software:

1. Clients
2. Transactional middleware
3. Systems of record (legacy systems and new systems)

The attributes of the proposed intermediate layer make this three-tier architecture more flexible than traditional two-tier client-server architectures.

Tier 1 systems (clients) provide a desktop graphical user interface (GUI) to the end users. These systems have seamless access to a set of abstract transaction services in a location-transparent manner through an object request broker (ORB). The interface between tier 1 and tier 2 operates in a client-server manner. The security services in the ORB provide a security perimeter around the client.

Tier 2 is the middleware layer that provides application services to the clients. These services appear to the clients as business functions and may be transactional in nature.

A single tier 2 business function can be composed of one or more transactions residing in different applications on tier 3. The location and native interfaces of these applications are hidden from the tier 1 desktop GUI clients by the tier 2 middleware.

Tier 2 is designed to support the evolution of the application interfaces and protocols between tier 2 and tier 3. These interfaces will change as new technologies such as the Open Software Foundation's (OSF's) Distributed Computing Environment (DCE) and the Object Management Group's (OMG's) Common Object Request Broker Architecture (CORBA) mature and become more widely available, and as tier 3 applications are modified or new ones added.[1,2] Figure 1 shows the disposition of functions with intertier communications paradigms.

A Standards-based Architecture

Digital implemented the three-tier architecture using standards-based software to offer the highest level of interoperation with systems offered by other standards-compliant vendors. Standards compliance also facilitates the porting of applications across platforms.

The standards organizations most relevant to this architecture are

- International Organization for Standardization (ISO)
- American National Standards Institute (ANSI)
- Open Software Foundation
- Object Management Group
- X/Open Company Limited
- Nippon Telegraph and Telephone's (NTT's) Multivendor Integration Architecture (MIA) and the Network Management Forum (NMF's) Service Providers' Integrated Requirements for Information Technology (SPIRIT), together referred to in this paper as MIA/SPIRIT[3,4]

ISO and ANSI are true standards bodies. The other organizations are either influential industry consortia aimed at defining common standards for important emerging and maturing technologies driven by user needs or customer-driven consortia chartered to define common purchasing standards backed by substantial purchasing power.

OSF's Distributed Computing Environment

The Open Software Foundation's Distributed Computing Environment is an important standard that defines a set of services and tools that support the creation, use, and maintenance of client-server applications in heterogeneous multivendor environments. The OSF has defined and assembled this technology; Digital is a major provider of components. As illustrated in Figure 2, the components of OSF's DCE are

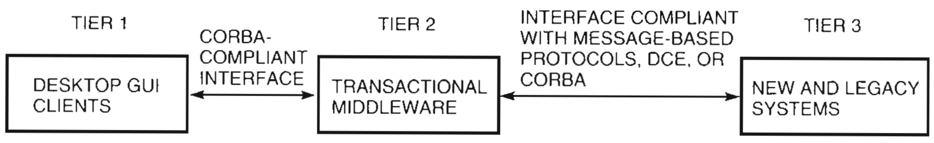

TIER 1
DESKTOP GUI CLIENTS
CORBA-COMPLIANT INTERFACE
TIER 2
TRANSACTIONAL MIDDLEWARE
INTERFACE COMPLIANT WITH MESSAGE-BASED PROTOCOLS, DCE, OR CORBA
TIER 3
NEW AND LEGACY SYSTEMS

Figure 1
Disposition of Functions with Intertier Communications Paradigms

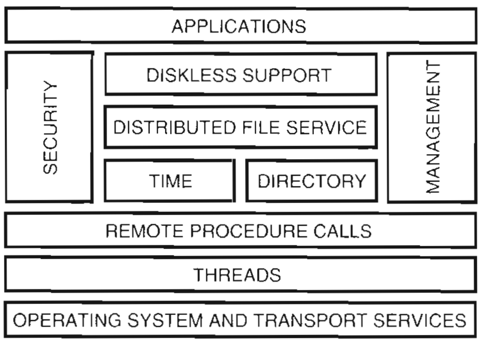


Figure 2
OSF's Distributed Computing Environment

- Remote procedure calls (RPCs)
- DCE threads, which is a standardized multithreading service
- Distributed time service, which synchronizes clock time across globally distributed systems
- Cell directory service (CDS), which provides authentication, access control, and encryption, and uses a Kerberos-based private key security model
- Global directory service, which provides directory services between cells using the X.500 standard
- Distributed file service, which provides location-transparent access to files across a network

DCE has been rapidly adopted as a technology for distributed systems and is now available on a large number of vendor platforms, including Digital, IBM, Hewlett-Packard, Sun, and Microsoft.

CORBA

The Common Object Request Broker Architecture is a standard specification for the central communication and integration of software objects at the enterprise level and across enterprises. CORBA and its specification were developed by the OMG, a consortium of information systems vendors, including Digital, Hewlett-Packard, HyperDesk, Symbios Logic (formerly NCR), Object Design, and SunSoft.[5] Digital's CORBA-compliant product, namely ObjectBroker integration software, has been ported to the industry's leading range of platforms.[6]

The ObjectBroker product reduces the time and costs associated with providing access to critical business applications across multivendor platforms. It allows legacy applications to be integrated into heterogeneous client-server environments without source code changes.

Microsoft Corporation has developed a parallel approach as evidenced in its Object Linking and Embedding (OLE) software, which is focused on integrating objects in a desktop environment.[7] Microsoft and Digital are working to integrate the CORBA and OLE software into a combined architecture called the Common Object Model (COM), which allows the full interoperation of applications developed under either constituent architecture.

X/Open Distributed Transaction Processing

The X/Open distributed transaction processing (DTP) committee is defining standards for DTP systems that use flat transactions. In Figure 3, the TX interface allows applications to coordinate global transactions via the transaction manager (TM); the XA interface connects the TM to resource managers (RMs), typically relational databases or file systems; and the XA+ interface connects the TM to communications resource managers (CRMs). The interface between an application and a CRM is specific to the CRM type, of which three are defined.

- Transactional remote procedure call (TxRPC), which is derived from the work led by Digital for the MIA/SPIRIT remote task invocation protocol (discussed in more detail later in this section).

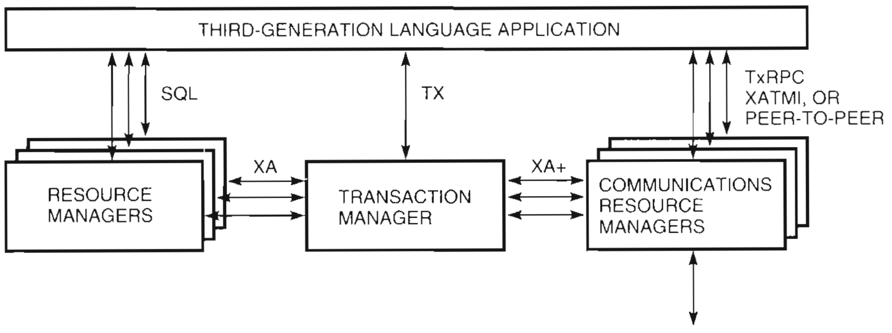


Figure 3
X/Open Distributed Transaction Processing Model

- XATMI, which is a non-RPC-based client-server that originated with Unix System Laboratories' transaction processing monitor for the UNIX operating system, namely the Tuxedo product.
- Peer-to-peer, by which messages are exchanged between applications. The messages are sent and received in an order based on prior agreement between the implementers of the applications. Peer-to-peer uses Common Programming Interface for Communications (CPI-C), which is derived from IBM's System Network Architecture (SNA) message-based protocol of the same name.

MIA/SPIRIT

MIA is a software architecture developed by a consortium of five vendors under the sponsorship of NTT: Digital, IBM, Fujitsu, Hitachi, and NEC. MIA adopted existing industry standards and defined standards in areas where none were available. One of the areas most lacking in standards was DTP. NTT requested technology proposals and received responses from all the vendors in the consortium. Digital submitted its Application Control and Management System (ACMS) transaction processing monitor model and was selected to lead the development of the specifications because of ACMS' modern, highly structured model and transaction processing application programming interface (API).

MIA achieves application portability and interoperability across a variety of vendor operating systems and platforms by using standardized APIs as integrative constructs and by using standardized systems interconnection interfaces (SIIs) for communication.

Two significant MIA standards that Digital contributed are

- Structured Transaction Definition Language (STDL), which is a high-level programming language suited to transactional client-server programming[8,9]
- Remote task invocation (RTI), a service definition and protocol for RPCs that are in a multivendor environment and that use the two-phase commit protocol

As a follow-on to NTT's MIA, the work in the field of transaction processing standards has passed to the SPIRIT consortium, which is managed by the Network Management Forum. NMF's list of members includes telecommunications service providers, such as AT&T, BT, Deutsche Telekom, ETIS (itself a consortium that represents 27 European Postal, Telegraph, and Telephone Administrations), France Telecom, KDD, Telecom Italia, and Telefonica; computer vendors, such as Digital, Hewlett-Packard, Fujitsu/ICL, Hitachi, IBM, NEC, Siemens Nixdorf, and Unisys; and software vendors, such as Microsoft and Oracle. The goal of the SPIRIT consortium is to produce a common, agreed-upon set of specifications for a general-purpose computing platform for the telecommunications industry by July 1995. The combined annual computing expenditures are estimated to exceed $20 billion.

MIA/SPIRIT standards are working their way into international standards bodies. X/Open and the NMF have extended their collaborative agreement to include the work of SPIRIT in acknowledgment of the difficulties that diverging standards would create. X/Open publishes the SPIRIT documentation alongside its own CAE specifications and guides. Furthermore, after conducting a survey of major transaction processing users, X/Open recently voted to use its fast-tracking process to accelerate progress in the adoption of STDL as an X/Open standard.

Digital delivered a platform that supports STDL in January 1993, IBM offered STDL on the CICS platform in the second quarter of 1993, and Hewlett-Packard has made STDL available on Transarc Corporation's Encina transaction processing monitor. NEC, Hitachi, and Fujitsu have already shipped STDL platforms. Unisys plans to demonstrate a SPIRIT platform with STDL in October 1995.

In July 1994, an interoperability demonstration using STDL was conducted successfully in Tokyo, Japan. The demonstration, which also included RTI, involved systems provided by Hewlett-Packard and Fujitsu on Transarc Corporation's Encina transaction processing monitor, Digital on its Application Control and Management System/Cross-platform (ACMSxp) transaction processing monitor, and IBM on both the MVS/CICS and OS/2 platforms.

Architecture Components

Figure 4 illustrates the overall three-tier client-server architecture. This section discusses the various components.

Tier 1 Desktop Environment

The architecture must provide for the connection of a wide variety of desktop platforms to the server layer, i.e., the tier 2 middleware services. This connection must be accomplished in a secure, extensible, reliable, and location-transparent manner. Standards-based solutions are always desirable and more effective over the multiyear life of an enterprise-wide solution. Digital therefore selected its CORBA-compliant ObjectBroker software as the mechanism to connect tier 1 clients to tier 2 middleware servers.

CORBA provides a flexible approach to developing a distributed application by decoupling the client and

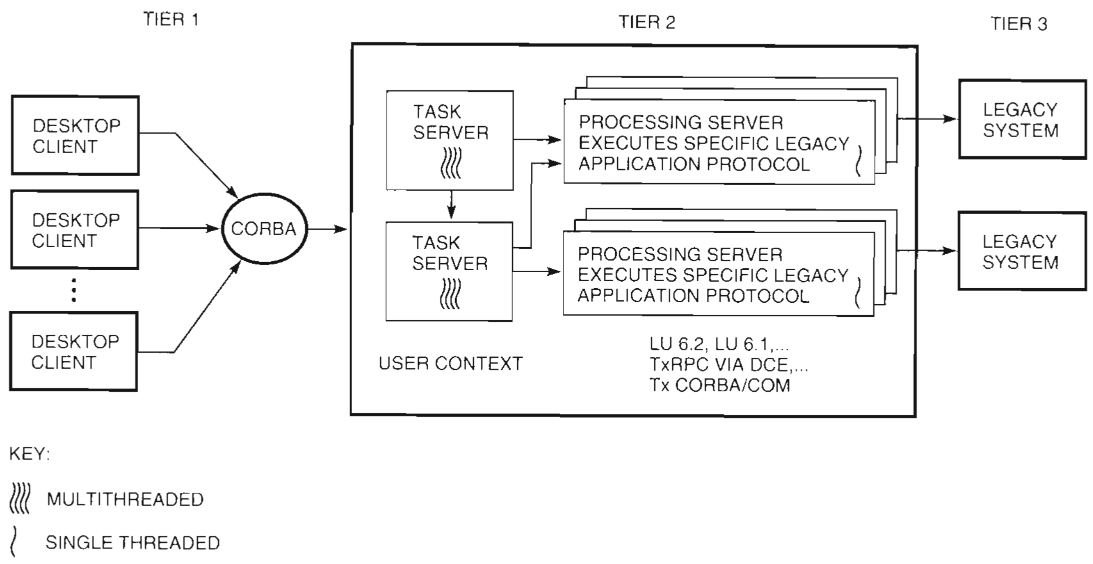


Figure 4
Overall Three-tier Client-Server Architecture

server portions of the application. CORBA specifies a common set of interfaces that allows client programs to make requests to and receive responses from server programs without direct knowledge of the information source or its location. CORBA defines the ORB as an intermediary between clients and servers that delivers client requests to the appropriate server and returns the server responses to the requesting client. Figure 5 shows how the ORB allows a client application to request a service without knowing where the server is located or how it will fulfill the request.

In the CORBA model, client applications need to know only what requests they can make and how to make the requests; they do not need to be coded with any implementation details about the server. Server programs need to know how to fulfill the requests but not how to return information to the client program. Clients using objects to request a service do not need to know which server will fulfill that request. The server fulfilling the request does not need to know which client initiated the request. The GUI clients can be developed using any tool that provides a call-level interface or an object-oriented interface to CORBA-compliant client services on the specific platform.

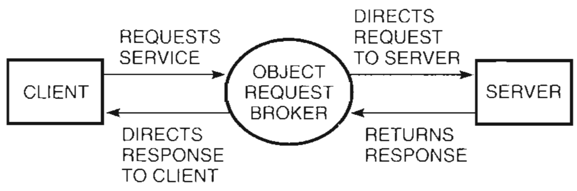


Figure 5
CORBA Client-Server Request/Response Flow

Communications are conducted through RPCs. The RPCs are carried over a network transport, e.g., a transmission control protocol/internet protocol (TCP/IP) or a DECnet transport. The RPC connects with ObjectBroker's ORB, which then reroutes the RPC directly to the selected service instance.

Digital expects future versions of its CORBA-compliant ObjectBroker product to support OSF's DCE and thus provide standards-based directory, security, and RPC services. DCE provides rigorous security services for authenticating users, granting privileges, and controlling access to important networked resources. These services are based on the highly secure Kerberos model, which is the standard security model for many financial institutions and a major reason why they have standardized on DCE. All interaction from tier 1 clients must go through the Kerberos-based DCE security perimeter. Desktop and mobile computer users log in to the DCE cell to gain their credentials for performing their business. DCE authenticates users and grants them the appropriate privileges and controlled access to the authorized business functions. No clear-text passwords are required, even for mobile users who access the middleware layer by means of dial-up lines. Remote or mobile users are able to perform DCE login over a serial line internet protocol (SLIP) connection. Confidentiality is ensured by data encryption.

Tier 2 Middleware Services

The tier 2 middleware of this architecture is founded on the ACMSxp transaction processing monitor. The ACMSxp software product for transactional applications conforms to the X/Open DTP and MIA/SPIRIT standards previously described. The software is layered on DCE and the transaction manager and Structured File Services of the Encina toolkit. The primary API to the ACMSxp product is STDL, as defined by MIA/SPIRIT.

STDL is used to define the control flow and transaction demarcation of applications in a highly structured and modular fashion. In addition, STDL supports the features needed for RPC-based DTP.

The ACMSxp software is structured in a three-part model, which is shown in Figure 6. The model separates client presentation functions from transaction flow control and data access and processing functions, which are typically SQL database code or communications code to access legacy systems. A very important feature of ACMSxp is that the STDL compiler generates complete DCE client-server stubs, thus freeing the developer from having to perform complex programming at the DCE level.

ACMSxp applications comprise clients that call application task servers using DCE RPCs. The application task servers execute the STDL programs in DCE multithreaded processes, maintaining the context of the users and performing the control flow and transaction semantics (two-phase commit protocol). Application tasks are free to call other tasks in the same group or in remote task groups that are reached through DCE directory services, either transactional (composable) or nontransactional (noncomposable). Tasks can also call processing procedures that are grouped in single-threaded procedure servers. Processing procedures normally provide data access, typically by means of C, COBOL, and SQL in conjunction with X/Open DTP-compliant databases. Processing procedures are also ideal for applications that communicate to legacy systems by means of message-based protocols, such as IBM's SNA Logical Unit 6.2 (LU 6.2).

Figure 7 shows client tasks accessing data by means of Digital ACMSxp servers. Note that ACMSxp supports and manages pools (groups) of servers in a given class, i.e., servers that provide the same set of services. Through the system management interface, operators can set and change dynamically the desired number of servers in a given pool. If a server fails, ACMSxp system management logs the event and automatically starts and activates a replacement server.

Digital's ACMSxp transaction processing monitor, in conjunction with the ObjectBroker software, is used to implement the tier 2 middleware layer. ACMSxp clients are no more than simple business function call statements stripped of DCE programming by the STDL client run-time services. ACMSxp clients are encapsulated in wrappers to make them CORBA objects accessible through the ObjectBroker software. These wrappered client processes can be declared as servers to the sophisticated ACMSxp transaction processing monitor system management. The system management can then be instructed to set dynamically the desired number of instances, to maintain that number in the event of failures, and to raise alerts. These ACMSxp client processes are wrappered to include the code that registers them as objects with the ORB as instances of CORBA implementations. When activated by ACMSxp system management, these processes register with the ORB and wait for requests for the tasks that they invoke. When stopped by ACMSxp system management, the processes deregister from the ORB.

This architecture can be instantiated using Digital's CORBA-compliant ObjectBroker product with the ACMSxp transaction processing monitor, presenting the desktop clients on tier 1 with an object-oriented interface to business functions executing on tier 2. As shown in Figure 4, ObjectBroker is substituted for CORBA and tier 2 is implemented on the ACMSxp transaction processing monitor. An unlimited number of tier 2 nodes can be configured dynamically, started, and stopped. Transactional business functions can be distributed and replicated across the nodes as required. The clients gain access to these business functions in a completely location-transparent manner. The interface to legacy systems is provided by

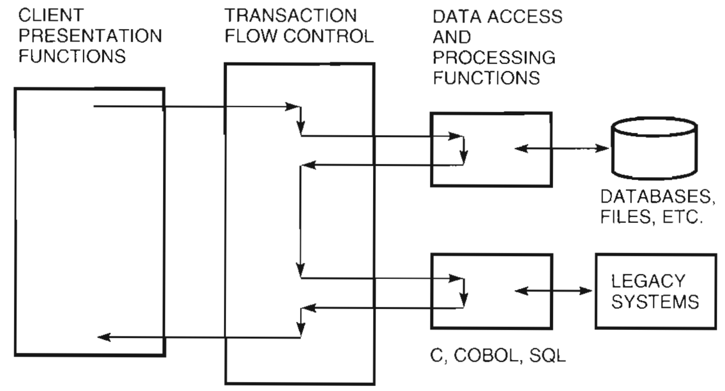


Figure 6
ACMSxp Three-part Model

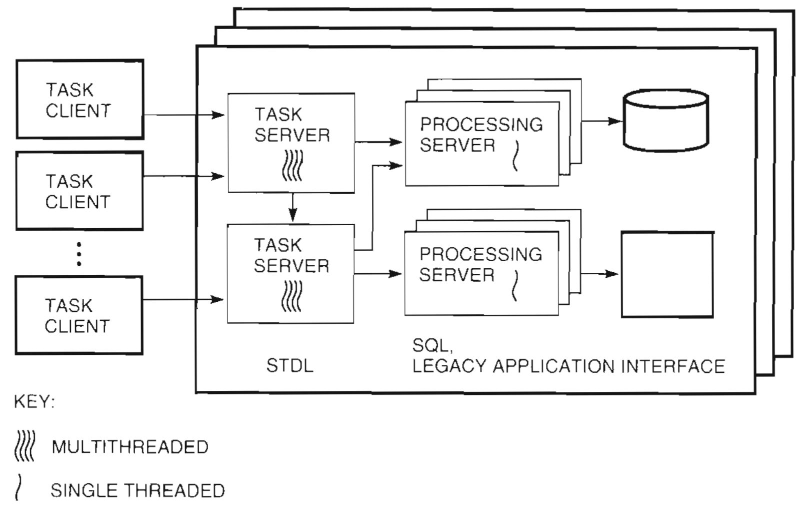


Figure 7
ACMSxp Application Components

application code that resides in processing servers.

The business functions are written as STDL tasks and can be composed of multiple legacy application transactions. When tier 3 applications support standards-compliant TxRPC, transactions can be called directly as tasks in STDL from the tier 2 business functions.

Security into tier 2 is handled by the ObjectBroker software. Within tier 2, security is enforced according to the rules of OSF's DCE. Security between tier 2 and tier 3 is mandated by the rules of each specific legacy system.

To provide operational support for production applications, sophisticated system management features were built into the ACMSxp product. A system management interface is available to any authorized operator on any node in the DCE cell. Through a single director, all ACMSxp objects can be managed in multiple transaction processing systems on all nodes in the network. The managed objects include transaction processing systems, event logs, request sessions, servers, processes, interfaces, and procedures. For example, system managers can examine and change the properties and execution state of servers. The number of instances of a given server class can be set and changed dynamically without stopping the system. ACMSxp system management can be induced to adopt servers that are normally external to its domain, such as the ObjectBroker method servers that provide the connection between the desktop clients and the transactional task servers in the ACMSxp product.

Tier 3 Legacy Application Interfaces

Intercommunications issues related to the differences between hardware and software architectures on disparate platforms are addressed by technologies such as DCE. DCE supports RPCs that enable applications on different platforms to interoperate by means of simple call statements with fully typed arguments. Data type differences between hardware architectures are bridged by the marshaling process that converts data to a canonical form and then to the target form as a normal process. Message-based protocols, such as LU 6.2, cannot adequately deal with mixed data types and place a burden on the application programmer in a multivendor environment.

The advent of reduced instruction set computer (RISC) architectures has exacerbated these problems. Gaps are frequently left in memory between variables in structures and records that contain mixed data types. These gaps in buffers, when processed by compilers on RISC machines, render the buffers unmappable unless redundant filler variables are added to the structure definitions.

Each legacy application method is encapsulated in an ACMSxp server class that is invoked transactionally by a simple STDL call. Thus, the developer of the STDL transactional business functions is shielded from the complexities of the native interface to the legacy data. This approach permits future update of the method without affecting the existing business functions.

The designer must select the most appropriate communications protocol for each tier 3 legacy system. Whenever possible, an application interface should be selected that avoids the so-called "screen scraping" techniques, in which the application emulates a user interacting with existing terminal screen forms.

For IBM mainframe systems, the SNA LU 6.2 protocol with Syncpoint Level 1 or 2 is often appropriate for interoperating with IBM transaction processing environments. This protocol may also be the appropriate choice for legacy systems from other vendors. If the application message protocol is designed in a manner that simulates a simple procedure call, future migration to an RPC model will be simplified.

Recently, IBM has made DCE available on MVS—OpenEdition and has provided application support for both the CICS and Information Management System (IMS) transaction processing environments. This feature allows DCE client programs to invoke transactions on the IBM mainframe by way of a DCE application server provided by IBM. An appropriate DCE client could be included in a data access processing procedure of an ACMSxp processing server as an alternative to SNA LU 6.2.

It should be noted that the desired throughput level for a given legacy system connection can be adjusted dynamically. An operator can use the system management of the ACMSxp transaction processing monitor to reset the number of active servers in the pool that implements that connection. Also, any number of tier 2 nodes can be configured to provide that service within the middleware layer. New nodes can be placed in service without interrupting currently running nodes.

Summary

A three-tier, object-oriented client-server architecture that includes an open systems transaction processing monitor can provide a basis for connecting users and customers to existing enterprise transaction processing systems by means of reengineered desktop systems that support GUIs. This approach provides

- A clear separation of function, i.e., client activities are separate from middleware control and management functions
- Data location transparency
- Location transparency for application interfaces and topological independence
- A means of defining new business functions by compounding existing transactions on different platforms, regardless of location
- Flexibility to support the continuous evolution of systems without disruption to end users
- Resilience to enhance overall availability
- Unrestricted scaling of the system (through replication of components) for performance adapted to the business growth
- A set of reusable objects to the tier 1 client

References and Note

1. *Open Software Foundation DCE Application Development Guide,* Document No. ED-DCEAPDEV1-1092-2 (Cambridge, Mass.: Open Software Foundation, 1993).

2. *Common Object Request Broker Architecture Specification,* draft 29, revision 1.2 (Framingham, Mass.: Object Management Group, Document No. 93-12-43, December 1993).

3. *Multivendor Integration Architecture Technical Requirements, Division 1, Overview* (Tokyo: Nippon Telegraph and Telephone Corporation Technology Research Department, Order No. TR550001, 1991).

4. *X/Open Consortium Specification, SPIRIT Platform Blueprint* (SPIRIT Issue 2.0, Volume 1.0), ISBN 1-85912-059-8 (Reading, U.K.: X/Open Company Ltd., Network Management Forum, Document No. J401, 1994).

5. Symbios Logic is the former NCR Microelectronic Products Division of AT&T Global Information Solutions Company.

6. *ObjectBroker Overview and Glossary* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q9KJA-TK, 1994).

7. *Microsoft Windows & MS-DOS Version 6.0,* Chapter 11 (Redmond, Wash.: Microsoft Press, 1993).

8. *SPIRIT Platform Blueprint, SPIRIT STDL Language Specification Platform* (SPIRIT Issue 2.0, Volume 3.0), ISBN 1-85912-063-6 (Reading, U.K.: X/Open Company Ltd., Network Management Forum, Document No. J403, 1994).

9. *SPIRIT Platform Blueprint, SPIRIT STDL Environment, Execution and Protocol Mapping Specification* (SPIRIT Issue 2.0, Volume 3.0), ISBN 1-85912-064-4 (Reading, U.K.: X/Open Company Ltd., Network Management Forum, Document No. J404, 1994).

Biographies

Norman G. Depledge
As manager of the Transaction Processing Design Consulting Group within Digital's layered software organization, Norman Depledge is responsible for managing the technology transfer interface between the TP engineering functions, the field organization, and strategic customers on a worldwide basis. His background in computers spans 33 years. He has held management positions in electrical engineering, software engineering, and marketing. For the past 18 years, he has specialized in on-line transaction processing at Honeywell, Bull, and Digital. Norman has an Honors Degree in electrical engineering from Manchester University, England, and holds three patents in electronic controls.

William A. Turner
William Turner is a consultant in the Transaction Processing Systems Group. He successfully constructed a working model and demonstration of an open, distributable, three-tier client-server architecture with transaction semantics. In previous work, William was a consultant in the Northeast Region and in the New York Production Systems Resource Center. Before joining Digital in 1987, he held positions as a systems manager for Electric Mutual and as a technical support manager for Honeywell Information Systems. William received a B.S. in mathematics from Villanova University in 1966.

Alexandra Woog
As a consultant with Digital's Transaction Processing Systems Group, Alexandra Woog consults with customers on the design and implementation of production systems. In prior work, she was the product manager for Digital's Remote Transaction Router (RTR, in Switzerland), a unit manager for consulting services, and an operating systems specialist.

The AlphaServer 8000 Series: High-end Server Platform Development

David M. Fenwick
Denis J. Foley
William B. Gist
Stephen R. VanDoren
Daniel Wissell

The AlphaServer 8400 and the AlphaServer 8200 are Digital's newest high-end server products. Both servers are based on the 300-MHz Alpha 21164 microprocessor and on the AlphaServer 8000-series platform architecture. The AlphaServer 8000 platform development team set aggressive system data bandwidth and memory read latency targets in order to achieve high-performance goals. The low-latency criterion was factored into design decisions made at each of the seven layers of platform development. The combination of industry-leading microprocessor technology and a system platform focused on low latency has resulted in a 12-processor server implementation—the AlphaServer 8400—capable of supercomputer levels of performance.

The new AlphaServer 8000 platform is the foundation for a series of open, Alpha microprocessor-based, high-end server products, beginning with the AlphaServer 8400 and AlphaServer 8200 systems and continuing through at least three generations of products. When combined with the power of the industry-leading 300-megahertz (MHz) Alpha 21164 microprocessor, this innovative server platform offers the outstanding performance and price/performance required in technical and commercial markets.[1] In uniprocessor performance benchmark tests, the AlphaServer 8400/8200 SPECfp92 rating of 512 is 1.4 times the rating of its nearest competitor, the SGI Power Challenge XL product. In multiprocessor benchmark tests of systems with 1 to 12 processors, the AlphaServer 8400 system posts SPECrate levels greater than 3.5 times those of the HP9000-800 T500 system. In the category of cost for performance, NAS Parallel Class B SP benchmarks show that the AlphaServer 8400 system provides 1.7 times the performance per million dollars of the SGI Power Challenge XL system.[2] Perhaps most impressive is the AlphaServer 8400 performance on the Linpack n×n benchmark.[3] With a Linpack n×n result of 5 billion floating-point operations (GFLOPS), a 12-processor AlphaServer 8400 achieves the performance levels of supercomputers such as the NEC SX-3/22 system and the massively parallel Thinking Machines CM-200 system.

There are two keys to the remarkable performance of the AlphaServer 8400 and AlphaServer 8200 systems: the Alpha 21164 microprocessor chip and the AlphaServer 8000 platform architecture. This paper is concerned with the second of these, the AlphaServer 8000 platform architecture. Specifically, it discusses the principal design issues encountered and resolved in the pursuit of the aggressive performance and product goals for the AlphaServer 8000 series. The paper concludes with an evaluation of the success of this platform development based on the performance results of the first AlphaServer 8000-series products, the AlphaServer 8400 and AlphaServer 8200 systems.

AlphaServer 8400 and AlphaServer 8200 Product Goals

The AlphaServer 8000 platform technical requirements were derived from a set of product goals. This set comprised minimum performance goals and a number of specific configuration and expandability requirements developed from Digital's server marketing profiles. The motivations that shaped the list of goals below were many. Support for legacy I/O subsystems and DEC 7000/10000 AXP compatibility, for example, was motivated by the need to provide Digital's customer installed base with a cost-effective upgrade path from 7000-series hardware to AlphaServer 8000-series hardware. The goals for low-cost I/O subsystem, peripheral component interconnect (PCI), and EISA support and for rackmount cabinet support were included to take advantage of emerging industry standards and open systems and their markets. The processor, I/O, and memory capacity goals were driven simply by the competitive state of the server market.

- Provide industry-leading enterprise and open-office server performance.
- Provide twice the overall performance of the DEC 7000/10000 AXP server products.
- Support up to 12 Alpha 21164 processors.
- Support at least 14 gigabytes (GB) of main memory.
- Support multiple I/O port controllers—up to 144 I/O slots.
- Provide a low-cost, integrated I/O subsystem.
- Support new, industry-standard PCI and EISA I/O subsystems.
- Continue to support legacy I/O subsystems, such as XMI and Futurebus+.
- Make centerplane hardware compatible with an industry-standard rackmount cabinet.
- Make centerplane hardware mechanically compatible with the DEC 7000/10000 AXP-series cabinet.

Performance Goals and Memory Read Latency Issues

Although "providing industry-leading performance" and "doubling the performance" of an existing industry-leading server present excellent goals for the development of a new server, it is difficult to design to such nebulous goals. To quantify the actual technical requirements for the new AlphaServer 8000 platform, the design team utilized a performance study of the DEC 7000/10000 AXP systems and conducted a detailed analysis of symmetric multiprocessing (SMP) system operation. As a result of the analyses, aggressive system data bandwidth and memory read latency goals were established, as well as a design philosophy that emphasized low memory read latency in all aspects of the platform development. This section addresses the read latency issues and goals considered by the design team. The 8000 platform development is the focus of the section following.

Read latency is the time it takes a microprocessor to read a piece of data into a register in response to a load instruction. If the data to be read is found in a processor's cache, the read latency will typically be very small. If, however, the data to be read resides in a computer system's main memory, the read latency is typically much larger. In either case, a processor may have to wait the duration of the read latency to make further progress. The smaller the read latency, the less time a processor waits for data and thus the better the processor performs.

Cache memories are typically used to minimize read latency. When caches do not work well, either because data sets are larger than the cache size or as the result of non-locality of reference, a computer system's processor–memory interconnect contributes significantly to the average read latency seen by a processor. The system characteristics that help determine a processor's average read latency are the system's minimum memory read latency and data bandwidth.

A system's minimum memory read latency is the time required for a processor to fetch data from a system's main memory, unencumbered by system traffic from other processors and I/O ports. As processors and I/O ports are added to a system, their competition for memory and interconnect resources tends to degrade the system's memory read latency from the minimum memory read latency baseline. A system's data bandwidth, i.e., the amount of data that a system can transfer between main memory and its processors and I/O ports in a given period of time, will determine the extent to which processors and I/O ports will degrade each other's read latency. As data bandwidth increases, so too does a system's ability to support concurrent data references from various processors and I/O ports. This increased bandwidth and concurrent data referencing serve to reduce competition for resources and, as a result, to reduce memory read latency. Thus we can conclude that the more available data bandwidth in a system, the closer the memory read latency will be to the minimum. This conclusion is supported by the results of a queuing model used to support the AlphaServer 8000 platform development. This queuing model, originally implemented to evaluate bus arbitration schemes, outputs the average read latencies experienced by each processor in a system as the number of processors and the number of memory resources are varied. It is important to note that in this model memory resources, or banks, determine the amount of system bandwidth: the more memory banks, the more bandwidth. It is also important to note that the minimum read latency in this model is 168 nanoseconds (ns). The results of the model are shown in Table 1. These results clearly show that latency degrades as the number of system processors is increased and that latency improves as the system's bandwidth—number of memory banks—is increased.

Many technical market benchmarks, such as the Linpack benchmarks and the McCalpin Streams benchmark, stress a computer system's data bandwidth capability. The regularity of data reference patterns in these benchmarks allows a high degree of data prefetching. Consequently, data can be streamed into a processor from main memory so that a piece of data has an unnaturally high probability of being resident in the processor's cache when it is needed for some calculation. Ironically, this amounts to using smart software to minimize read latency. By reading a piece of data into a processor's cache before it is actually needed, the software presents the processor with a small cache read latency instead of a long memory latency when the data is needed. Thus the streaming techniques applied in these benchmarks allow processors in high-bandwidth systems to stall for a full memory read latency period only when starting up a stream of data. Therefore memory latency can be amortized over lengthy high-bandwidth data streams, minimizing its significance. It is important to note, however, that although bandwidth is the system attribute that dominates performance in these workloads, it dominates performance through its effect on read latency.

Commercial workloads like the Transaction Processing Performance Council's benchmark suite, on the other hand, typically have more complex data patterns that frequently defy attempts to prefetch data. When some of these codes parse data structures, in fact, the address of each data access may depend on the results of the last data access. In any case where a processor must wait for memory read data to make progress, a system's memory read latency will determine the period of time that the processor will be stalled. Such stall periods directly affect the performance of computer systems on commercial workloads. These assertions supported by a study on the performance of commercial workloads on Digital's Alpha 20164-based DEC 7000/1000 AXP server.[4] It is important to note here that the latency ills flagged in this study cannot be cured with raw system data bandwidth or software-enhanced latency reduction. Low memory latency alone can address the needs of these workloads.

Comparable industry systems from IBM and Hewlett-Packard (HP) do not stress low memory latency system development in their respective RISC System/6000 SMP or Hawks (PA-8000–based) SMP systems.[5,6] In fact, neither directly acknowledges memory latency as a significant system attribute. This mind set is reflected in the results: Based on IBM's documentation, we estimate the RISC System/6000 SMP's minimum main memory read latency to be in the neighborhood of 600 to 800 ns.

IBM and HP do emphasize system bandwidth in their designs. HP provides a 960-megabyte-per-second (MB/s) "runway" processor–memory bus in its Hawks system. The actual data bandwidth of this bus is slightly less than the quoted 960 MB/s, since the bus is shared between address and data traffic. IBM, on the other hand, goes to the extent of applying a data crossbar switch in conjunction with a serial address bus to reach an 800-MB/s rate in its RISC System/6000 SMP system. The actual attainable data bandwidth in IBM's system is determined by the bandwidth of its address bus.

In the past, Digital's systems have shown much the same balance of bandwidth and latency as have their competitors. The DEC 7000/10000 AXP system has a minimum main memory read latency of 560 ns and a system data bandwidth of 640 MB/s. The AlphaServer 8000 platform, however, was developed with a marked emphasis on low memory read latency. This emphasis can be seen through nearly all phases of system development, including the system topology, clocking strategy, and protocol. This latency-oriented mindset is reflected in the results: The AlphaServer 8000 platform boasts minimum memory read latencies of 200 ns. The AlphaServer

Table 1
Average Read Latency as a Function of the Number of Processors and Bandwidth (Number of Memory Banks)

	Average Read Latency (Nanoseconds)			
Number of Processors	2 Memory Banks	4 Memory Banks	6 Memory Banks	8 Memory Banks
1	185	179	177	176
2	224	200	193	190
4	358	253	230	220
8	928	439	338	299

8400 and 8200 systems feature a minimum memory read latency of 260 ns. To back up these latencies, the AlphaServer 8000 platform supports a tremendous 2,100 MB/s of data bandwidth. The AlphaServer 8400 and 8200 systems, although not capable of providing the full 2,100 MB/s, still provide 1,600 MB/s of bandwidth. This gives the systems less than half the memory latency of comparable industry systems while providing nearly twice the bandwidth. Furthermore, these attributes improve upon the DEC 7000/10000 AXP attributes by factors of 2 to 3. Although difficult to determine exactly how these attributes would translate into overall system performance, they were accepted as sufficient to meet the AlphaServer 8000 platform performance goals. A comparison of the maximum DEC 7000/10000 AXP SPECrates of approximately 25,000 integer and 40,000 floating point with the maximum AlphaServer 8400 SPECrates of 91,580 integer and 14,0571 floating point indicates that these attributes were sound choices.

AlphaServer 8000 Platform Development

Referring to the AlphaServer 8000 platform as a "foundation" for a series of server products does not give a clear picture of what constitutes a server platform. The AlphaServer 8000 platform has both physical and architectural components. The physical component consists of the basic physical structure from which 8000-series server products are built. This includes power systems, thermal management systems, system enclosures, and a centerplane card cage that implements the interconnect between processor, memory and I/O port modules. The processor, memory, and I/O modules are printed circuit board (PCB) assemblies that can be implemented with varying combinations of processor, dynamic random-access memory (DRAM), and application-specific integrated circuit (ASIC) components. The assemblies are inserted into the platform centerplane card cage in varying configurations and in varying enclosures to create the various 8000-series products. The AlphaServer 8200 system, for example, comprises up to six Alpha 21164-based TLEP processor modules, TMEM memory modules, or ITIOP and TIOP I/O port modules in an industry-standard rack-mount system. The AlphaServer 8400 system comprises up to nine TLEP processor modules, TMEM memory modules, or ITIOP and TIOP I/O port modules in a DEC 7000 AXP-style data center cabinet.

The architectural component of the AlphaServer 8000 platform consists primarily of a collection of technological, topological, and protocol standards. This collection includes the processor–memory interconnect strategy, the bus interface technology, the clock technology and methodology, and the signaling protocols. For example, the TLEP, TMEM, and TIOP modules all implement bus interfaces in the same integrated circuit (IC) packages with the same silicon technology and drive their common interconnect bus with the same standard bus driver cell. Furthermore, all these modules apply nearly identical clocking circuits and communicate by means of a common bus protocol. The ephemeral architectural standards that constitute the "platform" specify exact physical requirements for designing the AlphaServer processor–memory–I/O port interconnect and the various modules that will populate it. It is important to note that the key to AlphaServer 8000 performance is found in these standards. As we explore the design decisions and tradeoffs that shaped the AlphaServer 8000 platform, it is this collection of architectural standards that we actually probe.

Throughout this analysis of the AlphaServer 8000 architecture, two themes frequently recur: low memory latency and practical engineering. As discussed in the context of the AlphaServer 8000 goals, low memory read latency was identified as the key to system performance. As such, low latency was factored into nearly every system design decision. Design decisions in general can be thought of as being resolved in one of two ways: by emphasizing Digital's superior silicon technology or by effecting architectural finesse. Use of superior technology is self-explanatory; it involves pushing leading-edge technology to simply overwhelm and eliminate a design issue. Architectural finesse, on the other hand, typically involves a shift in operating mode or configuration that allows a problem to be avoided altogether. Practical engineering is the art of finding a balance between leading-edge technology and architectural finesse that produces the best product.

Layered Platform Development

Platform development typically involves a simple three-layer process: (1) determine a basic system topology, (2) establish the electrical means by which various computer components will transmit signals across the system topology, and (3) apply a signaling protocol to the electrical transmissions to give them meaning and to allow the computer components to communicate. System topology determines how processor, memory, and I/O components of a computer system are interconnected. Computer interconnects may involve simple buses, multiplexed buses, switches, and multitiered buses. The electrical means for transmitting signals across a computer interconnect may involve bus driver technology, switch technology, and clock technology. Signaling protocols apply names to system interconnect signals and define cycles in which the signals have valid values. This naming and definition allows each computer component to understand the transmissions of other components.

As the AlphaServer 8000 platform development progressed, this simple three-layer platform development model was found to be insufficient. Efforts to achieve the low-latency performance goal and the simple product goals uncovered unexpected design issues. The resolution of these design issues led to the creation of a more robust seven-layer platform development model. When certain multi-driver bus signals threatened the cycle time of the AlphaServer 8000 system bus, for example, the system's latency goals were threatened as well. The practical solution to this multi-driver signal problem was the creation of specific signaling conventions for problematic classes of signals. This innovation led to the birth of the Signaling Layer of the development model. Similarly, when the integration of PCI I/O into the system was found to conflict with primary protocol elements that were key to low latency processor–memory communication, the concept of a "superset protocol" was created. This led to the creation of the Superset Protocol Layer of the development model. The seven-layer platform development model is contrasted with the simple three-layer development model in Figure 1.

The analysis of the AlphaServer 8000 platform design presented here traces the key system design decisions through each of the seven layers of the development process. Each layer will be described in greater detail as this analysis proceeds.

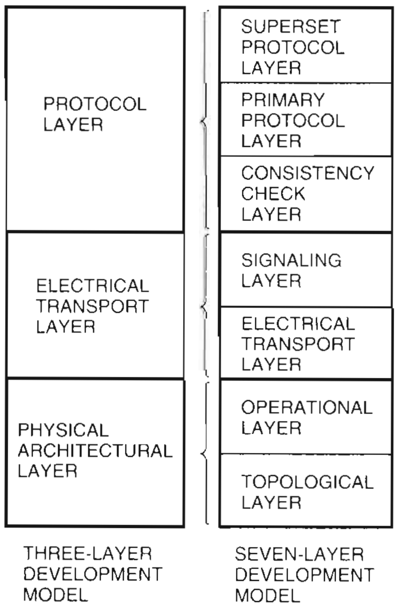

PROTOCOL LAYER
ELECTRICAL TRANSPORT LAYER
PHYSICAL ARCHITECTURAL LAYER
THREE-LAYER DEVELOPMENT MODEL
SUPERSET PROTOCOL LAYER
PRIMARY PROTOCOL LAYER
CONSISTENCY CHECK LAYER
SIGNALING LAYER
ELECTRICAL TRANSPORT LAYER
OPERATIONAL LAYER
TOPOLOGICAL LAYER
SEVEN-LAYER DEVELOPMENT MODEL

Figure 1
Comparison of Conventional Three-layer Model with Seven-layer Platform Development Model

Topological Layer

Server-class computers typically comprise processor, memory, and I/O port components. These components are usually found in the form of PCB modules. A computer system's topology defines how these computer components are interconnected. Computer topologies are many and varied. The IBM RISC System/6000 SMP, for example, links its modules by means of an address bus and a data switch. Its memory modules are grouped into a single memory subsystem with one connection to the address bus and one connection to the data switch. The HP Hawks SMP system, by comparison, links its modules by means of a single bus onto which address and data are multiplexed. The Hawks system also groups its memory into a single memory subsystem with one connection to the multiplexed bus.[7] Digital's DEC 7000/ 10000 AXP also uses a single multiplexed address and data bus. Unlike the IBM and HP systems, the DEC 7000/10000 AXP system allows its memory to be distributed, with multiple connections to its multiplexed bus.

None of the IBM, HP, or prior Digital systems meet the latency goals of the AlphaServer 8000 platform. Exactly how much system topology contributes to these systems' latencies is unclear. A multiplexed address and data bus certainly creates a system bottleneck and can contribute to latency. Likewise, unified memory subsystems can often have associated overhead that can translate into latency. In addition to performance issues, topologies such as the IBM switch-based system have significant cost issues. If, for example, a customer were to purchase a sparsely configured —two processors perhaps—IBM system, such a customer would be required to pay for the switch support for up to eight processors. This creates a high system entry cost and a potentially lower incremental cost as functionality is added to the system. In a simple bused system, a customer pays only for what is needed to support the specific functionality required. This creates a more manageable entry cost and a smooth, if slightly steeper, incremental cost. From Digital's marketing perspective, this makes a bused system preferable, provided it can satisfy bandwidth and latency requirements.

Uniprocessor computer topologies, an example of which is shown in Figure 2, typically exhibit the lowest memory read latencies of any computer class. As such, this simple uniprocessor topology was chosen as the basis from which to develop the AlphaServer 8000 platform topology. In the uniprocessor model, processor chips communicate with DRAM arrays through separate address and data paths. These paths include address and data interfaces and buses. The AlphaServer 8000 topology was created by adding a second set of interfaces between the address and data buses and the DRAM array, and connecting additional

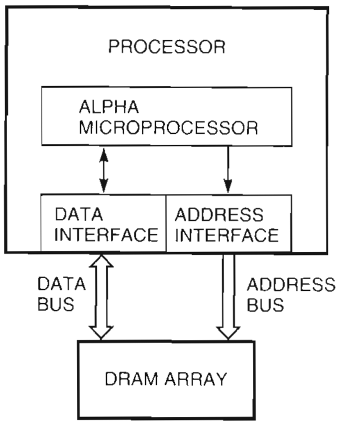


Figure 2
Simple Uniprocessor System Topology

microprocessors, memory arrays, and I/O ports to the buses by means of similar interfaces. The resultant topology is shown in Figure 3. This topology features separate address and data buses. These buses together are referred to as the AlphaServer 8000 system bus.

The topology presented in Figure 3 is an abstract. To flesh out this abstract and measure it against specific system goals, signal counts, cycle times, and bus connection (slot) counts must be added. It is in this effort that practical engineering must be applied. To achieve the system's bandwidth goal, for example, the data bus could be implemented as a wide bus with a high clock frequency, or it could be replaced with a switch-based data interconnect, like that of the IBM RISC System/6000 SMP. The high-frequency bus presents a significant technological challenge in terms of drivers and clocking. This challenge grows as the number of bus slots grows. The growth of the technological challenge is a significant issue given the system's configuration goals. The switch interconnect, on the other hand, avoids the technological challenges by providing more data paths at lower clock frequencies. The lower clock frequencies, however, can translate directly into additional latency. Given the emphasis placed on memory latency and the advantages associated with simple bused systems, the practical design choice was to adopt a wide, high-frequency data interconnect. The resultant AlphaServer 8000 system bus features 9 slots, an address bus that supports a 40-bit address space, and a 256-bit (plus error-correcting code [ECC]) data bus. To meet configuration goals, processor modules necessarily support at least two microprocessors per module, memory modules support up to 2 GB of DRAM storage, and I/O port modules support up to 48 PCI slots. To meet performance goals, both buses must operate at a frequency of 100 MHz (10-ns cycle).

The AlphaServer 8000 platform topology has a number of advantages. The most significant advantage is that memory read latency from any processor to any memory array is comparable to the latency of a uniprocessor system. The delay associated with two interfaces—one address interface and one data interface—is all that is added into the path. In addition, the platform's simple bus topology features a low entry cost, a simple growth path (just insert another module) and flexible configuration (just about any module can be placed in any slot).

Operational Layer

The Operational Layer is so named for lack of a better descriptor. The layer is actually a place to define a high-level system clocking strategy. This strategy has two key components: definition of target operating frequencies and definition of a design methodology to support operation across all the defined operating frequencies. The design methodology component of this strategy may seem better suited for a higher order development layer, such as the Protocol Layer. However, because the methodology is logically associated with the system's operating frequency range and the operating frequency range provides a foundation for the Electrical Transport Layer, it seemed appropriate to include both components of the strategy in the Operational Layer.

In personal computer (PC)-class microprocessor systems, clock rates are typically slow (33 MHz to 66 MHz). Complementary components capable of operating at these speeds are readily available, e.g., transceivers, static random-access memory (SRAM), ASIC, DRAM, and programmable array logic (PAL). Therefore entire PC systems are typically run synchronously, i.e., the system logic (typically a motherboard) and the microprocessor run at identical clock speeds. Alpha processors, on the other hand, run at clock rates exceeding 250 MHz. The current state of complementary components makes running system logic at Alpha processor rates impractical if not impossible. Many of these components cannot perform internal functions at a 250-MHz rate, let alone transfers between components.

Digital's DEC 7000/10000 AXP systems solved the problem of Alpha microprocessor and system clock disparity by running both the Alpha microprocessor and the DEC 7000/10000 AXP system hardware at their respective maximum clock rates and synchronizing address and data transfers between the microprocessor and the system. Each time a transfer was synchronized, however, a synchronization latency penalty was added to the latency of the transfer. In the DEC 7000/10000 AXP system, two synchronization penalties—one for an address transfer to the system and one for a data transfer to the processor—are added to each memory read latency. With multiple data transfers, the data transfer from the system to the processor can be particularly large. When combined,

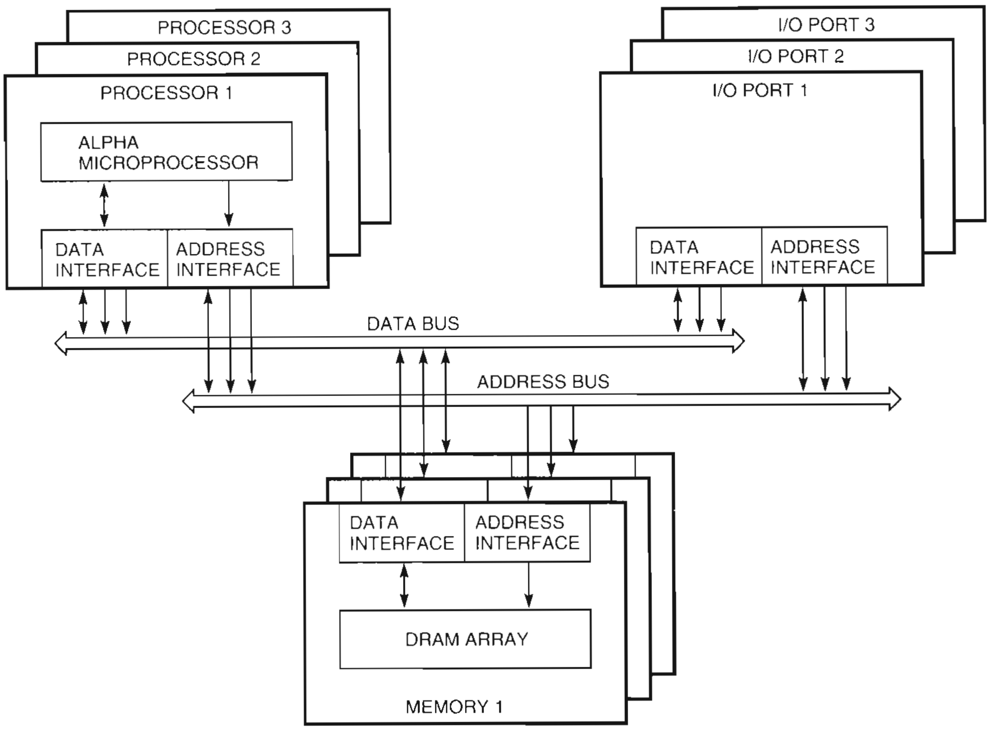


Figure 3
AlphaServer 8000 Multiprocessor System Topology

the two penalties added nearly 125 ns to the DEC 7000/10000 AXP read latency, or approximately 25 percent of the total 560-ns latency. The same 125 ns, however, could add another 60 percent to the AlphaServer 8000 platform's lower target latency of 200 ns.

Given its latency goals, the AlphaServer 8000 platform implements a clocking methodology that minimizes synchronization penalties and thus minimizes read latency. This methodology involves clocking the entire AlphaServer system—up to the I/O channels—synchronous to the microprocessor in such a way that the Alpha microprocessor operates at a clock frequency that is a direct multiple of the system clock frequency. With a 100-MHz (10-ns cycle) clock rate, for example, the AlphaServer 8000 could support a 200-MHz (5-ns cycle) Alpha processor using a 2× clock multiplier. Since the processor must still synchronize with a system clock edge when transferring address and data to the system, synchronization penalties are not eliminated altogether. They can, however, be limited to less than 10 ns, or 5 percent of the AlphaServer 8000 platform's total read latency.

Synchronous clocking by means of clock multiples is not unique and innovative in and of itself. The uniqueness of the AlphaServer 8000 clocking strategy lies in its flexibility. Since the AlphaServer 8000 platform must support at least three generations of Alpha processors to satisfy its product goals and the specific operating frequencies of those processors is difficult to predict, the AlphaServer 8000 platform must be capable of operating across a range of clock frequencies. Specifically the AlphaServer 8000 platform is capable of operating at clock frequencies between 62.5 MHz (16-ns cycle) and 100 MHz (10-ns cycle).

Operating across a range of frequencies may seem a trivial requirement to meet; if logic were designed to operate at a 10-ns cycle time, it should certainly continue to function electrically at a 16-ns cycle time. The real issues that this frequency range creates, however, are much more subtle. DRAMs, for example, require a periodic refresh. The refresh period for typical DRAM may be 50 milliseconds (ms). If a system were designed to a 10-ns clock rate, the system would be designed to initiate a DRAM refresh every 5,000,000 cycles. If the system were to be slowed to a 16-ns clock rate, the system would initiate a DRAM refresh every 80 ms based on the same 5,000,000 cycles. This could cause DRAMs to lose state and corrupt system operation. Similarly, DRAMs have a fixed read access time. The AlphaServer 8400/8200 TMEM module, for example, uses 60-ns DRAMs. If the DRAM's controller is designed as a 7-cycle controller and clocked at a 10-ns clock rate, it would access the 60-ns DRAM in 70 ns. If the system were slowed to a 16-ns clock rate, the system would, using the same controller, consume 112 ns in accessing the same 60-ns DRAM. This application of a single simple controller over a frequency range directly increases the DRAM's read latency and decreases the DRAM's bandwidth. This non-optimal DRAM performance in turn directly increases the system read latency and decreases the system bandwidth.

The AlphaServer 8000 platform design addresses these issues by implementing controllers that can be reconfigured based on the system's specific operating frequency. The TMEM module, for example, implements a reconfigurable controller for sequencing the reads and writes of its DRAMs. This controller has three settings: one for cycle times between 10 ns and 11.2 ns, one for cycle times between 11.3 ns and 12.9 ns, and one for cycle times between 13 ns and 16 ns. Each setting accesses the DRAMs in differing numbers of system clock cycles, but all three modes access the DRAMs in approximately the same number of nanoseconds. By allowing flexible reconfiguration, this controller allows the TMEM to keep the DRAM's read latency and bandwidth as close to ideal as possible. Other examples of reconfigurable controllers are the TMEM's refresh timer and the TLEP's cache controller.

It should be noted here that the AlphaServer 8000 operating frequency range and processor-based frequency selection account for the disparities between the AlphaServer 8000 platform's bandwidth capability and the AlphaServer 8400 and 8200 products' bandwidth capabilities. The Alpha 21164 processor is the basis for the 8400 and 8200 products. This 300-MHz (3.33-ns cycle) microprocessor, combined with a 4× clock frequency multiplier, sets the system clock frequency at 75 MHz (13.3-ns cycle). This 13.3-ns cycle time, when applied to the 256-bit data bus, produces the 1,600 MB/s of data bandwidth. The cycle time increases the read latency of the 8400 and the 8200 to some extent as well, but the reconfigurable DRAM controllers help to mitigate this effect.

Electrical Transport Layer

When the bused system topology was selected in the Topological Layer of the AlphaServer 8000 platform development, a practical engineering decision was made to emphasize leading-edge technology as the means to accomplish our performance goals, as opposed to elegant architectural chicanery. It was observed in the topological discussion that, with the selected system topology, bus cycle time was critical to meeting the platform's performance goals. The Electrical Transport Layer of the platform development involved selecting or developing the centerplane, connector, clocking, and silicon interface technology that would allow the AlphaServer 8000 system bus to operate at a 100-MHz clock frequency. The most innovative of the technological developments that resulted from this effort were the platform's clocking system and its custom bus driver/receiver cell.

To put the AlphaServer 8000 100-MHz system bus goal in perspective, consider the operating frequencies of a number of today's highly competitive microprocessors.[8] The NexGen Nx586 operates at 93 MHz. The Intel Pentium, Cyrix M1, and AMD K5 all operate at 100 MHz. The Intel P6 operates at 133 MHz. In all these microprocessors, the 100+/− MHz operation takes place on a silicon die less than 1 inch square. To meet its goals, the AlphaServer 8000 system bus must transfer data from an interface on a module in any slot on the system bus to an interface on another module in any other slot on the system bus across a 13-inch-long wire etch, with nine etch stubs and nine connectors, in the same 10 ns in which these microprocessors transfer data across 1-inch dies. By any measure this is a daunting task.

A breakdown of the elements that determine minimum cycle time aptly demonstrates the significance of clock system design, bus driver design, and bus receiver design in the AlphaServer 8000 system bus development. Minimum bus cycle time is the minimum time required between clock edges during which data is driven from a bus driver cell on one clock edge and is received into a bus receiver cell on the next clock edge. An equation for determining the minimum cycle time is shown below. T_{cmin} is the minimum cycle time. T_{prop} is the time, measured from a rising clock edge, that is required for a bus driver to drive a new bus signal level to all system bus receivers. T_{setup} is the time a bus receiver needs to process a new bus signal level before the signal can be clocked into the receiver cell. T_{skew} is the variation between the clock used to clock the bus driver and the clock used to clock the bus receiver. T_{prop}, T_{setup}, and T_{skew} must all be minimized to achieve the lowest possible cycle time. The value of T_{skew} is determined by the system clock design. The values of T_{prop} and T_{setup} are determined by the bus driver/receiver cell design.

$$T_{cmin} = T_{prop} + T_{setup} + T_{skew}$$

AlphaServer 8000 System Bus Interface To provide some context for the clock and bus driver/receiver discussions, it is necessary to briefly describe the standard AlphaServer 8000 system bus interface. Each AlphaServer 8000 module implements a standard system bus interface. This interface consists of five ASICs: one interfaces to the AlphaServer 8000 address bus, and four interface to the AlphaServer 8000 data bus.[9] Each ASIC is implemented in Digital's 0.75-micrometer, 3.3-volt (V) complementary metal-oxide semiconductor (CMOS) technology and features up to 100,000 gates. Each ASIC is packaged in a 447-pin interstitial pin grid array (IPGA) and features up to 273 user I/Os.

Essential to the AlphaServer 8000 development were the speed of the CMOS interface ASIC technology and the development team's ability to influence the ASIC design process. "Influencing the design process" translated to the ability to develop a standard cell design library and process that is for and in concert with the development of the AlphaServer 8000 platform. The standard cell library, together with the CMOS silicon technology, provided the AlphaServer 8000 platform's required speed; complex logic functions (5 to 8 levels of complex logic gates) can be performed within a 10-ns cycle. "Influencing the design process" also translated to the ability to design a fully custom bus driver/receiver cell. Thus the development team could create a custom driver/receiver cell tailored to the specific needs of the AlphaServer 8000 system bus.

Clock Technology The primary goal of the AlphaServer 8000 platform clock distribution system was to maintain a skew (T_{skew}) as small as possible between any two clocks in the system, while delivering clocks to all clocked system components. The goal of minimum skew is consistent with attaining the lowest possible bus cycle time, the highest possible system data bandwidth, and the lowest possible memory read latency. It is important to note that in the AlphaServer 8000 platform, skew between clocks is not simply measured at the clock pins of the various clocked components. Skew is measured and, more important, managed at the actual "point of use" of the clock, for example, at the clock pins of ASIC flip-flops. This is an important point when dealing with ASICs. Since different copies of even the same ASIC design can have different clock insertion delays, additional skew can be injected between clocks after the clocks pass their ASIC pins.

The AlphaServer 8000 clock distribution system is implemented according to a two-tier scheme. The first tier, the *system clock distribution,* distributes a clean radio frequency (RF) sine wave clock to each system bus module. The second tier, the *module clock distribution,* converts the system RF sine wave clock to a digital clock and distributes the digital clock to each module's components. The module clock distribution tier also manages the skew between the system RF sine wave clock and all copies of each module's digital clock by means of an innovative "remote delay compensation" mechanism. The system clock distribution delivers clocks to the nine system bus module slots with a maximum of 40 picoseconds (ps) of skew. The module clock distribution delivers clocks to the various module components, most notably system bus interface ASICs, with a maximum of 980 ps of skew. The skew between any ASIC flip-flop on any AlphaServer 8000 module and any ASIC flip-flop on any other AlphaServer 8000 module is guaranteed to be less than 1,100 ps.

The AlphaServer 8000 system clock distribution begins on the system clock module with a single-ended RF oscillator, a constant impedance bandpass filter, and a nine-way power splitter. The power splitter, by way of the bandpass filter, produces nine spectrally clean, amplitude-reduced copies of the oscillator sine wave. These nine outputs are tightly matched in phase and amplitude. They are distributed to the nine system bus module connectors by means of matched-length, shrouded, controlled-impedance etch. This design provides the modules with low skew (30 to 40 ps), high-quality (greater than 20-decibel signal-to-noise ratio) clocks.

The RF sine wave clock was an ideal selection for system clock distribution. By eliminating all high-order harmonics, the edge rates and propagation times of the clock wave are fixed and predictable across the distribution network. This predictability eliminates variation in the clock as perceived by the clock receiver on each module, thus minimizing skew. It also greatly reduces constraints on the design of connectors, etch, termination, etc.

The AlphaServer 8000 module clock distribution is a boilerplate design that is replicated on each AlphaServer 8000 module. On each module, the system sine wave clock is terminated by a single-ended-to-dual-differential output transformer. This transformer produces two phase- and amplitude-matched differential clocks that are fed into one or two AlphaServer 8000 clock repeater chips (DC285 chips). These chips convert the sine wave clocks into CMOS-compatible digital clocks; distribute multiple copies of the digital clocks to various module components, including the system bus interface ASICs; and perform remote delay clock regulation on each clock copy.

The remote delay clock regulation is performed by a custom, digital delay-locked loop (DLL) circuit. This DLL circuit was devised specifically to deskew clocks all the way to their point of use in the system bus interface ASICs. The principles of DLL-based remote delay clock regulation are simple. The sum of the delays associated with (1) the clock repeater chips, (2) the module clock distribution etch, and (3) the ASIC clock distribution network constitutes the insertion delay of the ASIC point-of-use clock with respect to the system sine wave clock. With no clock regulation, this delay appears as skew between the system clock and the point-of-use ASIC clock. Between ASICs on different modules, a fixed portion of the clock insertion delay will correlate and need not be factored into the overall system skew. Since the insertion delay can easily approach 7 ns, however, the variation in the insertion delays to different ASICs, which must be factored into the overall system skew, can also be significant. To reduce the skew between the system sine wave clock and the point-of-use ASIC clock, the clock repeater uses a digital delay line to add delay to the clock repeater output clock. Enough delay is added so that the insertion delay plus the delay-line delay is equal to an integer multiple of the system clock. This delay moves the point-of-use clock ahead to a point where it again lines up with the system clock. As the system operates, the system and point-of-use clocks may drift apart. In response, the clock repeater adjusts its delay line to pull the clocks back together. This process of delaying clocks and dynamically adjusting the delay is called remote delay clock regulation. When the clock separation, or drift, is measured by a clock "replica loop" and the clock delay is inserted by means of a digital delay line, the process is called DLL-based remote delay clock regulation.[10] Using the clock repeater chips in this way, AlphaServer 8000 modules are able to achieve point-of-use to point-of-use skew of approximately 930 to 980 ps. Combined with the system module-to-module skew of 30 to 40 ps, this provides the quoted system-wide clock skew of no more than 1,100 ps.

It is worth noting that although the AlphaServer clock repeater was primarily developed for use with system bus interface ASICs, it is a generally versatile part. It may, for instance, be used with non-ASIC parts such as transceivers and synchronous SRAMs. In these cases, the clock pin of the non-ASIC part is treated as the point of use of the clock. The clock repeater may also be used for precise positioning of clock edges. On the TLEP module, for example, the Alpha 21164 microprocessor's system clock is synchronized to a clock repeater output by means of a digital phase-locked loop (PLL) on the microprocessor. The Alpha 21164's PLL operates in such a way that the 21164's clock is always in phase with or always trailing the system (reference) clock. It can trail by as much as 2 ns. Such a large clock disparity in this fixed orientation can create setup time problems for transfers from the Alpha 21164 to the system and hold-time problems for transfers from the system to the Alpha 21164. The TLEP design addressed this problem by lengthening the replica loop associated with the Alpha 21164 clock and thereby shifting the microprocessor clock 1 ns earlier than the balance of the clock repeater output clocks. Since the Alpha 21164 clock was either in phase or 2 ns later than its associated clock repeater clock, which is 1 ns earlier than the rest of the clock repeater clocks, the 21164 clock now appears to be either 1 ns earlier or 1 ns later than the rest of the clock repeater system clocks. This centering of the module clocks with respect to the 21164 clock halves the required setup or hold margin.[11, 12, 13, 14]

Bus Driver Technology Like the AlphaServer 8000 clock system, the AlphaServer 8000 system bus driver/receiver cell was specifically designed to minimize bus cycle time. As with the clock logic, the goal of minimizing cycle time was a result of the effort to minimize system read latency and maximize system data bandwidth. In the effort to minimize the bus cycle time, the design of the AlphaServer 8000 bus driver/receiver cell was focused on minimizing the propagation delay (T_{prop}) of the system bus driver circuit and minimizing the setup time (T_{setup}) of the system bus receiver.

The AlphaServer 8000 system bus driver/receiver cell is a fully custom CMOS I/O cell, which incorporates a bus driver, a bus receiver, and an output flip-flop and an input flip-flop in a single cell. Consisting of nearly 200 metal oxide semiconductor field-effect transistors (MOSFETs), the bus driver cell is powered by standard 3.3-V CMOS power, but drives the bus at a much lower 1.5-V level (i.e., voltage swings between 0 and 1.5 V). This low voltage output serves to reduce the bus driver's power consumption and permits compatibility with future CMOS technologies that are powered by voltages less than 3.3 V. Many of the bus driver cell's critical characteristics are "programmable," such as the 1.5-V output, the receiver switching point, the driver's drive current limit, and the driver's rise and fall times. These values are programmed and, most important, are held constant by means of reference voltages and resistances external to the bus driver/receiver cell's ASIC package. They allow the cell to produce uniform, predictable, high-performance waveforms and to transmit and receive data in a clock cycle of 10 ns.

The bus driver/receiver's high performance begins with its output flip-flop and driver logic. The output flip-flop is designed for minimum delay and is integrally linked to the output driver. This configuration produces clock-to-output times of 0.5 ns to 1 ns. The output driver itself, with its programmable output voltage and edge rates, allows the shape of the output waveform to be carefully controlled. The cell's programmable values are set such that the AlphaServer system bus waveform balances the edge rate effects of increased crosstalk with increased propagation delay. Furthermore, the bus waveform is shaped in such a way that it allows incident wave transmission of signals. As such, a signal can be received on its initial propagation across the bus centerplane, as opposed to waiting for signal reflections to settle. All the driver characteristics serve to reduce bus settling time. When combined with the low clock-to-output time of the output flip-flop, this reduced settling time produces a very low driver circuit propagation delay (T_{prop}).

The bus driver/receiver cell's receiver and input flip-flop further contribute to its high performance. Designed with a programmable reference voltage, the receiver has a very precise switching point. Whereas typical receivers may have a 200-millivolt (mV) to 300-mV switching window, the bus driver/receiver cell's receiver has a switching window as small as 40 mV. This diminished switching uncertainty directly reduces the receiver's maximum setup time. The input flip-flop's master latch is a sense-amplifier-based latch as opposed to a simple inverter-based latch. The sense amplifier, with its ability to resolve small voltage differentials much faster than standard inverters, allows the master latch to determine its next state much more rapidly than a standard latch. This characteristic serves to reduce both the receiver's setup and hold time requirements.

In general, the setup and hold time requirements of a state element are interrelated. The setup time, for example, can be reduced at the expense of hold time. Since setup time contributes to cycle time and hold time may not, reducing setup time is desirable. The AlphaServer 8000 bus driver/receiver cell requires at most 300 ps of combined setup and hold time. However, since the edge rates of the cell driver are so well controlled, the minimum propagation time for a bus signal is always guaranteed to exceed 300 ps. As a result, the bus receiver circuit is designed with all 300 ps charged as hold time. This renders a minimized receiver setup time (T_{setup}) of 0 ps.

The AlphaServer 8000 bus driver/receiver cells have a number of additional features that further reduce the propagation delay (T_{prop}) of the driver circuit. The cell, for example, features in-cell bus termination, which provides the system bus with full, distributed termination. Simulations have shown that such distributed termination can provide an advantage of 500 ps over common end termination. The bus driver/receiver cell's termination resistance, like other cell parameters, is programmable and made identical throughout all system ASICs by means of a reference resistor external to each ASIC.

The bus driver/receiver cell also features a special preconditioning function that improves the driver's propagation delay by as much as 1,500 ps. This feature causes all bus drivers to begin driving toward the opposite state each time they receive a new value from the bus. If the bus is changing state from one cycle to the next, the feature causes all drivers to begin driving the bus to a new state in the next cycle. In doing so, all bus driver cell drivers contribute current and accelerate the bus transition. If the bus is not changing from one cycle to the next, the drivers simply push the state of the bus toward the opposite state, but only to a benign voltage well short of the switching threshold.

All of the bus driver cell's programmable features, such as switching point, output voltage, edge rates, and termination resistance, make the bus driver cell a very stable and high-performance interface cell. The existence of these features, however, is an element of the bus driver cell's complementary process-voltage-temperature (PVT) compensation function. PVT compensation is meant to make a device's operating characteristics independent of variations in the semiconductor process, power supply voltage, and operating temperature. By applying PVT compensation in every AlphaServer system bus interface ASIC, bus driver cells in different ASICs, for example, can drive nearly identical system bus waveforms even if those ASICs come from manufacturing lots with varying speed characteristics. AlphaServer 8000 PVT compensation is based on reference voltages and resistances provided by very precise, low-cost, module-level components. The PVT compensation circuit measures these references and configures internal voltages and resistances so that all bus driver cells can operate uniformly and predictably. By creating predictability and thus reducing uncertainty and skew, bus cycle time is minimized.

Signaling Layer

Powerful though it may be, the AlphaServer 8000 bus driver/receiver cell is not without limitations. During its development, it was found that the bus driver cell could be developed to drive the AlphaServer 8000 system bus in 10 ns under a limited number of conditions. When the driver cell asserted a deasserted (near 0 V) bus line or deasserted a bus line that had been asserted (near 1.5 V) for only one cycle, for example, 10-ns timing could readily be met. When the driver attempted to deassert a bus line that had been asserted for more than one cycle by multiple drivers, however, 10-ns timing could not be met. These limitations have significant implications for protocol development. Protocols typically have a number of signals that can be driven by multiple drivers. These may include cache status signals and bus flow control signals. Protocols also typically include a number of signals that can be asserted for many cycles. These may include bank busy signals or arbitration request signals. Clearly the implications are that the limitations of the bus driver/receiver cell would cause the system either to fall short of its cycle time and performance goals or to be incapable of supporting a workable bus protocol.

With the bus driver/receiver cell pushing technology to its limits, the solutions to this problem were extremely limited. The system cycle time could be slowed down to accommodate all signal transitions within a single cycle, regardless of the charge state of the signal line; or a signaling protocol could be developed that would avoid charging a signal to the point where it could not transition in 10 ns; or the physical topology of the system could be reconsidered with the goal of finding a new topology that met the system goals at a slower clock rate. The first option of slowing the clock was clearly unacceptable; it could not satisfy the system's latency and bandwidth goals given the system's topology. The third option could potentially satisfy the system's latency and bandwidth goals, but came at the expense of the favorable qualities of the simple bus outlined in the Topological Layer and at the risk that the new topology would suffer similar, unforeseen pitfalls. The option of developing a signaling protocol, on the other hand, could satisfy the system's performance goals with little or no risk. A signaling protocol was clearly the practical solution to the bus driver/receiver cell limitations.

The Signaling Layer of the platform development model introduces the AlphaServer 8000 signaling protocol. This protocol was developed by creating a list of signal classes, based on driver counts and assertion and deassertion characteristics, and by associating a specific signaling protocol with each class. The signal classes and their protocols are listed in Table 2. As the AlphaServer 8000 primary protocol was developed, each bus signal was assigned a signal class. As AlphaServer 8400/8200 hardware was developed, each bus signal was designed to operate according to the signaling protocol associated with its signaling class. The system bus address and data signals, for example, fall into the second class of signals. As a result, the AlphaServer 8400/8200 modules are designed to leave tristate cycles between each address and data transfer on the system bus.

The AlphaServer system bus cache status signals (TLSB_Shared and TLSB_Dirty) and the system bus flow control signals (TLSB_Hold and TLSB_Arb_Suppress) demonstrate a noteworthy paradigm that results from the AlphaServer 8000 signaling protocol. All these signals are defined such that at times they must be asserted for multiple cycles. All these signals also fall into the fourth signal class, which expressly prohibits driving the signals for multiple cycles. When these two contradictory requirements exist, the result is a class of signals pulsed to indicate multiple cycles of constant assertion. Logic inside each AlphaServer 8000-based module must be designed to convert these pulsed signals to constantly asserted signals within its system bus interface. Note that when signals such as these are discussed in the protocol sections of this paper, the term "asserted" is used to imply constant assertion, with the understanding that the signals may in fact be pulsed.

Consistency Check Layer

The Consistency Check Layer defines a method for maintaining system integrity. Specifically, it defines methods for detecting errors and inconsistencies in the system and, more important, methods for logging errors in the presence of historically disabling errors. Although it does not contribute directly to the AlphaServer 8000 platform's performance goals or stated product goals, the Consistency Check Layer contributes an extremely useful feature to the AlphaServer 8000 products. It is included in the paper for the sake of completeness in the analysis of the seven-layer platform development model.

The AlphaServer 8000-based systems employ a number of error-checking mechanisms. These include transmit checks, sequence checks, assertion checks, and time-outs. If any error is detected by an AlphaServer 8000 module by means of these mechanisms, the module responds by asserting a special "Fault" signal on the AlphaServer 8000 system bus. This Fault signal has the effect of partially resetting all system bus interfaces and processors, and trapping the processors to "machine check" error-handling routines. The partial reset clears all system state, with the exception of error registers. This resynchronizes all system bus interfaces and eliminates all potentially unserviceable transactions left pending in the system. Thus the system can begin execution of the machine-check routines in a reset system. Although the routines are not guaranteed to be able to complete an error log in the presence of an error, it is believed that this mechanism will increase the probability of a successful error log.

Table 2
AlphaServer 8000 Signal Classes

Signal Class	Driver Count and Signal Assertion/Deassertion Characteristics	Signaling Protocol
1	Single driver with multiple receivers	Never driven more than two consecutive cycles
2	Multiple drivers with multiple receivers	Tristate cycle on the bus when driver changes
	One driver at a time	Never driven more than two consecutive cycles
3	Multiple drivers with multiple receivers	Value received on signal deassertion is unpredictable and must be ignored
	Many drivers at once possible	Tristate cycle on the bus when driver changes
	Assertion time may differ from driver to driver	Never driven in two consecutive cycles
	Deassertion time is fixed	
4	Multiple drivers with multiple receivers	Value received on signal deassertion is unpredictable and must be ignored
	Many drivers at once possible	Tristate cycle on the bus when driver changes
	Timing is fixed	Never driven in two consecutive cycles

The AlphaServer 8000 platform's Fault error-handling feature is particularly useful in recovering error state from a computer in a "hung" state. A computer enters a hung state when an error occurs that stops all progress in the computer system. If a processor is waiting for a response to a read, for example, and the read response is not forthcoming due to an error, the system hangs while waiting for the response. The desktop model for error handling would require a system reset to recover from such an error. The process of the system reset, however, would purge error state. The purge, in turn, makes error diagnosis extremely difficult. This desktop model is not unique to desktop systems. It is also employed in server-class machines such as Digital's DEC 7000/10000 AXP systems. Although this model may be acceptable on the desktop, it is most undesirable in an enterprise server system. The AlphaServer 8000-based systems use a time-out counter to detect a hung system and the Fault error-handling technique to recover an error log in the event of a hung system. The result is a robust error-handling system that is appropriate in an enterprise server.

Primary Protocol Layer

The Primary Protocol Layer of the platform development assigns names and characteristics to the various system bus signals and uses these names and characteristics to define higher-order system bus transactions and functions. System bus transactions may include reads of data from memory or writes of data to memory. These transactions are the primary business of a computer system and its protocol. If a system efficiently executes read and write transactions, it will perform better than a system that does not. System bus functions may include mapping memory addresses to specific memory banks or arbitrating for access to system buses. These functions enable system bus transactions to operate in environments with multiple processors arbitrating for access to the system bus and multiple banks of memory.

AlphaServer 8000 system bus transactions relate directly into the platform's performance metrics. The system's memory read latency, for example, is equal to the time it takes for a processor to issue and complete a system bus read transaction. The number of system bus transactions and their associated data that the system bus can process in a given period of time define the system bus bandwidth.

The components of a typical memory read transaction are shown in a timeline in Figure 4. This timeline of components is based on a system that is an abstract of the DEC 7000/10000 AXP systems. To minimize a system's memory read latency, each component of the read transaction timeline must be minimized. Components 1, 3, 7, and 8 of the timeline are simply data and address transfers across buses and through interfaces. The delays associated with these components are largely determined by system cycle time; they cannot be affected by the protocol to any great extent. Component 5 is the DRAM access time. It is minimized by the reconfigurable controllers described in the Operational Layer. The remaining components, (2) address bus arbitration, (4) memory bank decode, and (6) data bus arbitration, fall into the domain of the primary protocol. These elements must be designed to contribute minimal delay to the overall latency.

The effects of protocol on a system's data bandwidth are a little more difficult to quantify than the effects of protocol on memory read latency. In general, the theoretical maximum system bandwidth is equal to either the sum of the bandwidths of the system's memory banks or the maximum system bus bandwidth, whichever is smaller. If the system bandwidth is limited by memory module bandwidth, it is essential to keep as many memory modules active as possible. If, for example, eight banks of memory are required to sustain 100 percent of the maximum system bandwidth, but the system can support only four outstanding commands, only four banks can be kept busy and only 50 percent of the maximum bandwidth can be rendered. In another example, if 10 percent of the time this system freezes all but one bank of memory to perform special atomic functions on special data blocks, the system's bandwidth will suffer nearly a 10 percent penalty (73/80 possible memory accesses versus 80/80 possible memory accesses). If the system bandwidth is limited by the bandwidth of the system bus, the maximum system bandwidth can be achieved only when the protocol allows system modules to drive data onto the system data bus in every available cycle on the data bus. When a processor reads a block

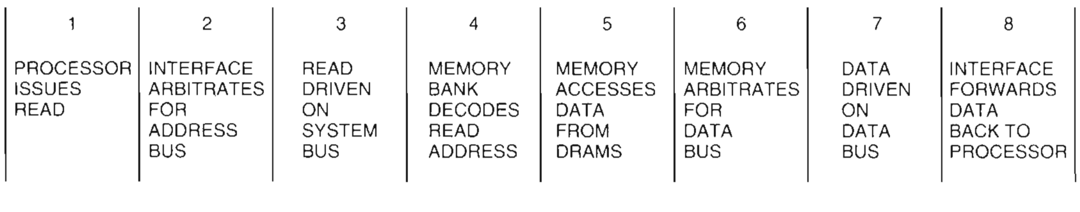

1
PROCESSOR ISSUES READ
2
INTERFACE ARBITRATES FOR ADDRESS BUS
3
READ DRIVEN ON SYSTEM BUS
4
MEMORY BANK DECODES READ ADDRESS
5
MEMORY ACCESSES DATA FROM DRAMS
6
MEMORY ARBITRATES FOR DATA BUS
7
DATA DRIVEN ON DATA BUS
8
INTERFACE FORWARDS DATA BACK TO PROCESSOR

Figure 4
Components of Memory Read Latency

of data from a second processor's cache, for example, the second processor may have to stall the data bus to allow it to drive the read data onto the system's data bus as prescribed by the system protocol. A stall of the data bus translates into unused data bus cycles and degradation of real system bandwidth. Thus to maximize real system bandwidth, system bus and memory bank utilization must be maximized, and stalls in system bus activity and stalls in memory bank activity must be minimized.

The following sections begin with an overview of the basic AlphaServer 8000 platform protocol and how this basic protocol influences system performance. This section is followed by a discussion of how the various protocol components identified as elements of memory read latency (i.e., memory bank mapping, address bus arbitration, and data bus arbitration) affect the latency. These sections conclude with a discussion of subblock write transactions and their effects on system bandwidth.

AlphaServer 8000 Protocol Overview The platform development Topological Layer defined the AlphaServer 8000 system bus as having separate address and data buses. The AlphaServer 8000 system bus protocol defines how system bus transactions are performed using these two buses. According to the protocol, processor and I/O port modules initiate read and write transactions by issuing read and write commands to the system address bus. These address bus commands are followed sometime later by an associated data transfer on the data bus. All data transfers are initiated in the order in which their associated address bus commands are issued. Cache coherency information for each system bus transaction is broadcast on the system bus as each transaction's data bus transfer is initiated. Each data transfer moves 64 bytes of data (only 32 bytes of which are valid for programmed I/O transfers). Figure 5 shows an example of AlphaServer 8000 system bus traffic. In cycle 1 a read transaction, *r*0, is initiated on the system address bus. In cycle X, the data transfer for read *r*0 is initiated on the system data bus by means of the system bus Send_Data signal, the assertion of which is indicated with a value of *i*0. As this data transfer is initiated, the status, *s*0, is also driven on the system bus. In cycle X+2, all system bus modules have an opportunity to stall or to control the flow to the system data bus. In this example, the bus is not stalled, as indicated by a value of n. Finally, given that the bus is not stalled, the 64 bytes of read data associated with read *r*0 are transferred across the system bus during cycles X+5 and X+6. In addition to read *r*0, Figure 5 also illustrates the execution of a write, *w*1, and another read, *r*2. Note that data transfer initiation, data bus flow control, and data transfer are pipelined on the system data bus in the same order as their associated commands were issued to the address bus. Note further that this diagram represents 100 percent utilization of the system data bus (one data transfer every three cycles). With a 10-ns cycle time, this utilization would translate to 2.1 GB per second of bandwidth.

The AlphaServer system address bus uses two mechanisms to control the flow of system bus transactions. First, processor and I/O port modules are not allowed to issue commands to memory modules that are busy performing some DRAM access for a previously issued system bus transaction. The state of each memory bank is communicated to each processor by means of system bus Bank_Available signals. If a processor or I/O port seeks access to a given memory bank and that memory bank's Bank_Available signal indicates that the bank is free, the processor or I/O port may request access to the address bus and, if granted access by the system arbitration logic, issue its transaction to the address bus. If a processor or I/O port seeks access to a given memory bank and that memory bank's Bank_Available signal indicates that the bank is not free, the processor or I/O port will not request access to the system address bus. Thus, unless all memory banks are busy or unless the total of the busy memory banks includes all banks that are needed to service the system's processors and I/O ports, the address bus will continue to transmit commands. The second mechanism for controlling the flow through the address bus is the system bus Arb_Suppress signal. If any system bus module runs out of any command/address-related resource, such as command queue

CYCLE	1	2	3	4	5	•••	X	X + 1	X + 2	X + 3	X + 4	X + 5	X + 6	X + 7	X + 8	X + 9	X + 10	X + 11	X + 12
ADDRESS BUS COMMAND	*r*0		*w*1		*r*2	•••													
DATA BUS SEND_DATA						•••	*i*0			*i*1			*i*2						
CACHE STATUS						•••			*s*0			*s*1			*s*2				
DATA BUS FLOW CONTROL									*n*			*n*			*n*				
DATA BUS						•••						*r*0	*r*0		*w*1	*w*1		*r*2	*r*2

Figure 5
AlphaServer 8000 System Bus Traffic

entries, it can assert this signal and prevent the system arbitration logic from granting any more transactions access to the bus. The Arb_Suppress signal is useful, for example, in a system configuration with 16 memory banks but only eight entries worth of command queuing in a processor.

The AlphaServer 8000 system data bus has its own flow-control mechanism, the system bus Hold signal, which is independent of the address bus flow-control mechanisms. The Hold signal, shown as Data Bus Flow Control in Figure 5, is asserted in response to the initiation of a data bus transfer. Normally, data bus transfers are initiated on the data bus when an AlphaServer 8000 memory module asserts the Send_Data signal. Send_Data is asserted by a memory module based on the state of the module's DRAMs: When servicing a read transaction, the memory will assert Send_Data when its DRAM read is complete; when servicing a write transaction, the memory will assert Send_Data as soon as its turn on the data bus comes up. Five cycles after the assertion of Send_Data, some module drives data onto the data bus. If a module is required to drive data in response to an assertion of Send_Data and is unable to do so, it will assert the Hold signal two cycles after the assertion of Send_Data. This may occur if a processor module must source read data from its cache and cannot fetch the data from the cache as quickly as the memory module can fetch data from its DRAMs. If, on the other hand, a module is required to receive data in response to an assertion of Send_Data and is unable to do so, it too will assert the Hold signal two cycles after the assertion of Send_Data. This may occur if no receiving module's data buffers are available to receive data. Each module that asserts Hold two cycles after Send_Data will continue to assert Hold every other cycle—as prescribed by the AlphaServer 8000 signaling protocol—until it is ready for the data transfer. Three cycles after all modules are ready and deassert the Hold line, data is finally transferred. Figure 6 shows a read, *r*0, that experiences one pulse of the system bus Hold signal.

It is important to note that the address bus and the data bus have independent means and criteria for initiating transactions and controlling the flow of transactions. The address bus initiates address bus commands based on processor and I/O port module requests and controls the flow based on the state of address-related resources. The data bus initiates data transfers in the same order as the address bus transmitted commands by means of the Send_Data signal. Send_Data is usually asserted by a memory module based on the state of the module's DRAMs. The data bus flow is controlled based on the state of various data-related resources. The differing means and criteria for initiation and flow control allow the two buses to operate almost independently of one another. This independence translates into performance because it allows the address bus to continue to initiate commands even as the data bus may be stalled because of a conflict. Continuous command initiation translates into more continuous system parallelism and thus more system bandwidth. Figures 6 and 7 illustrate this point. Both figures illustrate systems that are issuing a series of processor reads to blocks that must be sourced from another processor's cache. In both cases, processors require two more cycles than main memory banks to source read data. As such, two cycles of Hold assertion must periodically occur on the data bus. Figure 6 illustrates the operation of the AlphaServer 8000 system bus, showing that although the data bus had to be held in cycle 6, the address bus was able to continue issuing commands. As a result, each processor sourcing data begins its read of cache data as soon as possible and is guaranteed to be ready to drive data without Hold cycles when its turn comes up on the data bus. With the illustrated series of five reads, the two Hold cycles result in a 12 percent degradation in system bandwidth. If the series of reads is lengthened toward infinity, the percent of degradation approaches 0. Figure 7 illustrates the operation of a rigidly slotted bus, like that of the DEC 7000/10000 AXP system, normalized to the AlphaServer 8000 topology. As shown, each time the data bus is stalled, so too is the

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ADDRESS BUS COMMAND	*r*0		*r*1		*r*2		*r*3		*r*4															
DATA BUS SEND_DATA				*i*0					*i*1			*i*2			*i*3			*i*4						
CACHE STATUS						*s*0		*s*0			*s*1			*s*2			*s*3			*s*4				
DATA BUS FLOW CONTROL						H		*n*			*n*			*n*			*n*			*n*				
DATA BUS									–	–	*r*0	*r*0		*r*1	*r*1		*r*2	*r*2		*r*3	*r*3		*r*4	*r*4

Figure 6
Read with One Cycle of Hold — Five Reads Sourced by a Processor

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ADDRESS BUS COMMAND	*r*0			*r*1			*r*2			–	–	*r*3			*r*4									
DATA BUS SEND_DATA				*i*0			–	–	*i*1			*i*2			*i*3			–	–	*i*4				
CACHE STATUS						*s*0		*s*0			*s*1			*s*2			*s*3		*s*3			*s*4		
DATA BUS FLOW CONTROL						H		*n*			*n*			*n*			H		*n*			*n*		
DATA BUS									–	–	*r*0	*r*0		*r*1	*r*1		*r*2	*r*2		–	–	*r*3	*r*3	

Figure 7
Five Reads Sourced by a Processor in a Rigidly Slotted System

address bus. This prevents the fourth and fifth reads from getting the headstart necessary to prevent subsequent stalls of the data bus. The result is a 20 percent degradation in performance for the five reads illustrated. If the series of reads is lengthened toward infinity, the percent of degradation settles to 18 percent. Clearly the AlphaServer 8000 approach produces superior data bandwidth characteristics.

It is also important to note that the AlphaServer 8000 address bus and data bus have different maximum bandwidths. Commands can be issued to the address bus every other cycle. With a 10-ns cycle time, this translates into 50 million commands per second. The data bus, on the other hand, can transfer one block of data every three cycles. With a 10-ns cycle time, this translates into 33.3 million data blocks per second. This excess of address bus bandwidth is useful in the development of low-latency arbitration schemes.

Memory Bank Mapping Digital's previous server systems, like the VAX 6000 series and the DEC 7000/10000 AXP series, have employed a common approach to address-to-memory-bank mapping. In this approach, all memory modules implement address range registers. As commands and addresses are transmitted across the system bus, the memory banks compare the addresses against their address range registers to determine if they must respond to the command. An address range comparison can involve a significant number of address bits and, as a result, can become logically complex enough to consume two 10-ns cycles of time. These two cycles can be added directly to memory read latency.

The low-latency focus of the AlphaServer 8000 platform prompted a change in bank mapping schemes. In AlphaServer 8000 systems, the address range registers have been moved onto the processor and I/O port modules. The range registers output a 4-bit bank number that is shipped across the system bus with each command and address. Each memory bank compares each bank number transmitted across the system bus to 4 bits in a programmable bank number register to determine if it should respond to the system bus command.

This bank mapping logic configuration helps to reduce AlphaServer 8000 memory read latency. Because the bank mapping is done on the nodes that issue commands to the address bus, the lengthy address comparison can be done in parallel with address bus arbitration, eliminating its two-cycle delay from the memory read latency. The address comparison traditionally done in the memory bank logic is now replaced with a simple 4-bit comparison, which can easily be done in a single cycle. The overall effect is that the AlphaServer 8000 bank mapping protocol consumes at least one cycle less than Digital's traditional bank mapping protocol. This equates to one less cycle—10 ns minimum—of memory read latency.

Address Bus Arbitration AlphaServer 8000 systems employ a distributed, rotating-priority arbitration scheme to grant access to their address buses. Processor and I/O port modules request access to the address bus based on requests from microprocessors and I/O devices, and on the state of the system's memory banks, as described in the section AlphaServer 8000 Protocol Overview. Each module evaluates the requests from all other modules and, based on a rotating list of module priorities, determines whether or not it is granted access to the bus. Each time a module is granted access to the bus, its priority is rotated to the lowest priority spot on the priority list.

The AlphaServer 8000 arbitration scheme operates in a pipelined fashion. This means that modules request access to the bus in one cycle, arbitrate for access to the bus in the next cycle, and finally drive a command and address onto the bus one cycle later. In terms of processor-generated read requests, this means that, at best, a system bus read command can be driven onto the system address bus two cycles after its corresponding cache read miss is generated on the processor module. This adds two cycles of delay to the memory read latency.

To reduce memory read latency in components associated with address bus arbitration, the AlphaServer 8000 platform employs a technique called "early arbitration." Early arbitration allows a module to request access to the address bus before it has determined if it really needs access to the data bus. If the module is granted access to the address bus but determines that it does not need or cannot use the access, it will drive a No-Operation or NoOp command in the command slot that it is granted. This feature is particularly useful on processor modules. It allows a processor to request access to the bus for a read command in parallel with determining if the read command will hit or miss in the processor's cache. If the read results in a cache hit and the processor is granted access to the address bus, then the processor issues a NoOp command. If the read results in a cache hit and the processor is not granted access to the address bus, the processor discontinues requesting access to the bus. When applied in this manner, this feature can remove two cycles of delay from the memory read latency. This feature is also key to the AlphaServer 8000 memory bank decode feature that allows address-to-memory bank decode to proceed in parallel with system bus arbitration. This is to say, it allows a processor or I/O port module to request access to the address bus before it can determine which memory bank it is trying to access and before it can determine if that memory bank is available. If a module is granted access to the bus and the bank it is trying to access is not available, then the module issues a NoOp command. If a module is not granted access to the bus and the bank it is trying to access is not available, then the module discontinues requesting access to the bus until the bank becomes available. When applied this way, this feature eliminates at least one cycle from the memory read latency, as described in the section Memory Bank Mapping.

The excess address bus bandwidth noted in the protocol overview allows some amount of early arbitration to take place without affecting system performance. When system traffic increases, however, excessive early arbitration can steal useful address bus slots from nonspeculative transactions and as a result degrade bus bandwidth. In fact, in certain pathological cases, excessive early arbitration by modules with high arbitration priority can permanently lock out requests from lower priority modules. To eliminate the negative effect of early arbitration, the AlphaServer 8000 employs a technique called "look-back-two" arbitration. This technique relies on the fact that modules must resolve all cache miss or bank availability uncertainties for early arbitrations within the two cycles required for an early request and its arbitration. This fact implies that any module that has been requesting access to the address bus for more than two consecutive cycles is requesting in a nonspeculative manner. As such, the AlphaServer 8000 arbiter keeps a history of address bus requests and creates two prioritized groups of requests based on this history. It creates a high-priority group of requests from those requests that have been asserted for more than two cycles and a low-priority group of requests from those requests that have been asserted for two cycles or less. It applies the single set of rotating priorities, described above, to both sets of requests. If there are any requests in the high-priority group, the arbiter selects one of these based on the rotating priority set. If there are no high-priority requests, the arbiter selects a request from the lower priority group based on the rotating priority set. This functionality limits early arbitration to only those times when there are nonspeculative requests in the system. It allows the AlphaServer 8000 platform to take advantage of latency gains associated with early arbitration and processor and I/O port based bank decode, without degrading bandwidth in the process.

Data Bus Arbitration The AlphaServer 8000 data bus transfers blocks of data in the same order that the commands corresponding to those blocks are issued on the address bus. This eliminates data bus arbitration per se. In-order data return is accomplished by a simple system of counters and sequence numbers. Each time a command is issued to the address bus, it is assigned a sequence number. Sequence numbers are assigned in ascending order. Each time a block of data is driven on the data bus, a data bus counter is incremented. Each module waiting to initiate a data transfer in response to some address bus command compares the sequence number associated with its command with the data bus counter. When a module's sequence number matches its data bus counter, it is that module's turn to initiate a data bus transfer.

It is arguable that in-order data return is not the optimum data scheduling algorithm. If the scenario shown in Figure 6 were reshaped such that only read *r*0 sourced data from another processor and the penalty for sourcing data from a processor were more severe—a longer data bus Hold requirement—the result would be more significant bandwidth degradation. This new scenario is illustrated in Figure 8. With more efficient data scheduling, it is conceivable that data bus utilization could be improved by using data slots abandoned under the sizable Hold window in Figure 8. The latter scenario is illustrated in Figure 9. Clearly the system in Figure 9 has improved upon the bandwidth of the system in Figure 8.

What Figure 9 cannot show are all the implications of out-of-order data transfers. With as many as 16 outstanding transactions (8 in the AlphaServer

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ADDRESS BUS COMMAND	*r*0		*r*1		*r*2		*r*3		*r*4															
DATA BUS SEND_DATA				*i*0											*i*1			*i*2			*i*3			*i*4
CACHE STATUS						*s*0		*s*0		*s*0		*s*0		*s*0			*s*1			*s*2			*s*3	
DATA BUS FLOW CONTROL						H		H		H		H		*n*			*n*			*n*				
DATA BUS									–	–	–	–	–	–	–	–	*r*0	*r*0		*r*3	*r*3		*r*4	*r*4

Figure 8
Bandwidth Degradation as a Result of In-Order Data Transfers

CYCLE	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
ADDRESS BUS COMMAND	*r*0		*r*1		*r*2		*r*3		*r*4															
DATA BUS SEND_DATA						*i*1			*i*2			*i*0			*i*3			*i*4						
CACHE STATUS								*s*1			*s*2			*s*0			*s*3			*s*4				
DATA BUS FLOW CONTROL								*n*			*n*			*n*			*n*			*n*				
DATA BUS											*r*1	*r*1		*r*2	*r*2		*r*0	*r*0		*r*3	*r*3		*r*4	*r*4

Figure 9
Improved Bandwidth with Out-of-Order Data Transfers

8400/8200) active in the system at any one time, the task of producing a logic structure capable of retiring the transactions in order is enormous. Furthermore, the retiring of transactions out of order complicates the business of maintaining coherent, ordered memory updates. Finally, it was felt that the parallelism made possible by the independent address and data bus would help to mitigate many of the negative effects associated with the in-order data transfers. For these reasons, a practical decision was taken to transfer data on the system data bus in the order that the associated commands were issued to the system address bus.

Subblock Writes To support a range of I/O subsystems, AlphaServer 8000 I/O port modules must support writes of data as small as longwords (32 bits), words (16 bits), and bytes. Given the AlphaServer system bus block size of 64 bytes, these writes are referred to as subblock writes. The execution of a subblock write consists of reading a block of data from a system memory bank, overwriting just the portion of the block addressed by the subblock write, and writing the entire block back to memory. The difficulty with performing this operation arises when a "third-party" module—defined here as a module other than the one performing the subblock write—modifies the block between the read portion of the subblock write and the write portion of the subblock write. To correctly complete the subblock write, the I/O port module must merge the subblock write data into the block as it was after the third-party module modified it. This problem can be resolved in one of two ways: (1) by means of a small cache on the I/O port module that updates the I/O port's copy of the block based on the third-party write, or (2) by means of an atomic read-modify-write that disallows the third-party write altogether.

In an ideal world, I/O port modules would implement a small one-block cache for the purpose of subblock writes. This cache would allow the I/O module performing the subblock write to update its copy of the block targeted by the subblock write with modified data from third-party modules. Unfortunately, not all processors broadcast modified data to the system. Many processors, for example, use a read-invalidate protocol. In a read-invalidate protocol, when a processor wishes to modify a block, it issues a command that invalidates all other copies of that block in the system and then modifies the block of data in its cache. If such an invalidate command invalidated the block in an I/O port module's subblock write cache, the I/O port module would be forced to re-read the block. There is no guarantee, however, that another invalidate will not occur between the re-read of the block and the write of merged data back to memory.

As such, the I/O port module may never be able to complete the subblock write. I/O port caching is therefore not a workable solution.

Atomic read-modify-write sequences disallow third-party writes to a given block between the read portion of a subblock write and the write portion of a subblock write. As such, the atomic read-modify-write sequence does guarantee the timely completion of a subblock write. Implementations of atomic read-modify-write sequences are designed to disallow accesses to some size portion of the memory region that contains the subblock address, between the read and write portions of the subblock write. The size of the memory region can vary from a single block of data to a single bank of memory to the entirety of memory. If the size of the memory region is small, such as a single data block, design complexity is significant; but the impact of locking out access to a single block of memory is insignificant to bandwidth. Conversely, if the size of the memory region is large, such as the entirety of memory, design complexity is insignificant; but the impact of locking out accesses to the entirety of memory for any period of time can be significant to system bandwidth.

The AlphaServer 8000 platform supports atomic read-modify-write sequences by locking out accesses within memory-bank-sized memory regions. This middle ground memory-region size provides the AlphaServer 8000 with a practical balance between design complexity and system bandwidth. The AlphaServer 8000 platform implements memory bank granularity atomic read-modify-write accesses by means of special Read_Bank_Lock and Write_Bank_Unlock address bus commands, and by leveraging the existing memory bank flow control mechanisms. Specifically, Read_Bank_Lock commands function like normal read commands, except that their targeted memory banks are left busy after the read transaction is complete. Memory banks locked by Read_Bank_Lock commands remain busy until a Write_Bank_Unlock command is issued from the same module that issued the Read_Bank_Lock command. While a memory bank is busy, no module other than the module that locked the bank by means of a Read_Bank_Lock command will even request access to the bank, as required by standard arbitration protocol. This approach provides for atomic read-modify-write sequences and coherent subblock writes. This protocol works regardless of the number of I/O modules in the system and regardless of arbitration priorities.

Superset Protocol Layer

The AlphaServer 8000 primary protocol provides all the basic constructs required to perform basic system functions, such as memory reads and writes, local register reads and writes, and mailbox-based I/O register reads and writes. The protocol performs these basic functions with a high level of efficiency and performance. Some additional functionality, such as PCI direct-programmed I/O register accesses, can be functionally satisfied by the primary protocol but cannot be satisfied in a way that does not severely degrade the performance of the entire AlphaServer 8000 system. As such, the AlphaServer 8000 platform allows for Superset Protocols, i.e., protocols that are built upon the basic constructs (reads and writes) of the AlphaServer 8000 primary protocol.

PCI direct-programmed I/O register reads can take more than a microsecond to complete. If these reads were completed by means of the AlphaServer 8000 nonpended, strictly ordered primary protocol, the AlphaServer system data bus would be stalled for a full microsecond each time a PCI programmed I/O read was executed. Such stalls would have a disastrous effect on system bus bandwidth and system performance.

The PCI programmed I/O problem is solved on the AlphaServer 8000 platform by implementing a PCI-specific pended read protocol using the simple read and write commands already included in the basic AlphaServer 8000 primary protocol. This special superset protocol works as follows:

- When a microprocessor issues a PCI programmed I/O read, the read is issued to the AlphaServer 8000 system bus as a register read. This read is pended with a unique identification number that is associated with the issuing processor by driving the identification number on the system bank number lines when the register read command is issued to the system address bus. The bank number lines are otherwise unused during register accesses. The issuing processor also sets a flag, indicating that it has issued a PCI programmed I/O read command.
- The I/O port module interfacing to the addressed PCI local bus responds to the register read by forwarding the read to the PCI, storing the processor identification number specified by the address bus bank number lines and driving "dummy data" onto the data bus in the register read's associated data slot. The value of the dummy data is irrelevant; it is ignored by all system bus modules and is typically whatever was left in the I/O ports register read buffer as a result of the last read it serviced.
- When the PCI local bus returns read data to the I/O port module, the I/O module issues a register write to a special PCI read-data-return register address on the system bus. This write is pended

with the issuing processor's identification number, which was stored by the I/O port module. This identification number is again pended by driving it onto the system bank number lines as the register write command is issued to the system address bus. The PCI read data is returned in the data cycle associated with this register write.

- When a processor module identifies a register write that addresses the PCI read-data-return register address, it checks the state of its PCI read flag and compares the value driven in the system bank number lines with its unique identification number. If the PCI read flag is set and the value on the bank number lines matches the processor's identification number, then the processor completes the PCI programmed I/O read with the data supplied by the register write.

The AlphaServer 8000 PCI programmed I/O read superset protocol allows AlphaServer 8000 systems to complete PCI programmed I/O reads without stalling system buses. Furthermore, it allows AlphaServer systems to support PCI I/O in such a way that system bus modules not participating in the superset transaction need not be alerted to the presence of special bus transactions and therefore need not contain logic that recognizes and responds to these special cases. This approach demonstrates a practical way to simplify overall system design without affecting system performance.

AlphaServer 8400 and AlphaServer 8200 Systems

The AlphaServer 8400 and 8200 systems are the first products based on the AlphaServer 8000 platform. The AlphaServer 8200 system is an "open office"-class server (i.e., the AlphaServer 8200 can be located in any office area, for example, where photocopier machines are typically placed). It features up to six system bus modules in an industry-standard 47.5-centimeter (19-inch) rackmount cabinet. The 8200 system can support up to six 300-MHz Alpha 21164 microprocessors, 6 GB of main memory, and 108 PCI I/O slots. The AlphaServer 8400 system is an "enterprise"-class server (i.e., a machine on which a business can be run). It features up to nine system bus modules in a DEC 7000-style cabinet. It can support up to twelve 300-MHz Alpha 21164 microprocessors, 14 GB of main memory, and 144 PCI I/O slots.

The clock frequencies of both the AlphaServer 8400 system and the AlphaServer 8200 system are determined by the clock frequency of the 300-MHz (3.33-ns cycle time) Alpha 21164 microprocessor chip. Both systems use a 4× clock multiplier to arrive at a system clock frequency of 75 MHz (13.3-ns cycle time). At this speed, the systems feature 265-ns minimum read latencies and 1,600 MB/s of data bandwidth.

Both systems are based on the same set of AlphaServer 8000 architecturally compliant system bus modules. In addition, both systems support a new PCI I/O subsystem designed specifically for these classes of systems. The constituent modules and I/O subsystems that compose the AlphaServer 8400 and the AlphaServer 8200 systems are as follows.

TLEP Processor Module—Each TLEP processor module supports two 300-MHz Alpha 21164 microprocessors. Each Alpha 21164 processor is paired with a 4-MB external cache. This cache is constructed with 10-ns asynchronous SRAMs. The cache latency to first data is 20 ns, and with one 3.33-ns processor cycle of wave pipelining, its maximum bandwidth is 915 MB/s. The TLEP module operates with a 75-MHz (13.33-ns cycle time) clock frequency.

TMEM Memory Module—Each TMEM memory module is implemented with two equal-sized DRAM banks. TMEM modules are available in 128-MB, 256-MB, 512-MB, 1024-MB, and 2048-MB sizes. The TMEM module is designed to operate at a 100-MHz (10-ns cycle time) clock frequency.

TIOP I/O Port Module—The TIOP module interfaces the AlphaServer 8000 system bus to four I/O channels, called "hoses." Each hose can interface to one XMI, Futurebus+, or PCI/EISA I/O subsystem. Each TIOP can support up to 400 MB/s of I/O data bandwidth and is designed to operate at a 100-MHz (10-ns cycle time) clock frequency.

ITIOP Integrated I/O Port Module—The ITIOP module interfaces the AlphaServer 8000 system bus to one hose I/O channel and one semipreconfigured PCI local bus, which is integrated onto the ITIOP module. The integrated PCI bus features one single-ended small computer systems interface (SCSI) controller, three Fast Wide Differential SCSI controllers, one NI port, and optional FDDI and NVRAM controllers. Each ITIOP can support up to 200 MB/s of I/O data bandwidth and is designed to operate at a 100-MHz (10-ns cycle time) clock frequency.

PCIA PCI I/O Subsystem—The PCIA PCI I/O subsystem consists of hose-to-PCI adapter logic and a 12-slot PCI local bus. This 12-slot bus is created from 4-slot PCI buses interfaced such that they appear as a single bus. The high slot count provides the connectivity essential in an enterprise-class server. The PCIA optimizes direct memory access (DMA) reads by means of the PCI Read_ Memory_Multiple command. The Read_Miss_Multiple command allows the PCIA to stream DMA read data from memory to the PCI bus. Consequently, the PCIA can increase DMA read bandwidth, offsetting any latency penalties that result from the AlphaServer 8000 platform's multilevel I/O architecture. The PCIA's adapter logic includes a 32K entry map RAM for converting PCI addresses (32 bits) to AlphaServer 8000 system bus addresses (40 bits). This map RAM features a five-entry, fully associative translation cache.

AlphaServer 8400 and AlphaServer 8200 Performance

A number of performance benchmarks have been run on the AlphaServer 8400 and AlphaServer 8200 systems. The results of some of these benchmarks are summarized in Table 3.

The AlphaServer SPECint92 and SPECfp92 ratings demonstrate outstanding performance. In both ratings, the AlphaServer 8400 system performance is over 3.5 times the ratings of the HP9000-800 T500 system. The SPECfp92 rating of 512 is 1.4 times its nearest competitor, the SGI Power Challenge XL system. Similarly, a six-processor AlphaServer 8400 system achieves the same 1,900 million floating-point operations per second (MFLOPS) as an eight-processor SGI Power Challenge XL system. Finally, the AlphaServer 8400 system's 5-GFLOPS Linpack n×n result is beyond the performance of all other open systems servers, placing the AlphaServer at supercomputer performance levels with systems such as the NEC SX-3/22 system and the massively parallel Thinking Machines CM-200 system.

Acknowledgments

Several members of the AlphaServer 8000 Development Team in addition to the authors were key contributors to the generation of this technical article. These individuals are John Bloem, Elbert Bloom, Dick Doucette, Dave Hartwell, Rick Hetherington, Dale Keck, and Rich Watson.

Table 3
AlphaServer 8400 and 8200 System Performance Benchmark Results

Benchmark Name	Processor Count	Units	AlphaServer 8200	AlphaServer 8400
SPECint92	1	–	341.4	341.4
SPECfp92	1	–	512.9	512.9
SPECrate_int92	1	–	8551	8551
	6	–	50788	50788
	12	–	not applicable	91580
SPECrate_fp92	1	–	11981	11981
	6	–	71286	71286
	12	–	not applicable	140571
Linpack 100×100	1	MFLOPS	140.3	140.3
Linpack 1000×1000	1	MFLOPS	410.5	410.5
	6	MFLOPS	1821	1902
	8	MFLOPS	not applicable	2282
	12	MFLOPS	not applicable	2675
Linpack n×n	1	MFLOPS	428.3	428.3
	6	MFLOPS	2445	2445
	12	GFLOPS	not applicable	5.0
AIM III Performance Rating	8	AIMs	not applicable	1649.8
AIM III User Loads	8	Maximum quantity	not applicable	9384
AIM III Throughput	8	Jobs/min	not applicable	16168.2
McCalpin Copy	1	MB/s	not available	186.29
	8	MB/s	not applicable	898.61
McCalpin Scale	1	MB/s	not available	174.4
	8	MB/s	not applicable	829.74
McCalpin Sum	1	MB/s	not available	198.3
	8	MB/s	not applicable	891.84
McCalpin Triad	1	MB/s	not available	195.15
	8	MB/s	not applicable	982.13

References

1. W. Bowhill et al., "Circuit Implementation of a 300-MHz, 64-bit Second-generation CMOS Alpha CPU," *Digital Technical Journal,* vol. 7, no. 1 (1995, this issue): 100–118.

2. S. Saini and D. Bailey, "NAS Parallel Benchmarks Results 3-95," Report NAS-95-011 (Moffet Field, Calif.: Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, saini@nas.nasa.gov, April 1995).

3. J. Dongarra, "Performance of Various Computers Using Standard Linear Equations Software," Document Number CS-89-85, available on the Internet from Oak Ridge National Laboratory, netlib@ornl.gov, April 13, 1995.

4. Z. Cventanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance Using TP and SPEC Workloads," *Proceedings of the 1994 International Symposium on Computer Architecture:* 60–70.

5. J. Nicholson, "The RISC System/6000 SMP System," *COMPCON '95,* March 1995: 102–109.

6. L. Staley, "A New MP HW Architecture for Technical and Commercial Environments," *COMPCON '95,* March 1995: 129–132.

7. B. Allison and C. van Ingen, "Technical Description of the DEC 7000 and DEC 1000 AXP Family," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 100–110.

8. L. Gwennap, "Intel's P6 Uses Decoupled Superscalar Design," *Microprocessor Report,* February 16, 1995: 15.

9. J. Basmaji et al., "Digital's High-performance CMOS ASIC," *Digital Technical Journal,* vol. 7, no. 1 (1995, this issue): 66–76.

10. R. Watson, H. Collins, and R. Iknaian, "Clock Buffer Chip with Absolute Delay Regulation Over Process and Environmental Variations," *1992 Custom Integrated Circuits Conference,* paper 25.2: 1–5.

11. E. Davidson, "Delay Factors for Mainframe Computers," *Proceedings of the 1991 Bipolar Circuits and Technology Meeting:* 116–123.

12. D. Cox et al., "VLSI Performance Compensation for Off-Chip Drivers and Clock Generation," *Proceedings of IEEE 1989 Custom Integrated Circuits Conference:* 14.3.1–14.3.4.

13. D. Chengson et al., "Dynamically Tracking Clock Distribution Chip with Skew Control," *1990 Custom Integrated Circuits Conference Proceedings:* 15.6.1–15.6.4.

14. M. Johnson et al., "A Variable Delay Line Phase Locked Loop for CPU—Coprocessor Synchronization," *ISSCC88 Proceedings:* 142–143.

Biographies

David M. Fenwick
Dave Fenwick is the AlphaServer 8000-series system architect. As leader of the advanced development group and of the design team, he has been responsible for definition of the product and its characteristics, and for the system implementation. Dave moved from Digital's European Engineering organization in 1985 to join the U.S.-based VAXBI program and subsequently was processor architect for the VAX 6000 vector processor. A consulting engineer, he holds 3 major U.S. patents and has 13 patent applications pending. He received an Honours Degree in electrical and electronic engineering from Loughborough University of Technology, United Kingdom.

Denis J. Foley
A principal hardware engineer in the AlphaServer group, Denis is the project leader for the TLEP CPU module. He joined Digital in Clonmel, Ireland, in 1983 after receiving a bachelor's degree in electrical engineering from University College Cork, Ireland. He has contributed to the development of several communications and computing projects. Currently, he is working on the design of a CPU module for the AlphaServer 8000 platform that is based on the next generation of the Alpha microprocessor. Denis is listed on 12 patent applications that relate to his work on the AlphaServer CPU and bus designs.

William B. Gist
Bill Gist's recent responsibility was the development of the high-performance I/O system bus circuit architecture for the AlphaServer 8000-series ASICs. A principal engineer and a member of the Server Platform Development Group,

he is currently developing high-performance I/O architectures for low-cost plastic packaging technologies. Joining Digital in 1977, he began work on PDP-11 systems development and later became a member of the VAX 6000-series engineering team, focusing on clock chip development and vector processor ASIC development. Bill has a B.S. degree in electrical engineering from Worcester Polytechnic Institute and holds three patents for the AlphaServer 8000-series I/O circuit architecture.

Stephen R. VanDoren
In 1988, Steve VanDoren came to Digital to work with the VAX 6000 vector processor design team. He later joined an advanced development team responsible for evaluating system technology requirements for what would become the AlphaServer 8000 series of products. During the AlphaServer project, he lead the design of the address interface on the TLEP processor module. He is listed as a coinventor on 10 patents filed on the AlphaServer 8000-series architectural features. Steve is currently working on new server processor designs. He is a member of Eta Kappa Nu and Tau Beta Pi and holds a B.S. degree in computer systems engineering from the University of Massachusetts.

Daniel Wissell
Consulting engineer Dan Wissell has more than 20 years of computer industry experience in analog and digital circuit design and test. While at Digital, he has worked on the VAXcluster and DEC 7000/1000 systems development teams, and more recently he contributed to the AlphaServer 8000-series design effort. He is recognized within Digital as an expert in the areas of distributed power systems, on-module energy management, and high-speed clock systems. Dan holds three patents and has filed several patent applications for his work on current and future Digital products. He has degrees in engineering from Kean College and the Milwaukee School of Engineering.

Digital's High-performance CMOS ASIC

Jean H. Basmaji
Kay R. Fisher
Frank W. Gatulis
Herbert R. Kolk
James F. Rosencrans

A high-performance ASIC has been developed to serve as the interface for the 10-ns bus in the new AlphaServer 8000 series server systems from Digital. The CMOS standard-cell alternative (CSALT) technology provides a timing-driven layout methodology together with a correct-by-construction approach for managing the complex device physics issues associated with state-of-the-art CMOS processes. The timing-driven layout is coupled with an automated standard-cell design approach to bring the complete design process directly to the logic designer.

Today, high-performance microprocessors designed with complementary metal-oxide semiconductor (CMOS) processes are much more demanding on the support logic used to interface them to the rest of the system. Microprocessors, like Digital Semiconductor's Alpha 21164 chip, are extending the external logic cycle times to the point where custom-integrated circuits are necessary to realize the full performance potential. The CMOS standard-cell alternative (CSALT) technology developed at Digital satisfies these high-performance needs without resorting to a complex, custom design process.

CSALT technology provides a timing-driven layout methodology together with a correct-by-construction approach for managing the complex device physics issues associated with state-of-the-art CMOS processes. The timing-driven layout is coupled with an automated standard-cell design approach to bring the complete design process directly to the logic designer. Using CSALT, logic designers can take their application-specific integrated circuit (ASIC) designs from a concept on their desktops to a completed layout that is ready for fabrication.

Other design approaches address portions of the process, but the CSALT tool suite is complete and automated. Many ASIC vendors transfer the logic designs to a different set of engineers, using different tools and skills, to complete the physical implementation before post-layout timing analysis can take place. Any problems encountered after the layout tend to result in the design being returned to the logic designers. The artificial boundary erected between logic designers and layout implementers can result in delays. In complex designs, multiple iterations may be necessary before the design converges into an acceptable solution. This convergence process becomes more complicated with the introduction of synthesized logic, because the process is extended to include the synthesis tools.

CSALT's timing-driven methodology eliminates the need for the many chip layout specialists and ASIC vendor experts who normally complete a multichip project. In addition, the timing-driven methodology eliminates the need for the traditional chip floorplanning step in which the designer maps the logical design onto the physical chip architecture. The floorplanning step often becomes a critical and time-consuming effort when the design is being optimized for performance.

The automated and batch-driven CSALT methodology can turn a logically complete design into a working, timing-correct chip layout within three compute-intensive days. Previous platform development projects used industry-standard ASICs, manual layouts, and hundreds of manual cell placements to meet the tight design timing requirements within their high-performance ASICs. These methods typically added months to the layout phase of these projects. CSALT's timing-driven layout was specifically developed to address these high-performance requirements and to make the complete design process available to logic designers.

This paper discusses some implementation pieces of CSALT technology and emphasizes the unique timing-driven approach and results. It explains the goals that were established for CSALT development as well as several features of the physical technology. The paper concludes with a discussion of the layout process operations and the process controller.

The Need for CSALT

During the technology evaluation phase of the AlphaServer 8000 series platform, various ASIC technology vendors were evaluated and compared against the aggressive performance needs demanded by the platform's designs and the customization that was necessary within these technologies to meet system bus timing. Based on the experience of developing designs for the previous platform generation and due to the anticipated months of iterative and interactive manual place and route necessary to meet timing, it became clear that technology was a high-risk item to the program. Requirements for the AlphaServer 8000 series systems exceeded the performance capabilities of existing ASIC technologies and the available CAD tools. In addition, access to the internal silicon structure of the ASICs was required to customize bus interface drivers. The risk and cost of developing these capabilities through working with ASIC vendors would have added months of valuable schedule time to the program.

As a result, the decision was made to focus the effort on CSALT technology and to move it from its advanced development stage to a production-quality one. Given the selection criteria that were emphasized, a set of goals was established for the CSALT development:

- Incorporate an integrated timing-constrained driven placement.
- Implement technology in a 3.3-volt (V) stable CMOS process.
- Eliminate chip floorplanning and let timing constraints drive the placement.
- Eliminate manual interaction in the tools to reduce design time and defects.
- Develop very conservative layout rules to eliminate the need for cross talk and electromigration analysis.
- Automate the development and characterization of cell elements including thorough checking.
- Deliver more robust and accurate prediction of chip performance through integrated SPICE simulation and expanded cell library performance tables.[1]
- Use proven algorithms and software whenever possible.

Overview and Description

The front-end logic design and verification process is based on the ASIC standard tools for gate array design that include schematics capture, timing and logic verification, pre-layout delay estimation, and post-layout delay feedback and analysis. The performance data for the library elements is housed in lookup tables that have multiple slope/intercept data entries based on output drive loading as well as input edge rate delay correction factors. Unique delays are calculated for each cell instance. CSALT supports a low-skew balanced clock distribution net.

The back-end layout tools for CSALT include several internally generated tools as well as research tools from academia. The heart of the place-and-route process is TimberWolf from the University of Washington.[2] One of the important features of the TimberWolf tools is their ability to be constraint-driven. These constraints are automatically generated from the timing verification step and then passed to the TimberWolf tools. TimberWolf prioritizes these critical path nets during the placement process in an attempt to meet the timing requirements. Constraints can also be manually generated through a separate user-generated file that feeds into the process. Once parameter files and constraint files are established, the place-and-route process proceeds in a completely automated and batch-driven mechanism all the way to a completely verified design layout file (DLF). The speed of the process execution is limited only by the batch queues available and the performance of the underlying processor type.

The silicon fabrication process relies on Digital Semiconductor's CMOS line. All the physical design and process fabrication rules are built into the layout tools and driven through the parameter files specified at start-up. CSALT has built-in correct-by-construction custom design rules that guarantee all aspects of the automated layout to be free from any design rule violation. The tools account for all aspects of the physical design, such as electromigration rules, coupling capacitance effects on timing, as well as analysis of any electrical hot spots resulting from excessive logic switching in a dense localized area.

Physical Technology

The ASIC designs targeted for this technology needed to meet the physical, electrical, and thermal requirements of the AlphaServer 8000 series platform. The system functions that the ASIC designs satisfy belong to three classes:

- Class 1—Interface between the system bus and the CPU
- Class 2—Interface between the CPU and the local I/O
- Class 3—Interface between the local I/O and the Peripheral Component Interconnect (PCI)

An enhanced ASIC design style was used to reduce the time to market and to minimize design and verification resources. The enhancements to the conventional ASIC design (such as timing-driven layout and automated incorporation of SPICE delays) significantly improved ease of design for high-performance, 100-megahertz (MHz) very large-scale integration (VLSI) chips.[1]

Several features of the CSALT physical technology and their advantages are discussed in the following sections.

Low-skew Clock Distribution

There is one low-skew, single-phase clock net distribution available to the user. This is implemented through three stages. First, the buffered input clock receiver drives two high-power cells located on opposite sides of the chip. In the second stage, the high-power cells drive a central trunk that bisects the die and delivers the clock signals to each half row. In the third stage, separate local clock buffers in each half row are connected to the central trunk and deliver the clock signals to all logic elements in that particular half row.

Skew in this distributed net is controlled through automatic load balancing on the local clock buffers along each row. Cell capacitive loads are calculated for each row, and appropriate balance cells are added to bring the capacitive loads to a predefined value. This method equalized delays across the chip with less than 100 picoseconds (ps) of skew.

Other clock distributions, however, are available to the user. These clock nets are distributed through a single high-power cell driving a metal trunk along the chip. Skew within these clocks can be on the order of 300 ps, although this skew is more dependent on loading and cell distribution for each particular design.

5.0-V Compatible I/O Cells

CSALT arrays developed in Digital's fourth-generation CMOS process are powered by a 3.3-V supply for both I/O and internal core. CSALT ASICs can receive but not send 5.0-V I/O. The input receivers for both the bidirectional and the input-only cells have transistor-transistor logic (TTL) input levels and can be used in either a 3.3-V or a 5.0-V signaling environment. The CSALT PCI interface cell meets the PCI 5.0-V specification, without requiring the external module termination recommended by most ASIC vendors.

Performance-tuned Library Elements

The performance targets for the cell elements in CSALT were determined from a number of sources. First, previous ASIC designs, library performance, and heuristics were used to establish a baseline. The heuristics of the number of cell logic levels between two state elements in the DEC 7000 platform designs were analyzed. Second, the fourth-generation CMOS silicon process, electrical interconnect data, and transistor properties were used to arrive at new scaled estimates based on unit load, cell timing, and interconnect delay. Third, cycle times and system skews of the target platform were used to determine a new estimate of the levels of logic that can be placed between two state elements. The analyses resulted in the generation of baseline performance targets that were used in the design of an ASIC library tuned to cycle 100K gates at 100 MHz.

Delay Calculation

CSALT post-layout timing analysis and net delay generation are based on conservative approximations and consist of six uncorrelated, additive components:

1. Intrinsic gate delay (also referred to as intercept)
2. Effect of lumped total net capacitance on delay
3. Effect of input edge rate on delay
4. Setup/hold time
5. Effect of input edge rate on setup time
6. Wire transit delay

The first five components are derived for each standard-cell type from lookup tables created using SPICE simulation.[1] The sixth component, wire transit delay, is calculated during layout for each net in each CSALT design using a specific methodology for bounding the solution.[3]

Both worst-case and best-case analyses are performed and are guaranteed to be more conservative than SPICE, because components 1, 2, and 6 of delay are measured in a conservative fashion. Paths that fail this timing analysis are then simulated with SPICE. These paths are automatically extracted from the timing analysis result files and submitted for SPICE simulation. The results of SPICE simulation are then back-annotated into timing analyses, and the design is reanalyzed using SPICE accuracy for delays on critical path nets that had failed previously. This strategy allows us to time designs quickly with the accuracy of SPICE where needed.[1]

SPICE Library Characterization

The entire cell timing data set and cell performance tables are generated automatically through a suite of automated tools called SPICE Library Characterization (SLiC). SLiC's automated procedure will create SPICE input files to fully characterize a library of CSALT cells, execute SPICE on these files, and post-process the results into a format readable by the timing tools.[1]

For cell delay slopes and intercepts, the SLiC process produces delay tables for each input-to-output path combination through each library macrocell. This is done by simulating in SPICE with 11 discrete output capacitance values attached to the cell output. The total range of loads is broken into four windows, and a best-fit line through each window is determined. Each line is then translated so that all discrete points within the window fall on or below this line (for worst-case parameters) or above this line (for best-case parameters). This translation is one mechanism for ensuring timing conservatism. Figure 1 shows the CSALT library performance approximation.

For edge-rate effect on delay, SLiC measures output edge rates for each of the 11 capacitance values attached to each output cell described above and stores them. In addition, SLiC creates ten simulations for each input-to-output path through each library

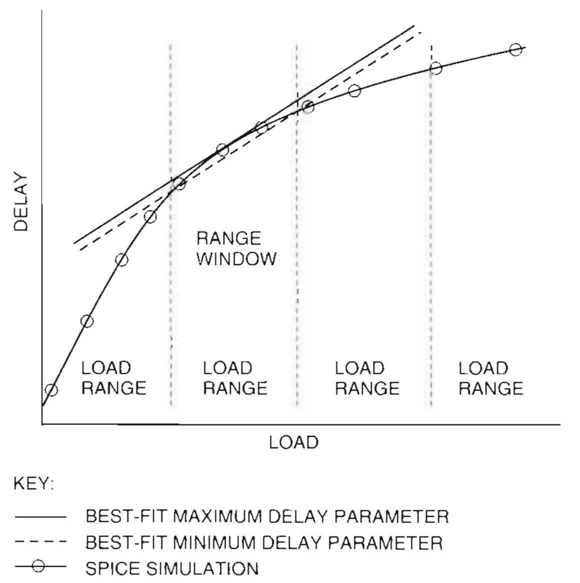


Figure 1
CSALT Library Performance Approximation

macrocell to model the range of input edge rates that the macrocell is expected to see. These two sets of data are used to create (1) a table of delay additives to gate propagation delay as a function of input edge rate and (2) a table of output edge rates as a function of gate propagation delay. These tables are then used during the timing analysis step.

The last component of delay, wire transit delay, is the only one not determined by SLiC. During layout, the bounds on the transit delay through every net are calculated. These bounds are generated very quickly and are quite accurate for short and lightly loaded nets. For longer, more heavily loaded nets, SLiC calculates more conservative bounds.[3] This conservatism contributes to inaccuracy in path timing and is the primary reason why another methodology was developed for determining more accurate delays with SPICE.[1]

This alternative methodology for calculating delay has been verified through comparisons of thousands of path delays with SPICE. In all cases, the timing was found to be conservative. Sixty-five percent of all calculated delays are within 10 percent of SPICE prediction, and virtually all delays are within 20 percent. This methodology complements the power of the fast turnaround time of static timing analysis tools by modeling the delays more accurately and closely to SPICE prediction. Large chips can be analyzed in less than one hour and be fully timed in a few hours if any SPICE simulation of large nets becomes necessary.[1]

Constraint Generation Overview

After each timing verification run, a report is generated listing all paths that fail and detailing all nets and primitives within each of these failing paths. This information is then iteratively processed through an algorithm to shorten each net in the path proportionately to its original length in the path, such that it satisfies the allowable time requirement. First, the allowable total wire delay in a path is calculated in picoseconds:

$$W = MaxTimeLimit - \sum CellPrimitiveDelays \\ - SetupTime$$

where W is the total allowable wire delay for an individual failing path, $MaxTimeLimit$ is the cycle time that the failing path needs to meet, $CellPrimitiveDelays$ is the intrinsic delay through all the primitives that exist in the failing path, and $SetupTime$ is the setup time required by the state element that ends the failing path.

Then every net in the path is apportioned according to its contribution in the current (failing) total wire delay:

$$NetNewDelay = W\left(\frac{NetFailingDelay}{ActualPathWire}\right)$$

where *NetNewDelay* is the allowable delay on a particular net in a failing path, *NetFailingDelay* is the actual delay on a net within a failing path, and *ActualPathWire* is the total accumulated wire delay of all nets in the failing path.

Since wires can be shared by more than one failing path, a change in the length of a wire in one path will cause other paths that have the same wire as an element to be scheduled for recalculation. A wire length may change several times before it is stable. During recalculation, the smaller wire produced is the one that will be used. This iteration algorithm continues until no nets are scheduled for reevaluation, and convergence is achieved. The number of iterations can be limited if convergence is not achieved in a timely manner.

At completion, *NetNewDelays* are then converted into wire lengths:

$$NetLength = \frac{NetNewDelay - (SlopeOfDriver \times \sum GateLoad)}{SlopeOfDriver \times CapUL}$$

where *NetLength* is the calculated net constraint in unit length, *SlopeOfDriver* is the slope of the cell driving the failing net in unit time per capacitance, *GateLoad* is the sum capacitance of all cells tied to that net, and *CapUL* is the capacitance per unit length for interconnect metal.

NetLength is then compared to a quench value, and the larger of the two is used as the new net constraint feeding back to a new layout. Quench values define the minimum wire length that a net can have, based on the number of pins (fan-out) in that net.

Physical Die Architecture

The CSALT die architecture, as shown in Figure 2, consists of the following sections:

- I/O cells—The outermost region where the I/O cells are located is also called the pad ring. Bonding pads are built into the I/O cells.
- High-power and decouple cells—This region of the array, also called the high-power ring, is filled predominantly with decouple cells. This region also allows for placement of a limited number of high-power driver cells designed to drive heavily loaded nets such as clock lines and reset lines.
- Core—This region holds the majority of the logic in the array implemented as standard cells. All these cells are the same height but vary in width according to functional complexity. Core cells are arranged in rows numbered from the bottom of the array. The number of rows in the core is a design-dependent variable. The space between the rows varies from row to row and is used for routing channels.

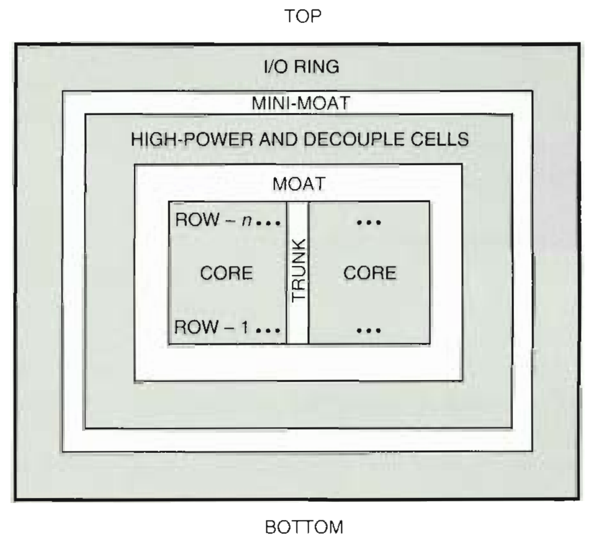


Figure 2
CSALT Die Architecture

Generally, power to the core is distributed by cell abutment on metal 3 over the cell rows. Horizontal signal routing in the core channels takes place on metal 2. Metal 1 is used for vertical core routing. To route in the vertical direction, the rows contain feeds. By design, many standard cells have vertical feeds to provide pass through. In addition, a standard feed cell can be automatically inserted by the layout tools when the demand for feeds is high. I/O bristles for each of the core cells are made available on the top and bottom of the cells to enhance routability.

- Trunk—The region splitting the core into left and right halves is referred to as the trunk. The trunk is a routing region used primarily to route clocks and power signals down the center of the core. These signals are then distributed to the left and right sides of the core on a row-by-row basis.
- Ring—Although not indicated in Figure 2, the term *ring* refers to the I/O ring, the mini-moat, and the high-power ring regions as a group. Even though the physical size of the ring is fixed, the total dimensions are determined by the package size of the array. The size of the ring establishes the available area remaining in the center of the array for the moat and the core.
- Mini-moat—The mini-moat is the region separating the I/O ring from the high-power ring. The layout process uses this region to route a small number of high fan-out nets that drive cells in the I/O ring. Layout parameters control the assignment of nets to the mini-moat.

- Moat—The moat is a routing region used by the layout tools for attaching the ring to the core. The size of the moat is determined by the amount of space that is left over when the core and trunk are placed and routed. Small arrays (low gate counts) result in small cores and large moat areas, and large arrays (high gate counts) result in large cores and small moat areas. During the layout of large chips, it is possible for the core to become so large that not enough moat space remains to make all the necessary routing connections.

Figure 3 is a photomicrograph of a CSALT die for one of the CPU gate arrays used in the AlphaServer 8000 series server systems.

Placement and Routing

The function of the layout tool suite is to provide a fully placed and routed array that meets or exceeds all the design timing criteria and that satisfies all electrical and physical layout rules required to release the array for mask generation. Several place-and-route features are discussed in the following sections.

Constraint-driven Layout System

When an array is submitted to layout, it is accompanied by a set of timing constraints. Timing constraints can be thought of as estimated restrictions, on a per-net basis, for the amount of metal lengths allowed to interconnect the net in the layout. These constraints drive the TimberWolf placement tool and are ultimately responsible for the placement of core cells in the final layout. Because a working design may not be achieved on the first layout iteration, the overall CSALT methodology provides mechanisms for analyzing post-layout timing delays and for generating a refined set of constraints that can be fed back into the layout for another pass. The layout process is iterated in this manner until it converges on a layout solution that meets the timing constraints.

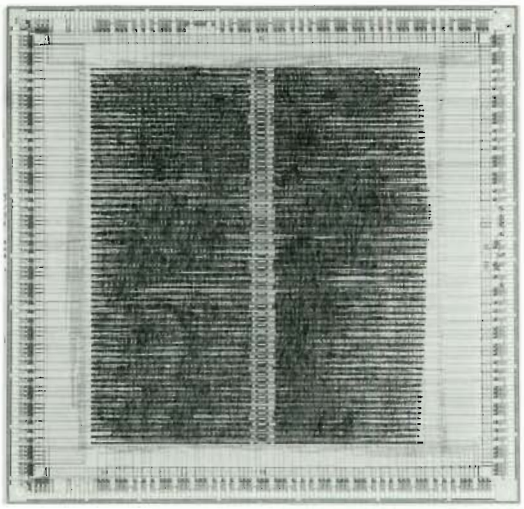


Figure 3
Photomicrograph of a CSALT Die

Routability

To ensure 100 percent routing, the routing process had to be kept simple, which required substantial planning during development of the chip architecture described above. As a result, the following elements of the architecture were defined: (1) Pins are available on both the top and the bottom of the cells; (2) Power and clock connections are defined by cell abutment; and (3) Total routing of the chip is divided into four areas (the core, the moat, the ring, and the megacell interface). This plan kept the routing problems similar from chip to chip, which allowed the routing tools to focus on particular solutions.

Quick Turnaround Time

One significant feature of the CSALT layout process is that it can complete a layout without manual intervention, saving time over manual processes. CSALT consistently demonstrated that the CAD suite can provide completed layouts in three to ten days from the time the wirelist enters the layout process. An array that has been in layout for ten days is likely to be one that is difficult to time and that has required four to six layout iterations to converge.

Cross-talk Effect Inclusion

In recent generations of ASIC technologies, interconnect metal widths and pitches have been shrinking while the clock frequencies have been on the rise. This raised some concerns about on-chip cross-talk effect due to the ability of signals traveling on one wire to affect the speed of signals traveling on adjacent or victim wires. In extreme cases, this cross talk can cause signals to spike on the victim wires. CSALT methodology compensates for such effects on wire delay calculation, and the compensation is integrated into the layout process.

The integration is implemented by factoring in a coupling capacitance extracted from layout and by using a worst-case signal-switching scenario. Conservative factors were chosen after analysis of cross talk on a representative cross section of CSALT layouts, using different routing pitches on signal interconnect metal 2. The goal was to find the right balance between metal pitch, area used, and chip timing. The study resulted in an optimum pitch definition of 3.75

micrometers (μm) for metal 2, and a coupling capacitance multiplier of 2. The core area increase, from that of using the minimum pitch for metal 2 at 2.625 μm, was less than 10 percent for the largest design. However, overall die area increase was negligible due to the designs being I/O-ring-limited in nature.

Free of Electromigration

When the current density in the aluminum interconnect used in today's high-density CMOS processes is too high, a detrimental physical phenomenon occurs. This phenomenon causes metal reliability problems in which metal molecules slowly migrate, resulting in open/short circuits inside ASICs. To eliminate the need for a long and manually tedious process of looking for these problems after layout, the CSALT strategy is to avoid electromigration problems during layout through analysis and implementation of built-in conservative layout disciplines.

Analysis of several CSALT arrays resulted in an increase in the contact capacity and the definition of maximum output load limits for each macrocell. The limit set was a maximum of 130 unit loads (7.8 picofarads [pF]) switching at maximum frequency (100 MHz). In addition to that limit being available to designers during the design phase for proper fan-out implementation, the tools automatically flag all nets that exceed the limit.

A number of other features are designed into the CSALT process to guarantee that layouts are free from electromigration problems:

- Library data tables are used to dynamically assign metal widths and corresponding contact sizes according to driver strengths and loads. This eliminates electromigration problems for dynamically sized metal routes such as clock nets and other high fan-out nets.
- The bulk of the power distribution is achieved by cell abutment. Cell power rails are conservatively designed to handle the largest row's current demand.
- As a final check on correctness, one of the layout process steps incorporates a hot row tool. This tool flags any rows in the core whose cells collectively exceed a predetermined current threshold defined by the handling capability of the power cells in the rows. This information is used to flag a potential electromigration situation in the contact structure, distributing power from the trunk to the row. When the row is flagged, the user manually reviews the result files and analyzes the row. Out of 15 separate designs completed, not one had to be changed due to flagged hot rows. This is due to the extremely conservative assumption used by the tool—it assumes all logic is switching at maximum frequency.

Correct-by-construction Concept

As it applies to all the critical device issues (for example, electromigration, cross talk, hot carrier injection, and latchup), acceptance of the concept of a layout being correct by construction has dramatically reduced turnaround time in the layout process by eliminating the need to perform these analysis operations on each array. Why does it work for CSALT? It works because the CSALT layout process is very deterministic, and correctness has been verified on a cross section of arrays. In the final analysis, all arrays use the same cells from a well-defined and characterized library. The architecture of the die is the same in all arrays. As a result, variation is likely to enter the system only during the routing process. This process incorporates conservative layout rules and checks to avoid and detect potential failure mechanisms.

CSALT Layout Process

As shown in Figure 4, the layout process encompasses five basic assembly and check operations: full wirelist preparation, pad-ring assembly, core assembly, chip assembly, and verification.

1. During wirelist preparation, the input wirelist is analyzed, names are manipulated to conform to layout naming conventions, and the design is partitioned into pad-ring and core components.
2. During pad-ring assembly, I/O and high-power cell/slot assignments are made according to bonding requirements. The ring is then globally routed.
3. During core assembly, floorplanning for the trunk and any random-access memory (RAM) devices takes place; timing constraints from several sources (pre-layout, user defined, current layout, and previous layouts) are merged into a worst-case set of composite constraints that are used by the TimberWolf tool to place and globally route the core. Also during this step, the balanced clock system and scan chain are synthesized and globally routed. The SCAR channel router is then used to route the standard-cell portion of the core. If the design contains RAMs, they are then placed in their floorplanned locations, globally routed, and finally attached to the core using the area router, Chameleon.
4. During chip assembly, the interfaces between the core, moat, and pad ring are refined. Chameleon is used to perform final routing of the ring and the moat. Thus far the chip has been completely placed and routed using cell outlines containing only

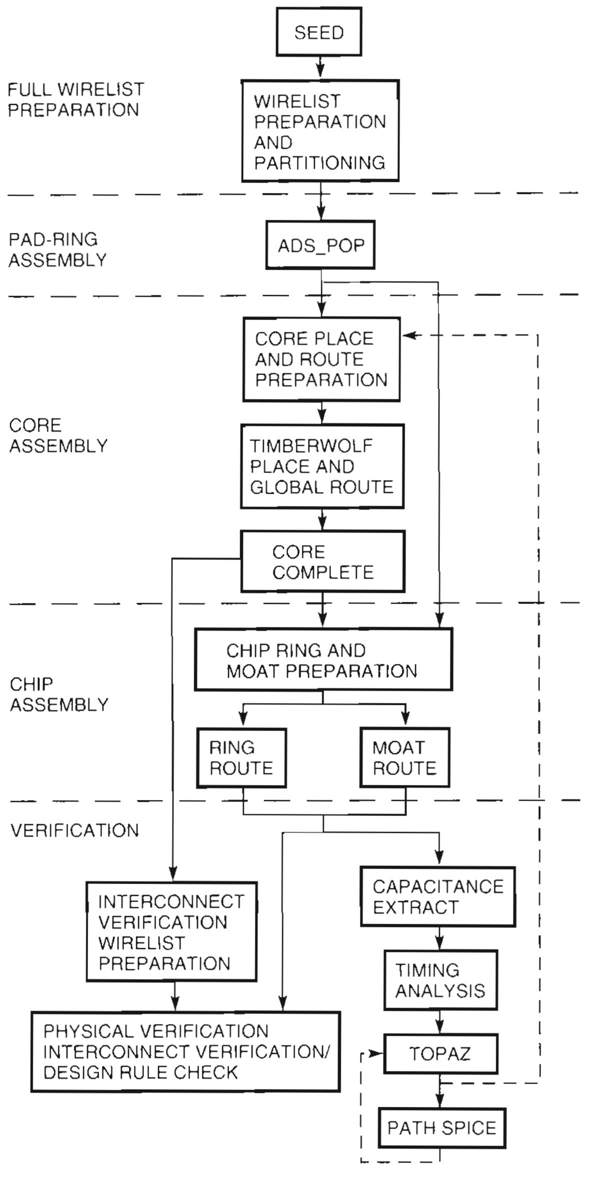

SEED
FULL WIRELIST PREPARATION
WIRELIST PREPARATION AND PARTITIONING
PAD-RING ASSEMBLY
ADS_POP
CORE PLACE AND ROUTE PREPARATION
CORE ASSEMBLY
TIMBERWOLF PLACE AND GLOBAL ROUTE
CORE COMPLETE
CHIP RING AND MOAT PREPARATION
CHIP ASSEMBLY
RING ROUTE
MOAT ROUTE
VERIFICATION
CAPACITANCE EXTRACT
INTERCONNECT VERIFICATION WIRELIST PREPARATION
TIMING ANALYSIS
PHYSICAL VERIFICATION INTERCONNECT VERIFICATION/ DESIGN RULE CHECK
TOPAZ
PATH SPICE

Figure 4
CSALT CAD Layout Overview

dimension information, relative coordinates of bristles, and bristle names. The ring and core cell outlines are now replaced with the actual cell layout information from the library, and a complete design layout file for the chip is produced. As the file is generated, alignment block and substrate ring data is added to make the physical representation file ready for mask set production.

5. The following procedures occur during the verification phase of the layout:
   - Physical design rules are checked.
   - A wirelist comparison is performed to ensure that the layout file is an exact match to the logical representation of the design.
   - Capacitance information from the layout file is extracted, and the process proceeds to calculate a conservative metal delay (including compensation for cross talk) for each net in the layout. These post-layout metal delays are fed back into the timing verification process. In addition, SPICE is run on clock nets and other critical nets predefined by the user, and the delays are made available for timing analyses.[1]

The entire process runs automatically and, when possible, steps are run concurrently. Manual intervention is unnecessary and is discouraged; it is used only during tool debugging or special customization.

Process Controller

CSALT's fully automated process controller (PC) ensures optimum use of system resources, orchestrates the entire layout process, provides all necessary data management functions, and provides the user with a very simple set of commands for operating an otherwise complex process.

The power of the PC is in its continuous dynamic decomposition of every layout into parallel batch streams. The PC runs the entire layout process in batch mode, taking full advantage of opportunities to use multiple processors and run independent parts of the layout process in parallel streams. Because it hides all the CAD and process complexity from a user, no previous CAD or layout skills are required to iterate layouts once initial layout parameters have been established for a given array.

Figure 5 illustrates the flow for the CSALT layout process. (A and B indicate connectivity points.) Each name in the process flow represents the name of a single process step. Execution of the layout process is controlled by a single command procedure called CSALT place and route (CPR). Although other CPR command line options exist, CPR is most often used in its simplest and most powerful form, CPR PC. This command causes CPR to invoke the dynamic PC that will analyze the current relationships of a layout database and begin automatic execution of the layout process from the next eligible process step.

Results and Conclusion

CSALT gate array technology was used extensively during the development of the AlphaServer 8000 server systems. This design methodology removed the product's critical dependency on the place-and-route portion of the design process. As a result, timing-correct ASIC layouts were produced in fewer than 72 hours. In addition, the CSALT ASIC logic designers had access to the proven 3.3-V silicon structures

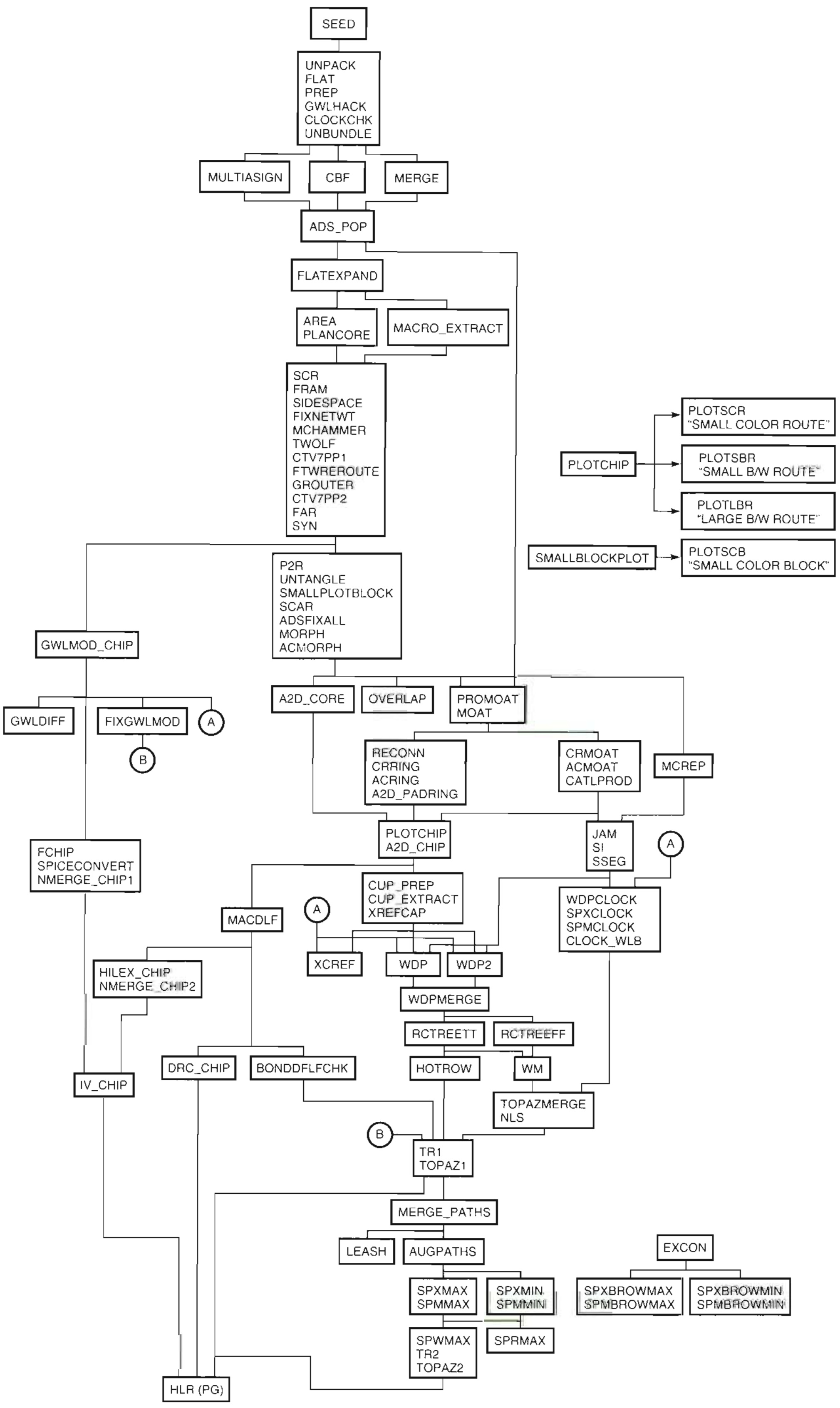


Figure 5
CSALT CAD Process Flow Diagram

developed by Digital Semiconductor. CSALT's timing-driven layout approach for designing and implementing a high-performance ASIC made the AlphaServer 8000 server systems' aggressive 10-nanosecond bus speed a reality, with minimum risk.[4]

Although CSALT's implementation pieces may not be unique, the approach that was taken to link the set of front-end design tools with the back-end layout has proven to be unique, with unmatched results. No ASIC vendor today (January 1995) can provide logic designers with the ability to do their own automated, timing-correct layouts from their desktops.

In less aggressive designs, a large number of working layout solutions exist. The number of these solutions starts to shrink when the technology is more aggressively used. Iterative timing-driven layout efficiently searches through the matrix of possible solutions to find a working layout. Coupling timing-driven layout with logic synthesis can bring us very close to achieving the "silicon compiler" goals of automatically producing working designs from high-level logic descriptions.

Acknowledgments

The authors would like to acknowledge the efforts of the following people, without whom the project would not have been successful: Meaghan Engdahl, Dave Caffo, and Paul Janson. We would also like to acknowledge other members of the CSALT team: Linda Greska, Kevin Gamache, Dennis Litwinetz, Bill Gist, Dave Vanderbeek, John Drasher, John Kennedy, and certainly Dick Davis and Darrel Donaldson for their CSALT vision. We would also like to acknowledge Professor Carl Sechen and Bill Swartz for their continued TimberWolf support.

References and Notes

1. SPICE is a general-purpose circuit simulator program developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

2. TimberWolf Mixed Macro/Standard Cell Floorplanning and Routing Package is a public domain automatic layout package. Developed and directed by Professor Carl Sechen from July 1986 through June 1992 at Yale University and later at the University of Washington, it is currently maintained and supported by TimberWolf Systems, Inc., Dallas, Texas.

3. J. Rubenstein, P. Penfield, and M. Horowitz, "Signal Delay in RC Tree Networks," *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems,* CAD-2 (3) (July 1983): 202–211.

4. D. Fenwick, D. Foley, W. Gist, S. VanDoren, and D. Wissell, "The AlphaServer 8000 Series: High-end Server Platform Development," *Digital Technical Journal,* vol. 7, no. 1 (1995, this issue): 43–65.

Biographies

Jean H. Basmaji
Jean Basmaji is a hardware consulting engineer in the Server Platform Development Group. As the CSALT development project leader and ASIC technology manager, he was responsible for the transition of the CSALT technology project from advanced development to production. Jean has been the technical director of computer-aided engineering and design verification testing for the Server Platform Development Group since the introduction of Digital's first multiprocessing computer, the VAX 6200 system. Jean joined Digital in 1978 after receiving a B.S.E.E. from Lowell Technological Institute in 1977.

Kay R. Fisher
Kay Fisher joined Digital in 1973 as a hardware instructor and course developer. He has more than 20 years of software engineering experience in the areas of memory testing, fault-tolerant multicomputer systems, and VAX systems. Currently a principal software engineer in the Alpha and VAX Servers Group, Kay is responsible for the process control software for Digital's CAD system. His code runs programs in parallel and synthesizes clock drivers, networks, and scan chains. Kay received a B.S. in computer science (magna cum laude, 1978) from Boston University–Metropolitan College.

Frank W. Gatulis
As supervisor of the CAD segment of the CSALT technology team, consultant engineer Frank Gatulis is responsible for defining and implementing automated layout tools and processes. A graduate of Don Bosco Technical Institute, he also attended the B.S.E.E. program at Northeastern University. Frank has also supervised the I/O diagnostic

team for DECsystem 10 and DECsystem 20 products as well as the development of the console-based fault detection system and automated isolation tool suite used on the VAX 8600 and VAX 8650. His architecture segments strategies were used to fabricate and test multichip units in VAX 9000 systems. Before joining Digital in 1973, Frank worked at EG&G where he developed real-time picture acquisition, processing, and compression software.

Herbert R. Kolk
A consulting software engineer who specializes in chip routing, Herb Kolk currently supports and enhances the CSALT layout process. Prior to that, he was the architect of the Chameleon router, which was used for routing gate arrays, multichip modules, and CPU boards for the VAX 9000. This router is still being used in the CSALT process. Before joining Digital in 1983, Herb designed software systems for the Communications Systems Division of GTE. Herb received a B.S.E.E. (with honors) from the Rochester Institute of Technology in 1973.

James F. Rosencrans
Jim Rosencrans is a principal hardware engineer in the AlphaServer Engineering Group. As the ASIC technologist for server system development, Jim supports the AlphaServer 8000 series design teams and acts as liaison for both the teams and test engineering. In addition to leading the CSALT micropackaging definition and design, he contributed to the development of CSALT technology. His previous work includes contributions to ASIC and ASIC/silicon technology selection, definition, and design support. Before joining Digital in 1988, Jim was with NCR Microelectronics, where he worked on custom and CMOS ASIC design and silicon process development. A member of IEEE, Jim received a B.S.E.E. from the University of Wisconsin in 1980.

Nitin D. Godiwala
Barry A. Maskas

The Second-generation Processor Module for AlphaServer 2100 Systems

The second-generation KN470 processor module for AlphaServer 2100 systems performs significantly better than the first-generation KN460 module and was designed to be swap-compatible as an upgrade. The KN470 processor module derives its performance improvements from the enhanced architecture of Digital's new Alpha 21164 microprocessor, the synchronous design of the third-level cache and system interface, the implementation of a duplicate tag of the third-level cache, and the implementation of a write-invalidate cache coherence protocol for the multiprocessor system bus. Additional design features such as read-miss pipelining, system bus grant parking, hidden coherence transactions to the duplicate tag, and Alpha 21164 microprocessor write transactions to the system bus back-off and replay were combined to produce a higher performance processor module. The scope of the project required implementing functionality in system components such as the memory, the backplane, the system bus arbiter, and the I/O bridge, which shipped one year ahead of the KN470 module.

The second-generation KN470 processor module for AlphaServer 2100 systems achieves a higher performance than the first-generation KN460 module while maintaining compatibility with the AlphaServer 2100 system environment. This paper describes the processor module project and the resulting design. Topics discussed are the elements that contribute to the compatibility and to the higher performance: coherence protocol, system bus protocol, system bus arbitration, system interface and shared data, and clocking. Some key design trade-offs are described. The paper concludes with a performance summary that presents measured attributes of the higher performance KN470 processor in the context of the AlphaServer 2100 family.

When the AlphaServer 2100 product family was being defined in late 1992, the processor module performance-over-time roadmap projected three performance variations based on increasing the clock rate of the Alpha 21064 microprocessor.[1] These modules were to be compatible with Digital's mid-range multiprocessor system bus and would support enhanced functionality such as direct-mapped I/O, up to four microprocessors, an I/O bridge to 32-bit Peripheral Component Interconnect (PCI) and Extended Industry Standard Architecture (EISA) buses, and an I/O expansion option module with an I/O bridge to a 64-bit PCI bus.[2] Two members of the DEC 4000 processor design team were assigned to deliver this first-generation processor module. At this time, there was no goal to develop a second-generation processor module. Therefore, the remainder of the team designed the arbiter chip and the enhancements required in the processor-module system interface chips and at the same time contributed to the Alpha 21164 microprocessor development effort.

Goals for contributions to the Alpha 21164 microprocessor development effort were partitioned into short- and longer-term goals. A short-term goal was to define a system for the new Alpha microprocessor.[3] The related longer-term goal was to ensure that the Alpha 21164 microprocessor could operate in that defined system. An architectural study resulted in a proposal and a project plan to develop a second-generation processor module that extended the performance and longevity of the AlphaServer 2100 family. In addition, the remainder of the team made requests of the Alpha 21164 microprocessor team to incorporate specific legacy-related AlphaServer 2100 functions such as support for 32-byte cache blocks, control of I/O address space read merging, and completion of memory barriers on the Alpha 21164 microprocessor. The business management team accepted the proposal and the project plan. The Alpha 21164 microprocessor team agreed to support the functionality requests. The design team staffing was completed by March 1993, and detailed design work began in May 1993. The design team's goal was to have a processor module ready to accept the Alpha 21164 microprocessor for installation when the microprocessor first became available. The team met this goal.

Since the first- and second-generation processor modules would operate in the same enclosure and with the same power supply, the size and shape (i.e., form factor), cooling demands, and power consumption of the new module had to be compatible with those of the first-generation module. Because of the presence of an on-chip, write-back second-level cache and an estimated longer access time to that cache from the system bus, the Alpha 21164 microprocessor architecture adopted an invalidate-on-write cache coherence protocol. The Alpha 21064 microprocessor supported an off-chip, write-back second-level cache that has a faster access time from the system bus. This faster access time enabled the implementation of a good-performing update-on-write cache coherence protocol. Support of these snooping, multiprocessor system bus coherence protocols required enhancements to the system bus transaction types. This resulted in minor logic changes to the memory interface chips and to the I/O bridge chip.[4,5] These changes were defined and implemented in time for the first-generation system power-on. Hence, the system components, the I/O bridge chip, the memory modules, and the system bus and backplane are compatible with the first- and second-generation processors. This basic difference in the system bus coherence protocols prevented the system from supporting the coexistence of the first- and second-generation processor modules because such a configuration has asymmetric attributes. Alpha operating system software does not support asymmetric multiprocessing; symmetry is assumed.

Another project goal was to maintain the AlphaServer 2100 family's position among the industry's leading high-performance server systems. This goal was achieved by exploiting the Alpha 21164 microprocessor's performance through the design of the processor module's third-level cache, by implementing a full-duplicate tag of this cache, and by implementing a synchronous clocking scheme. Combining the processor design attributes with a pipelined read transaction of a faster read-access system memory module enabled the team to achieve the project's goal of designing a higher performance processor module and multiprocessor system.

Overview of the Processor Module

The KN470 processor module provides an operational environment for the Alpha 21164 microprocessor. This environment, which is similar to that of the first-generation KN460 processor module environment, includes the following:

- Alpha 21164 microprocessor—a superscalar, superpipelined implementation of the Alpha architecture with low average cycles per instruction because of its four-instruction issue
- B-cache—a module or third-level write-back cache
- System interface—two application-specific integrated circuit (ASIC) chips that interface the Alpha 21164 microprocessor, B-cache, and duplicate tag store to the system bus
- Duplicate tag store—a tag store of the third-level write-back cache
- System bus clock repeater that provides system bus synchronous clocks to the module
- System bus arbiter that determines which system bus node can access the system bus
- Serial control bus subsystem that includes clock and reset control circuitry, a microcontroller with a serial interface, serial read-only memory with power-on firmware bits, and nonvolatile memory for processor configuration parameters

Figure 1 shows a block diagram of the KN470 processor module.

The Alpha 21164 microprocessor is organized with an on-chip 8-kilobyte (KB) primary instruction cache and an 8-KB write-through data cache, which are referred to as first-level caches. In addition, a 96-KB, second-level, three-way, set-associative write-back cache is implemented on the chip.

The module design includes a B-cache or third-level cache to minimize the miss penalty and to be configurable through the use of various densities of similarly packaged static random-access memory (RAM) chips. Such a design enabled final product definition late in the verification process based on static RAM costs and delivered performance from the B-cache. The size of the B-cache is either 1, 2, 4, 8, or 16 megabytes (MB). Each B-cache entry stores 32 bytes of data and the associated tag bits and is called a cache block. To facilitate read-fill data and victim-write data exchange with the system interface, the Alpha 21164 microprocessor and the system interface share the B-cache data port. The B-cache is controlled by the Alpha 21164 microprocessor, which has its second-level cache configured

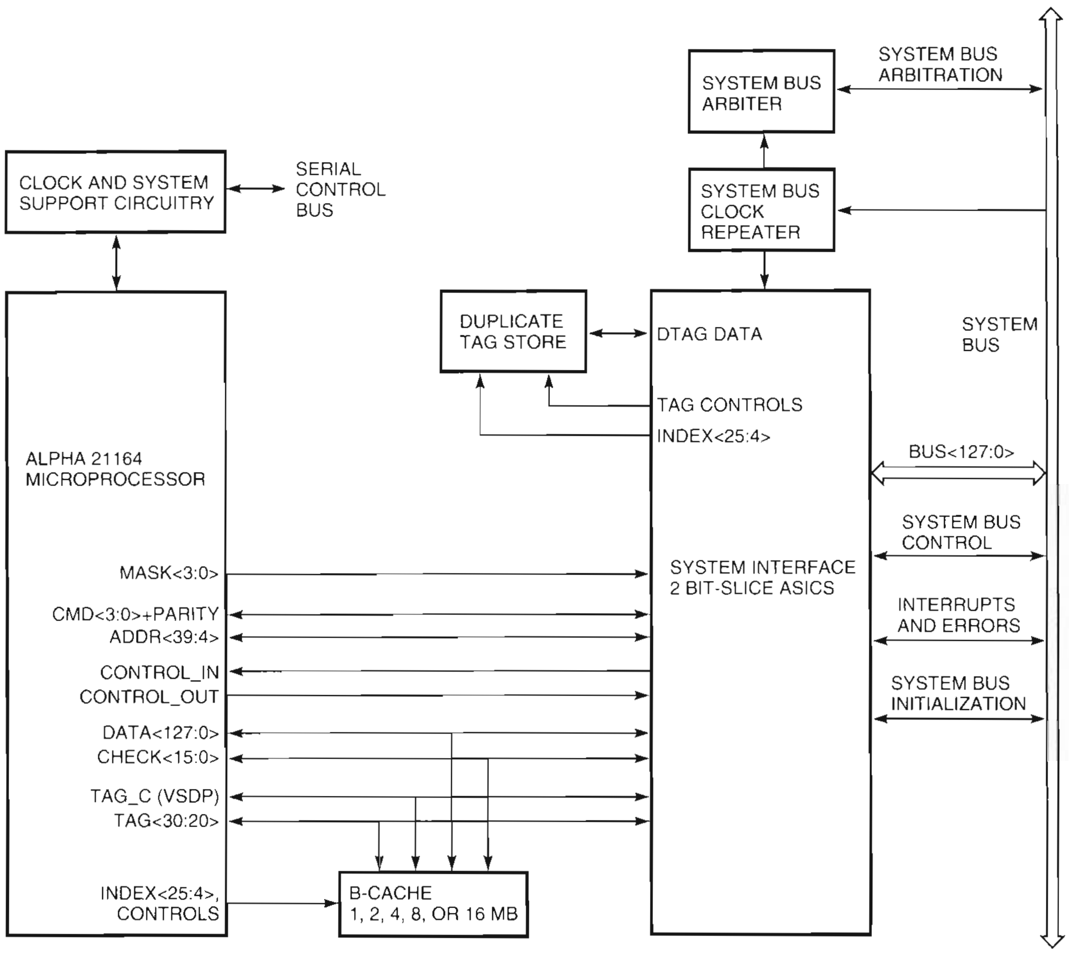


Figure 1
Block Diagram of the KN470 Processor Module

to operate in 32-byte instead of 64-byte cache block mode. This 32-byte mode of operation for the second-level cache was the most complex request made of the Alpha 21164 microprocessor design team. However, this design element was required to achieve the AlphaServer 2100 compatibility goal.

The system interface is a common boundary between the system bus, the Alpha 21164 microprocessor, and the third-level cache. The system interface provides the protocol and circuitry for the Alpha 21164 microprocessor to read or write devices connected to the system bus. Conversely, the system interface provides the protocol and circuitry for the processor module to respond to read or write transactions from the system bus. The system interface comprises two identical bit slices of an ASIC. The ASICs operate as even and odd slices, based on a mode-select pin on the module. The system interface selects the operating mode of the arbiter chip. It also encodes the system bus transaction type as read or write and then supplies a control signal to the arbiter chip. The arbiter chip must know the present system bus transaction type to remain synchronized with the system bus events and to know when to sample new requests for the system bus.

The module maintains a duplicate copy of tag control bits of each B-cache block in the duplicate tag store. The duplicate tag store is controlled by the system interface and is time multiplexed between system bus requests and Alpha 21164 microprocessor requests. This ability to pipeline transactions to the duplicate tag store from the Alpha 21164 microprocessor and the system bus allowed the Alpha 21164 microprocessor's requests to fill predictable time slots in parallel to the system bus transactions, hidden from the system bus. This is called cycle-stealing because the coherence transactions requested by the Alpha 21164 microprocessor do not require arbitration for or use of the system bus cycles. Cycle-stealing provided more useful system bus bandwidth while at the same time reduced the Alpha 21164 microprocessor latency for coherence transactions to the duplicate tag store.

The clock repeater chips generate complementary metal-oxide semiconductor (CMOS)–level clocks from positive emitter-coupled logic (PECL)–driven backplane clocks. These CMOS-level clocks are skew regulated and distributed to the module's components. The Alpha 21164 microprocessor has digital-lock-loop circuitry, which aligns the Alpha 21164 microprocessor's interface clock to the reference clocks that run to all other module components. This scheme is basically the synchronous clocking scheme.

The module includes the system bus arbiter chip. The decision to locate this chip on the processor instead of the backplane stemmed from concerns over compatibility between the first- and second-generation processor arbitration algorithms supported by this chip. The system bus arbiter chip was designed and fabricated for the first-generation processor and included the functionality of the second-generation processor. The chip design was completed prior to the design of the KN470 processor module's system interface. To minimize the chance of design error, the team performed extensive simulations to help the project realize a full-function, second-pass chip for use in the first- and second-generation processor modules.

The KN470 processor module implements the system bus reset control and serial control bus subsystems, with minor modifications, that were designed for the first-generation processor module.

Cache Coherence Protocol

To improve the in-system performance of the Alpha 21164 microprocessor and its write-invalidate cache coherence protocol, the KN470 module implements a duplicate tag store of the B-cache. The Alpha 21164 microprocessor has two levels of on-chip cache that are maintained as a subset of the B-cache. This discussion assumes that the first- and second-level caches remain coherent with the B-cache and duplicate tag store. Operations performed by the Alpha 21164 microprocessor and system interface maintain the B-cache and duplicate tag store subset rule for the on-chip caches. The duplicate tag store and the B-cache each keep three control bits to maintain coherence with the on-chip caches and also with system memory and other module caches. The three control bits are called valid (V), shared (S), and dirty (D). A combination of control bits makes up a state of a cache block. The five possible cache block states are as follows:

1. VSD = 000 A cache block is either empty or removed from the B-cache and hence invalid.
2. VSD = 100 A cache block is the only cached copy in the system.
3. VSD = 101 A cache block is valid, and this copy has been modified more recently than the copy in memory.
4. VSD = 110 A valid cache block may also be in another cache. This processor must write or broadcast write modifications of this block to the system bus.
5. VSD = 111 A valid cache block may also be in another cache, and the copy of this block has been modified more recently than the copy in memory.

Cache state transitions are synchronized to system bus transaction cycles because the system bus is the common point of coherence and coherence conflict resolution.

When the Alpha 21164 microprocessor requests a transaction for a cache block to be read or filled from system memory into the B-cache and on-chip caches, the cache block state is set to VSD = 100 in the duplicate tag store, the B-cache, and the on-chip caches. The first-level instruction and write-through data caches must maintain only a valid bit. The second-level write-back cache must maintain the VSD bits consistent with the B-cache and the duplicate tag store.

If an Alpha 21164 microprocessor's read transaction request is of the type intent-to-modify, then the cache block state makes a direct transition to VSD = 101. A block in the valid state of VSD = 100 will make a transition to VSD = 101 when an Alpha 21164 microprocessor's request to modify the cache state has reached the point of coherence, i.e., the system bus. However, the duplicate tag store is maintaining coherence with the system bus, so this request must find a nonconflicting cycle to effect the state transition. If another processor reads the same block before this processor's request has reached the point of coherence, then the cache block state makes a transition to VSD = 110. In this case, the system interface updates the shared state to VSD = 110 for the read with intent-to-modify transaction before the block is modified. Because the block is shared, this processor's request to modify the block must now also become a broadcast write transaction to the system bus. Once the modified block is written to the system bus, the next state transition is to VSD = 100. The broadcast write transaction is sometimes referred to as an unsharing transaction.

Once a cache block state is valid, i.e., VSD = 100, it can be invalidated or set to the state VSD = 000 from the system bus by another processor's read with intent-to-modify or by a write transaction to that block. The Alpha 21164 microprocessor does not allow an update of write data from the system bus but instead invalidates the block. Invalidation requires the duplicate tag store, the B-cache, and the on-chip caches to clear their V state. Implementation of system bus write transactions that cause block invalidation is required to support the cache coherence protocol of the Alpha 21164 microprocessor.

By filtering out system bus transactions that do not alter the coherence states of the Alpha 21164 microprocessor, the duplicate tag store serves to minimize the frequency with which the system bus transactions interrupt the microprocessor operations. Without the duplicate tag store filtering, the Alpha 21164 microprocessor would have to be interrupted on every system bus transaction, thus limiting the system performance.

System Bus Protocol

The KN470 processor module incorporates both an enhanced system bus protocol and a system bus arbiter that minimizes the arbitration latency.

Enhancement of Transactions

For the first-generation AlphaServer 2100 processor, the system bus protocol is the same as the one implemented in the DEC 4000 system. This system bus protocol is a snooping bus protocol in which all bus participants are required to monitor system bus transactions and to keep their cached copy of memory coherent. For the second-generation processor, the designers enhanced the DEC 4000 system bus protocol to support the write-invalidate cache coherence protocol of the Alpha 21164 microprocessor.

The DEC 4000 system bus protocol supports four types of transactions: read, write, exchange, and no operation. The exchange transaction performs a victim-write transaction to one memory location and a read transaction of another memory location. The two transactions are separated by the common lower 18 bits of address. The exchange transaction combines read transactions and victim-write transactions into one transaction, sharing the address cycle of the system bus. The exchange transaction is used to evict modified cache blocks from the caches back to system memory to allow a replacement block with a different tag to be allocated.

To support the second-generation processor's write-invalidate coherence protocol, transactions were added to the first-generation system bus protocol. These added transactions were needed to signal other processors and the I/O bridge chip to invalidate a block when a block was being read for the purpose of being modified. The exclusive-read and exclusive-exchange transactions were added to the four first-generation transaction types. The exclusive-read transaction is the read transaction that also causes cache invalidation by a bystander processor module and the I/O bridge chip of the block being read. The exclusive-exchange transaction is the exchange transaction that also causes cache invalidation by a bystander processor module and the I/O bridge chip of the block being read.

The KN470 module implemented the exclusive transaction types to establish private ownership of a block. Establishing private ownership to a previously shared block enables write transactions to complete without having to broadcast write transactions back to the system bus. This occurs because the block is invalidated by bystanders who were sharing the block.

The enhancements of the system bus transaction types did not affect the memory module. The implementation of the exclusive indication signal was such that memory would decode a read or exchange transaction and not know of the exclusive signaling. Because the I/O bridge chip caches translation addresses for direct memory access of devices on the PCI or EISA buses, minor modifications were designed into the I/O bridge chip to support these enhanced commands.

Minimization of Arbitration Latency

The system bus arbiter implemented a bus grant parking or pregrant signaling scheme that minimized the arbitration timing overhead. This scheme combined with the pipelining of the read-miss commands from the Alpha 21164 microprocessor enabled the system bus interface to use the available memory bandwidth.

The arbiter for the first-generation processor followed the protocol used in the DEC 4000 system. The arbiter samples the requests and then issues the grants according to round-robin arbitration rules. The arbitration rules allow processor modules to have fair access to the system bus. The elapsed time from when a processor makes a system bus request to the arbiter until it receives a grant is referred to as the arbitration cycle or arbitration overhead. The arbitration overhead increases the memory and direct-mapped I/O access latency, as well as the cache-miss penalty. Typically, the arbitration overhead for each processor appears low in a multiprocessor system in which bus utilization is extremely high. The appearance of low arbitration overhead results from the time the system bus waits to finish a transaction before the arbiter can issue the next grant. However, the arbitration overhead may be as high as 20 percent of the transaction time in a system configuration in which one processor module is consuming the available grants from the arbiter.

The arbiter used by the second-generation processor pregrants or parks a grant to the processor module whenever the system bus goes idle. This feature eliminates arbitration overhead. The result is a lower miss penalty and an ability to sustain a continuous stream of read transactions when the bus is not utilized by other system bus nodes. This arbiter enhancement does not cost additional arbitration overhead for other requests because the cost of unparking a grant was eliminated through the signaling protocol. This signaling protocol enabled the pregranted signal to be negated at the same time a new grant signal is asserted.

The Alpha 21164 microprocessor is capable of pending read-miss requests to the system interface. These read transaction requests sometimes have an associated victim that must be displaced by the requested read data. By pipelining these requests in relation to the system bus grants, a continuous stream of back-to-back system bus read or exchange transaction requests can flow because of the parked grant. Since the Alpha 21164 microprocessor is capable of continued execution while miss requests are pended, the processor designers had to carefully schedule the use of the B-cache. The fill data coming from system memory and the Alpha 21164 microprocessor are in contention for use of the B-cache. The system interface minimizes the time that the B-cache is allocated to accept the fill data while maintaining the flow of commands into the read miss transaction pipeline from the Alpha 21164 microprocessor. By allowing the microprocessor to have access to the B-cache before and after each fill, a continuous flow of transactions was realized. The continuous flow of transactions uses the available system bus bandwidth.

Handling of Shared Data

A shared-database environment in which write transactions are prominent uses the system bus exclusive transaction types to establish ownership of the cache blocks. These transaction types minimize the system bus bandwidth usage by avoiding write broadcast transactions of modified blocks.

In a multiprocessor environment, a block that is valid in more than one cache is called a shared block. The coherence state of a shared block is VSD = 110. The following example summarizes the problem associated with a write transaction to a shared cache block in a system bus protocol without the exclusive transaction types.

Processor A has a modified but unshared cache block with state VSD = 101. Processor B wants to write the cache block that Processor A has modified. Processor B issues a read transaction on the system bus and then must immediately follow the read transaction with a write broadcast transaction of the modified data. The write broadcast transaction must be issued by Processor B because the read transaction was shared. At the end of the two bus transactions that it issued, Processor B's cache block state will be VSD = 101. Processor A has invalidated its cache block. Thus, two bus transactions were required from Processor B to write the modified cache block. With fair arbitration, however, Processor B may not have access to the system bus after the read transaction. The write transaction may be blocked, thus creating other coherence situations. If two or more processors in a system are trying to write the same block, Processor B may not get access to the system bus to complete the write transaction. The system is potentially in deadlock.

The system bus protocol implemented by the KN470 enables the write transaction to complete but requires only one system bus exclusive-read transaction. In response to the Alpha 21164 microprocessor's request to modify a cache block, the processor initiates an exclusive-read transaction on the system bus. Other processor modules responding to this exclusive-read transaction provide the data if their block is dirty, but regardless of the dirty state, they also invalidate their cache block. The invalidation eliminates the shared state. If no other processor module has a dirty block, the data is returned from the system memory. The processor module that is issuing an exclusive-read transaction sets its cache block state to VSD = 101 as it fills. The write transaction that is pending in the processor can complete without broadcasting a write transaction to the system bus.

A system bus that does not support the exclusive transaction types requires a shared write transaction to a block to be decomposed into two system bus transactions. This can result in system bus bandwidth saturation. A system bus that supports the exclusive transaction types requires only one system bus transaction. In a shared-data environment in which write transactions to shared data are the prominent cause of cache misses, support for the exclusive transaction types helps preserve bus bandwidth. Also, the deadlock scenario presented above does not exist. The KN470 processor write transactions to a cached block consume only one system bus transaction and can always complete. The invalidate window does not exist during the time it takes for the write transaction to complete.

The system bus protocol implemented by the KN470 module allows forward progress during shared write transactions in the system. However, system software is expected to avoid repetitive write transactions to blocks that are shared without some higher level ownership protocol. Write transactions, if issued to a shared block by several processors, consume bus bandwidth and trigger false invalidations for bystanders. This may hinder forward progress and affect system performance.

Support of an Interlock Mechanism

The system interface implements an address lock register as specified in the *Alpha Architecture Reference Manual* to support software synchronization operations.[6] The address lock register in the system interface has a signal that reflects the state of a valid bit to the Alpha 21164 microprocessor. The microprocessor manages the lock address register in the system interface based on sampling this signal during fill transactions from the system bus.

The Alpha 21164 microprocessor has an internal lock register that is maintained consistent with the lock register in the system interface, which is referred to as the external lock register. The external lock register is a backup copy of the Alpha 21164 microprocessor's lock register and is used only when instruction stream prefetching causes the locked address to be evicted from the B-cache. The execution of a load with lock instruction by an Alpha 21164 microprocessor results in a transaction that sets both internal and external lock flags and lock address registers.

The external lock flag is cleared by the system interface if the lock address matches the system bus address of either a write transaction or an exclusive transaction. The internal lock flag is cleared by the Alpha 21164 microprocessor due to system bus probe transactions from the write or exclusive transaction to a valid cache block.

The lock address resolution is a single-aligned 32-byte block and is consistent with the size of cache blocks in this system. The Alpha 21164 microprocessor has 64-byte internal lock register resolution. Since the address of a load to memory and the corresponding store to memory must both be within the same 16-byte aligned region, the difference in the resolution of the internal and the external lock registers was determined to be insignificant to performance.[6]

The KN470 Module and System Bus Clocking

The KN470 module implements a low-cost synchronous clocking scheme. The scheme exploits the system bus clocking to run the Alpha 21164 microprocessor synchronous to the system bus. This scheme compensates for the half-cycle correction phase of the Alpha 21164 microprocessor's digital lock loop (DLL).

The AlphaServer 2100 system interconnect has an edge-to-edge clock architecture, and it implements an edge-to-edge data transfer scheme. The microprocessor has an internal DLL that synchronizes to a reference clock supplied by the clock repeater chip. Instead of trying to precisely control the clock skew across four different chips, data valid windows are set around the edge-to-edge data transfer clock edges to avoid setup or hold-time issues. This simpler clocking scheme takes advantage of the four delivered clock edges per cycle from the clock repeater chips. It also enables a simpler synchronous boundary between the Alpha 21164 microprocessor and the system interface. The synchronous clocking improves data transfer rates, lowers the miss penalty, and improves the pipeline efficiency among the components of the system.

Figure 2 shows the clocking scheme that is implemented on the KN470 module. The Alpha 21164 microprocessor accepts a differential clock at twice the desired internal clock frequency. The oscillator for the processor runs at 6, 7, 8, or 9 times the 41.66 megahertz (MHz) system bus clock frequency. The DLL subtracts one half of an internal clock cycle to maintain phase alignment with the system bus reference clock.

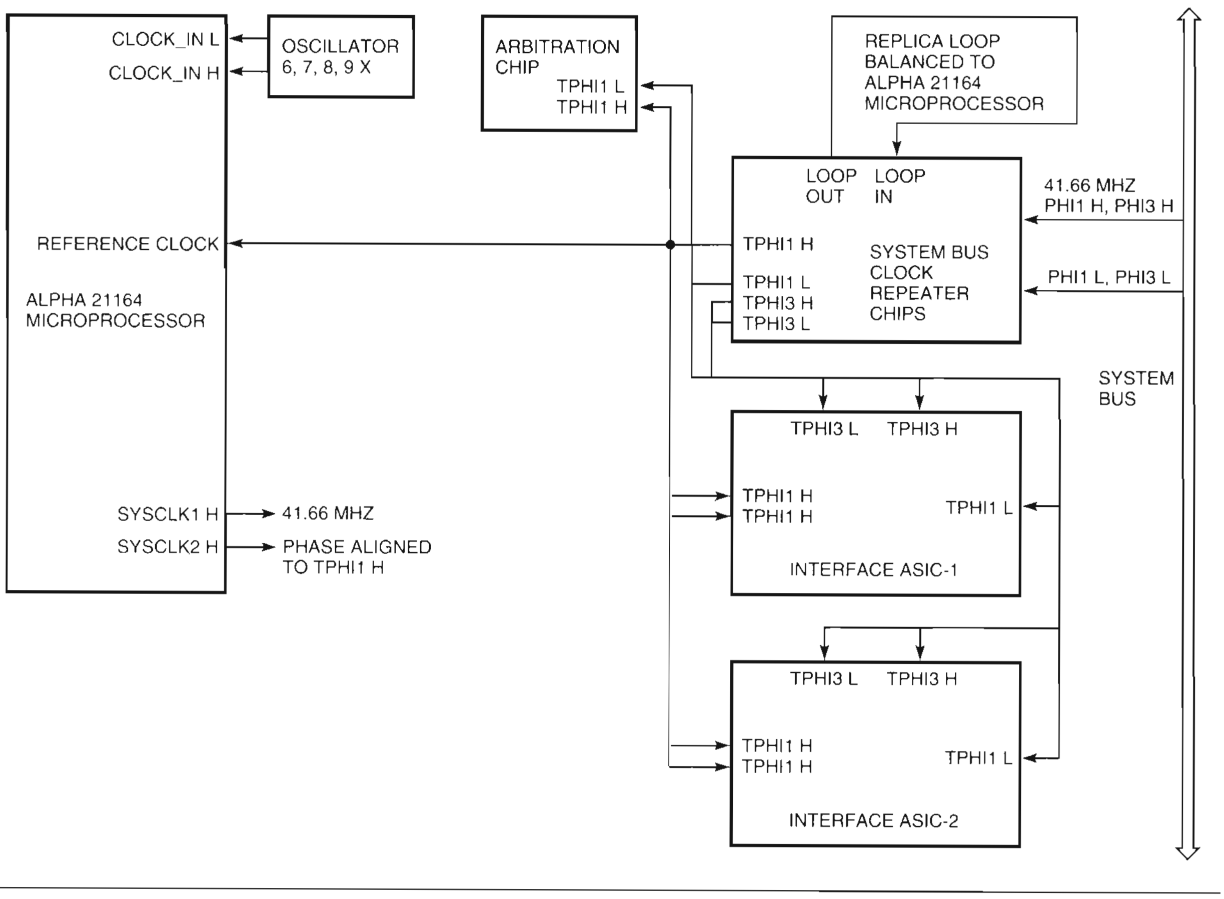


Figure 2
KN470 Clocking Scheme

This DLL scheme assumes that the internal clock frequency runs slightly faster than the system bus clock frequency. Given these scaling rates, the interface between the Alpha 21164 microprocessor and the system interface are locked at the system bus clock rate.

The AlphaServer 2100 backplane distributes PECL-level system bus clocks PHI1 L and PHI1 H, and PHI3 L and PHI3 H differentially to each module on the system bus. Each module receives, terminates, and capacitively couples the clock signals into PECL-to-CMOS–level converters to provide four edges per system bus clock cycle. This level conversion is completed in the clock repeater chips. System bus handshake and data transfers occur from clock edge to clock edge and thus form a primary clock in the system. The remaining three edges in a clocking cycle are secondary clocks. The clock repeater chip, a custom CMOS clock chip, provides module-to-module clock skew of less than 1 nanosecond (ns) and is implemented to provide skew-regulated clock copies to be consumed by components on the module. The skew regulation is maintained by the repeater chip through the use of a feedback path or replica loop of the primary clock path. The KN470 module uses this clock repeater chip to generate the references for synchronous clocking from a central point.

Components on the module are clocked by outputs from the clock repeater chips. The clock repeater chips generate six copies of the primary clock TPHI1 H. TPHI1 H clocks are distributed as follows: one copy to the Alpha 21164 microprocessor, two copies to each of the two system interface ASICs, and one copy to the system bus arbiter chip. The Alpha 21164 microprocessor uses its copy of the primary clock as a reference clock for its DLL. The data transfers between the microprocessor and the system interface are edge-to-edge transfers and are referenced to the primary clock. The clock repeater chip generates three secondary clocks: TPHI1 L, TPHI3 H, and TPHI3 L. The clock-edge relationships among these four clocks are specified such that each clock edge is 90 degrees out of phase with the other two clock edges. The relationships among the different clock phases are shown in Figure 3 for the case of the Alpha 21164 oscillator with a frequency six times that of the system bus clock. The system interface uses all three secondary clocks for on-chip data transfers, whereas the arbiter chip uses one secondary clock, TPHI1 L.

This synchronous clocking scheme works well if the driver turn-on and turn-off times are extremely fast for all components. However, the technologies selected could not guarantee such speed. The Alpha 21164

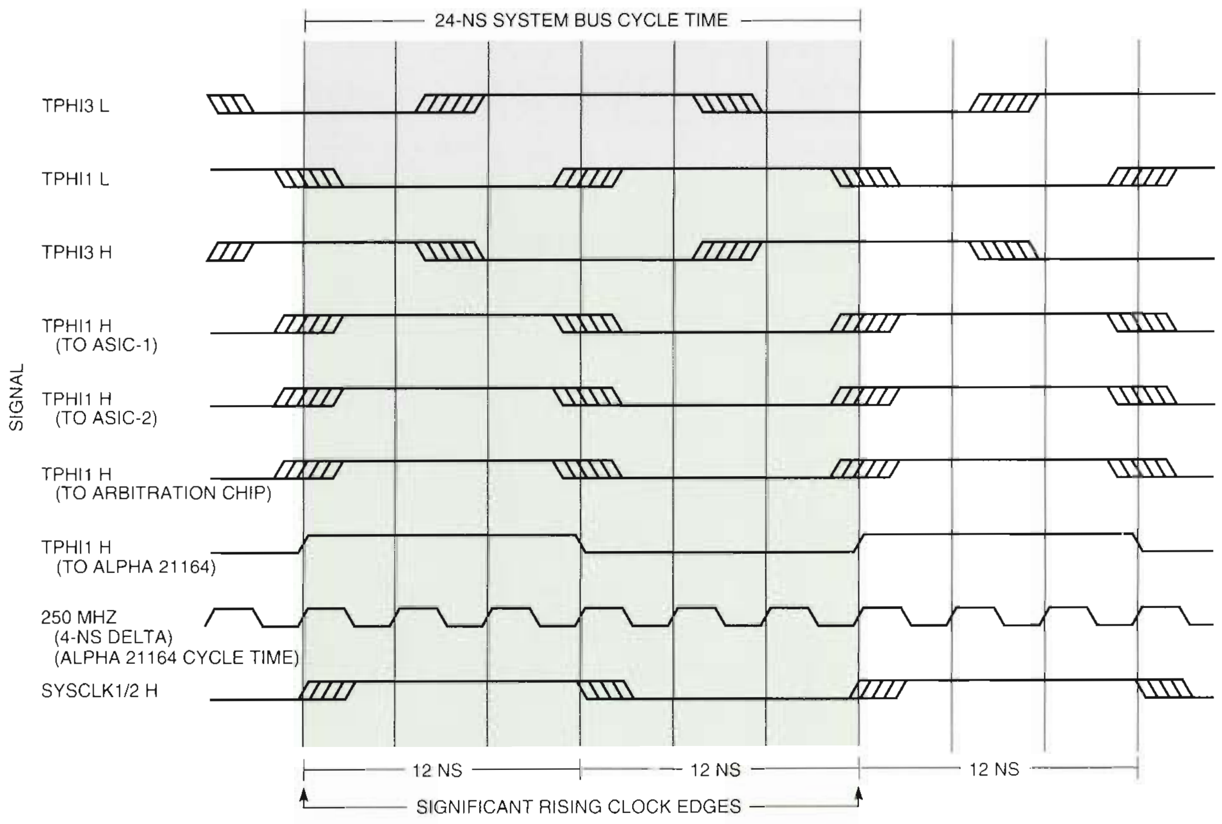

24-NS SYSTEM BUS CYCLE TIME
SIGNAL
TPHI3 L
TPHI1 L
TPHI3 H
TPHI1 H (TO ASIC-1)
TPHI1 H (TO ASIC-2)
TPHI1 H (TO ARBITRATION CHIP)
TPHI1 H (TO ALPHA 21164)
250 MHZ (4-NS DELTA) (ALPHA 21164 CYCLE TIME)
SYSCLK1/2 H
12 NS
12 NS
12 NS
SIGNIFICANT RISING CLOCK EDGES

Figure 3
Relationships among Different Clock Phases

microprocessor driver turn-on and turn-off times are fast, but the ASICs have slow turn-on and turn-off times. To compensate for the fast and slow driver characteristics, the edge-to-edge clocking scheme required a modification. The Alpha 21164 microprocessor uses its copy of TPHI1 H as the reference clock edge to align its SYSCLK1/2 H–generated interface output clocks. Though SYSCLK1/2 H does not physically connect to the system interface, the Alpha 21164 microprocessor uses the internal copy of the SYSCLK1/2 H edge to either drive data or receive data. The system interface uses its copy of the reference clock as the data receive edge for signaling from the Alpha 21164 microprocessor. To drive the data to both the microprocessor and the B-cache, the system interface uses the TPHI3 L secondary clock, which is phase-delayed 90 degrees from the primary clock TPHI1 H.

The above clocking scheme achieves single-clock, edge-to-edge data transfer rates without imposing overly strict constraints on clock routing and layout. The scheme can withstand larger than 1 ns of clock skew and compensates for the Alpha 21164 microprocessor's DLL half-cycle correction between the reference clock and SYSCLK1/2 H.

Design Trade-offs

The KN470 module design achieved aggressive schedule goals and achieved lower cost by means of the bit-slice design of the system interface. Also, the higher performance goal was realized while keeping the design complexity at a moderate level.

The bit-slice design of the system interface was motivated by the organization of the Alpha 21164 microprocessor's 64-bit error-correcting code–protected data bus. This forced at least a 64-bit slice organization. Other organizations were found to have too many pins or would have encountered system bus signal integrity problems because of long stubs and additional loads. The decision to also include the address and control functions was further motivated by the project's human resource constraints and its spending constraints. Designing one ASIC as a slice to implement the 128-bit-wide system interface was found to be the best choice.

The system interface controls the address and data paths between the Alpha 21164 microprocessor and the system bus. The system interface does not stall the system bus on transactions that require cache state changes in the B-cache. Instead, the interface posts a pended request to the processor for changing the cache state of the B-cache. The system interface stalls the system bus when the processor has not acknowledged a previously pended request and the present transaction on the system bus needs a cache state change request. At the cost of increased complexity, the design could have been implemented such that the system bus would not stall in the absence of acknowledgments of previously pended requests. This level of complexity avoided the more complex issues of managing a queue of block invalidate, set block to shared, and read block transaction requests.

The KN470 module design implements a scheme of write transaction back-off or replay that exploits the transaction replay queue of the Alpha 21164 microprocessor. This replay functionality helps the system interface handle cache state changes when simultaneous requests to write to the system bus and to invalidate from the system bus are made to the same cache block. The designers simplified the cache coherence management and logic design by avoiding the use of a pended write transaction in the system interface, which would have required a one-block write cache.

A write transaction from the Alpha 21164 microprocessor to the system bus is not considered complete until the system bus is granted. This nonpended scheme for write transactions enables write transaction replay from the Alpha 21164 microprocessor and avoids the requirement for the system interface to preserve logic states if a system bus transaction takes precedence. When the system bus transaction takes precedence, the system interface removes the arbitration request, signals the Alpha 21164 microprocessor to replay the write transaction, and flushes all states associated with the write transaction. The Alpha 21164 microprocessor must determine whether the write transaction has been affected by the change in its cache state and then decide to replay the write transaction or to perform another transaction such as a read transaction to revalidate the block.

Removing a system bus request from the arbiter chip rather than converting the write transaction to a no-operation transaction avoided a livelock condition. The livelock condition could have resulted from the system interface's completion of a no-operation transaction and re-requesting the system bus to complete the write transaction. While waiting for the grant to this second arbitration request, the system bus could force the Alpha 21164 microprocessor to replay the write again. In addition to avoiding the livelock condition, the replay scheme has the additional benefit of conserving bandwidth by not issuing no-operation transactions while the system bus interface is waiting for the Alpha 21164 microprocessor to replay the write transaction.

Removing a system bus request in response to other bus transactions reduces the probability of a timely completion of the write transaction from the Alpha 21164 microprocessor. More complex design approaches increase the probability that the write transaction will complete, but they do not guarantee the completion. This is a result of the uncertain time for a response from the Alpha 21164 microprocessor to replay the write transaction in relation to the next system bus grant. The designers chose the simpler implementation to reduce logic design complexity and verification time.

Performance of AlphaServer 2100 Systems with KN470 Modules

To validate the improved performance goal of the KN470 processor module in AlphaServer 2100 systems running Digital UNIX (formerly DEC OSF/1) version 3.2B, project engineers measured several industry-standard benchmarks. A brief description of each benchmark follows. Table 1 lists the benchmarks that were run on an AlphaServer 2100 Model 5/250 system, the number of processor modules in a configuration for each benchmark, the measured estimates or unaudited results of the benchmark, and the performance gain. Performance gain is reported as a ratio of the KN470 result to the top-performing, first-generation KN460 result. The ratios demonstrate that the KN470 processor module achieves the primary project goal by providing more performance to AlphaServer 2100 systems than the first-generation KN460 processor.

The AlphaServer 2100 Model 5/250 system uses the KN470 processor module that incorporates the Alpha 21164 microprocessor operating at 250 MHz with a 4-MB B-cache. The AlphaServer 2100 Model 4/275 system uses the KN460 processor module with the Alpha 21064 microprocessor operating at 275 MHz with a 4-MB B-cache. The AlphaServer 2100 system remained fixed as the processor models were swapped for these performance measurements.

The Standard Performance Evaluation Corporation (SPEC) was formed to identify and create objective sets of applications-oriented tests, which can serve as common reference points. SPEC CINT92 is a good base indicator of CPU performance in a commercial environment. This benchmark is the geometric mean of ratios by which the six benchmarks in this suite exceed the performance of the reference machine. SPEC CFP92 may be used to compare floating-point intensive environments, typically engineering and scientific applications. SPEC CFP92 is the geometric mean of ratios by which the 14 benchmarks in this suite exceed the performance of the reference machine. SPEC Homogeneous Capacity Method benchmarks test multiprocessor efficiency. They provide a fair measure for the processing capacity of a system, namely, how much work the system can perform in a given amount of time. The SPECrate is a capacity measure. It is not a measure of how fast a system can perform any task but of how many of those tasks the system completes within an arbitrary time interval.

Developed by AIM Technology, the AIM Suite III Benchmark Suite was designed to measure, evaluate, and predict UNIX multiuser system performance. The benchmark suite uses 33 functional tests, and these tests can be grouped to reflect the computing activities of various types of applications. The AIM Performance Ratings identify the maximum performance of the system under optimum usage of CPU, floating-point, and disk caching. At a system's peak performance, an

Table 1
Performance Data for an AlphaServer 2100 System That Incorporates the KN470 Processor Module

Benchmark	Number of Processor Modules per Configuration	AlphaServer 2100 Model 5/250	Performance Gain Expressed As a Ratio of Model 5/250 Performance to Model 4/275 Performance
SPEC CINT92			
SPECint92	1	277	1.4
SPECrate_int92	4	24,996	1.4
SPEC CFP92			
SPECfp92	1	410	1.4
SPECrate_fp92	4	37,926	1.4
AIM Suite III Benchmark Suite Performance (Estimated)			
Performance Rating	2	396	
Maximum User Loads		2,400	1.4
Performance Rating	4	719	
Maximum User Loads		3,100	1.3
LINPACK (MFLOPS)			
1000 × 1000	4	1,022	1.6
McCalpin			
copy	2	171	1.28
scale	2	169	1.27
sum	2	162	1.25
triad	2	162	1.27

increase in the workload will cause a deterioration in performance. The AIM Maximum User Load Rating identifies system capacity under heavy multitasking loads, where disk performance also becomes a significant factor. Throughput is the total amount of work the system processes, measured in jobs per minute. Maximum throughput is the point at which the system is able to process the most jobs per minute.

The LINPACK benchmark is a linear equation solver written in FORTRAN. LINPACK programs consist of floating-point additions and multiplications of matrices. The LINPACK 1000 × 1000 solves a 1,000-by-1,000 matrix of simultaneous linear equations. The result is a measure of the execution rate in millions of floating-point operations per second (MFLOPS).

The McCalpin benchmark is a public domain set of programs that measures the effective memory bandwidth available to each processor in MB per second. The four parts of this benchmark, which are shown in Figure 4, perform a double-precision operation *j* times, where *j* increments 2 million times. Often, the four numbers are averaged to show an effective memory bandwidth rating for the configuration.

Table 2 shows estimated AIM Suite III Benchmark Suite performance scaling for AlphaServer configurations of one to four processor modules. These results validate improvements in the ability of KN470 processor modules to scale in multiprocessor configurations.

Summary

The implementation of the write-invalidate coherence protocol combined with synchronous clocking, a duplicate tag store, and pipelining cache-miss requests led to a more efficient use of the system bus bandwidth. A higher complexity design could have been realized but only at the risk of missing schedule deadlines. The KN470 processor development project achieved the goals of AlphaServer 2100 compatibility and performance improvement that were established early in the project.

Acknowledgments

The development of this new generation of processor and its integration into the AlphaServer 2100 family required the outstanding dedication and contributions from many individuals. The authors wish to extend a large thank you to Steve Holmes for believing in and supporting this project. The authors also wish to acknowledge the key contributors to the project. The core design team of Chet Pawlowski, Jim Padgett, Judy Weiss Prescott, and Gary Zeltser devoted long hours from the concept development through the empirical verification and the manufacturing startup of this processor module and system model. The verification team of Abdollah Ataie, Erik Debriae, Norbert Eng, Jeff Metzger, Don Caley, Dick Beaven, and Ginny Lamere proved the design's integrity. Andy Ebert provided the ASIC test strategy and applications support. Peter Woods and Traci Post were responsible for the serial control subsystem and diagnostics. Stephen Shirron made the OpenVMS system boot on the new machine, and Kevin Peterson and Harold Buckingham provided the firmware and console bits and consultation regarding software issues. Janet Walsh and Jeff Kerrigan contributed operations support. Steve Brooks, Rich Freiss, and Dean Gagne

```
copy  c(j) = a(j)                  ;copy a to c
scale b(j) = 3.0 * c(j)            ;multiply c times 3, store
                                   ; result in b
sum   c(j) = a(j) + b(j)           ;add a to b and store in c
triad a(j) = b(j) + 3.0 * c(j)     ;multiply c times 3, add to
                                   ; b, store result in a
```

Figure 4
The Four Parts of the McCalpin Benchmark

Table 2
AIM Suite III Benchmark Suite Performance Scaling (Estimated)

	AlphaServer 2100 System			
Number of Processor Modules	1	2	3	4
Maximum Throughput Jobs/Minute	2,178	3,882	5,249	7,047
Model 5/250 Scaling	1.0	1.8	2.4	3.2
Maximum Throughput Jobs/Minute	1,451	2,229	2,998	3,587
Model 4/275 Scaling	1.0	1.5	2.1	2.5

developed the operating system software that supports this system. Simon Steely, Zarka Cvetanovic, and John Shakshober carried out performance analysis and validations. John Edmondson, Pete Bannon, Anil Jain, and Paul Rubinfeld designed the KN470-specific functionality in the Alpha 21164 microprocessor.

References

1. F. Hayes, "Design of the AlphaServer Multiprocessor Server Systems," *Digital Technical Journal*, vol. 6, no. 3 (Summer 1994): 8–19.
2. B. Maskas, S. Shirron, and N. Warchol, "Design and Performance of the DEC 4000 AXP Departmental Server Computing Systems," *Digital Technical Journal*, vol. 4, no. 4 (Special Issue 1992): 82–99.
3. J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," *Digital Technical Journal*, vol. 7, no. 1 (1995, this issue): 119–135.
4. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach* (San Mateo, Calif.: Morgan Kaufmann, 1990): 467–474.
5. A. Russo, "The AlphaServer 2100 I/O Subsystem," *Digital Technical Journal*, vol. 6, no. 3 (Summer 1994): 20–28.
6. R. Sites, ed., *Alpha Architecture Reference Manual* (Burlington, Mass.: Digital Press, Order No. EY-l520E-DP, 1992).

Biographies

Nitin D. Godiwala
Nitin Godiwala is a principal engineer in Digital's Server Product Development Group. His area of expertise is digital system architecture and pipelined machines. As a contributor to the AlphaServer 2100 server product, he was the project leader and principal architect of the arbitration ASIC and the system interface ASICs for the Alpha 21064 and 21164 microprocessor-based processor modules. In previous work, he was a principal architect and designer of the system interface ASIC for the DEC 4000 processor module. Before coming to Digital in 1986, Nitin worked for Analogic Corp., Gould Modicon, and Honeywell Inc. He received a B.E. from Bombay University and an M.S. in computer and electrical engineering from the University of Wisconsin, Madison. He holds four patents and has eight patents pending.

Barry A. Maskas
A consultant engineer in Digital's Server Product Development Group, Barry Maskas was the project leader responsible for the development of the Alpha 21164 microprocessor-based AlphaServer 2100 and 2000 processors and systems. He is currently involved in further Alpha-based server system development work. In earlier work, Barry was the project leader and architect for the DEC 4000 system bus, backplane, and processor modules and for the architecture and development of custom VLSI peripheral chip sets for VAX 4000 and MicroVAX systems. He was also co-designer of the MicroVAX II processor and memory modules. Barry joined Digital in 1979 after receiving a B.S.E.E. from Pennsylvania State University. He holds eight patents.

John H. Zurawski
John E. Murray
Paul J. Lemmon

The Design and Verification of the AlphaStation 600 5-series Workstation

The AlphaStation 600 5-series workstation is a high-performance, uniprocessor design based on the Alpha 21164 microprocessor and on the PCI bus. Six CMOS ASICs provide high-bandwidth, low-latency interconnects between the CPU, the main memory, and the I/O subsystem. The verification effort used directed, pseudorandom testing on a VERILOG software model. A hardware-based verification technique provided a test throughput that resulted in a significant improvement over software tests. This technique currently involves the use of graphics cards to emulate generic DMA devices. A PCI hardware demon is under development to further enhance the capability of the hardware-based verification.

The high-performance AlphaStation 600 5-series workstation is based on the fastest Alpha microprocessor to date—the Alpha 21164.[1] The I/O subsystem uses the 64-bit version of the Peripheral Component Interconnect (PCI) and the Extended Industry Standard Architecture (EISA) bus. The AlphaStation 600 supports three operating systems: Digital UNIX (formerly DEC OSF/1), OpenVMS, and Microsoft's Windows NT. This workstation series uses the DECchip 21171 chip set designed and built by Digital. These chips provide high-bandwidth, low-latency interconnects between the CPU, the main memory, and the PCI bus.

This paper describes the architecture and features of the AlphaStation 600 5-series workstation and the DECchip 21171 chip set. The system overview is first presented, followed by a detailed discussion of the chip set. The paper then describes the cache and memory designs, detailing how the memory design evolved from the workstation's requirements. The latter part of the paper describes the functional verification of the workstation. The paper concludes with a description of the hardware-based verification effort.

System Overview

The AlphaStation 600 5-series workstation consists of the Alpha 21164 microprocessor, a third-level cache that is external to the CPU chip, and a system chip set that interfaces between the CPU, the memory, and the PCI bus. The DECchip 21171 chip set consists of three designs: a data slice, one PCI interface and memory-control chip (called the control chip), and a miscellaneous chip that includes the PCI interrupt logic and flash read-only memory (ROM) control. The Intel 82374 and 82375 chip sets provide the bridge to the EISA bus.[2] Figure 1 shows a block diagram of the workstation.

The SysData bus transfers data between the processor, the CPU's tertiary cache, and the data slices. The 128-bit-wide SysData bus is protected by error-correcting code (ECC) and is clocked every 30 nanoseconds (ns). The data slices provide a 256-bit-wide data path to memory. Data transfers between the

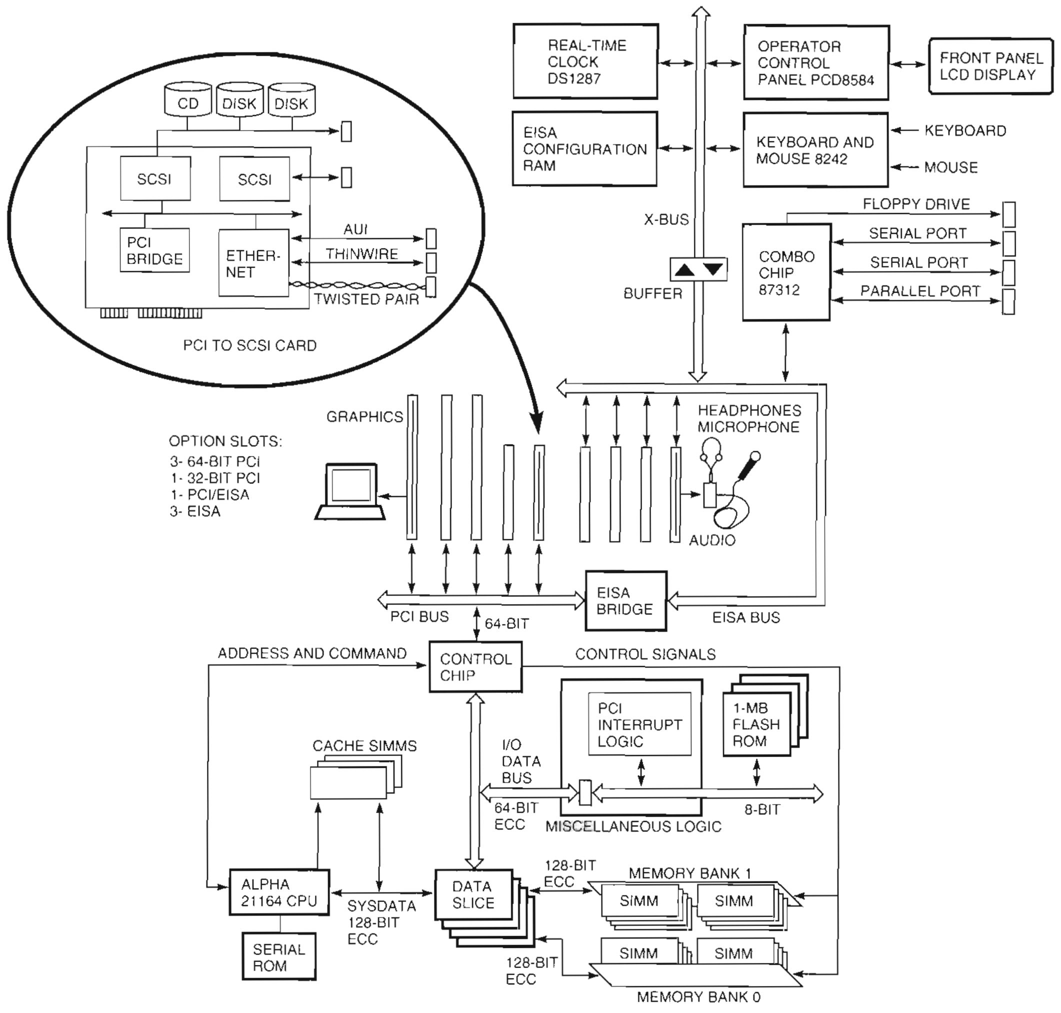


Figure 1
AlphaStation 600 5-series Workstation Block Diagram

PCI and the processor, the external cache (typically 4 megabytes [MB]), and memory take place through the control chip and four data slices. The control chip and the data slices communicate over the 64-bit, ECC-protected I/O data bus.

The major components and features of the system board are the following:

- The Alpha 21164 microprocessor supports all speed selections from 266 to 333 megahertz (MHz).
- The plug-in, external write-back cache (2 MB to 16 MB) has a block size of 64 bytes. Access time is a multiple of the processor cycle time and is dependent on the static random-access memory (SRAM) part used. With 12-ns SRAMs, typical access times are 24 ns for the first 128 bits of data, 21 ns for remaining data.
- The system board contains a 256-bit data path to memory (284 megabytes per second [MB/s] for sustained CPU reads of memory).
- From 32 MB to 1 gigabyte (GB) of main memory can be used in industry-standard, 36-bit, single in-line memory modules (SIMMs). All memory banks support single-sided and double-sided SIMMs.
- Eight option slots are available for expansion: four PCI, three EISA, and one PCI/EISA shared slot. The system design minimized logic on the mother board in favor of more expansion slots, which allow customers to configure to their requirements. The system uses option cards for small computer systems interface (SCSI), Ethernet, graphics, and audio.
- The system supports 64-bit PCI address and data capability.
- Due to its synchronous design, the system's memory, cache, and PCI timing are multiples of processor cycle time.
- The system provides an X bus for the real-time clock, keyboard controller, control panel logic, and the configuration RAM.

Data Slice Chips

Four data slice chips implement the primary data path in the system. Collectively, the data slices constitute a 32-byte bus to main memory, a 16-byte bus to the CPU and its secondary cache, and an 8-byte bus to the control chip (and then to the PCI bus).

Figure 2 shows a block diagram of the data slice chip. The data slice contains internal buffers that provide temporary storage for direct memory access (DMA), I/O, and CPU traffic. A 64-byte victim buffer holds the displaced cache entry for a CPU fill operation. The Memory-Data-In register accepts 288 bits (including ECC) of memory data every 60 ns. This register clocks the memory data on the optimal 15-ns clock to reduce memory latency. The memory data then proceeds to the CPU on the 30-ns, 144-bit bidirectional data bus. A set of four, 32-byte I/O write buffers help maximize the performance of copy operations from memory to I/O space. A 32-byte buffer holds the I/O read data. Finally, there are a pair of DMA buffers, each consisting of three 64-byte storage areas. DMA read operations use two of these three locations: the first holds the requested memory data, and the other holds the external cache data in the case of a cache hit. DMA writes use all three locations: one location holds the DMA write data, and the other two hold the memory and cache data used during a DMA write merge.

The data slice allows for simultaneous operations. For instance, the I/O write buffers can empty to the control chip (and then to the PCI) while a concurrent read from CPU to main memory is in progress.

Control Chip

The control chip controls the data slices and main memory and provides a fully compliant host interface to the 64-bit PCI local bus. The PCI local bus is a high-performance, processor-independent bus, intended to interconnect peripheral controller components to a processor and memory subsystem. The PCI local bus offers the promise of an industry-standard interconnect, suitable for a large class of computers ranging from personal computers to large servers.

Figure 3 shows a block diagram of the control chip. The control chip contains five segments of logic:

- The address and command interface to the Alpha 21164 microprocessor
- The data path from the PCI bus to the data slices by means of the I/O data bus
- DMA address logic, including a 32-entry scatter/gather (S/G) map (This is discussed in the section Scatter/Gather Address Map.)
- Programmed I/O read/write address logic
- The memory address and control logic

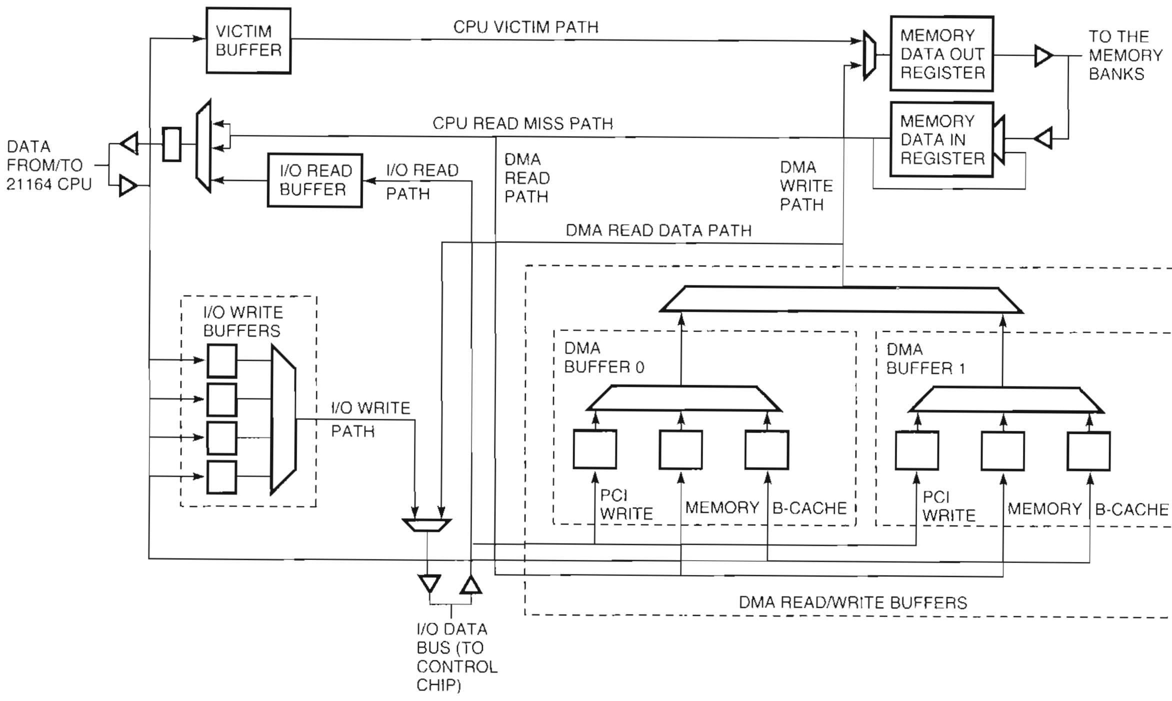


Figure 2
Data Slice Block Diagram

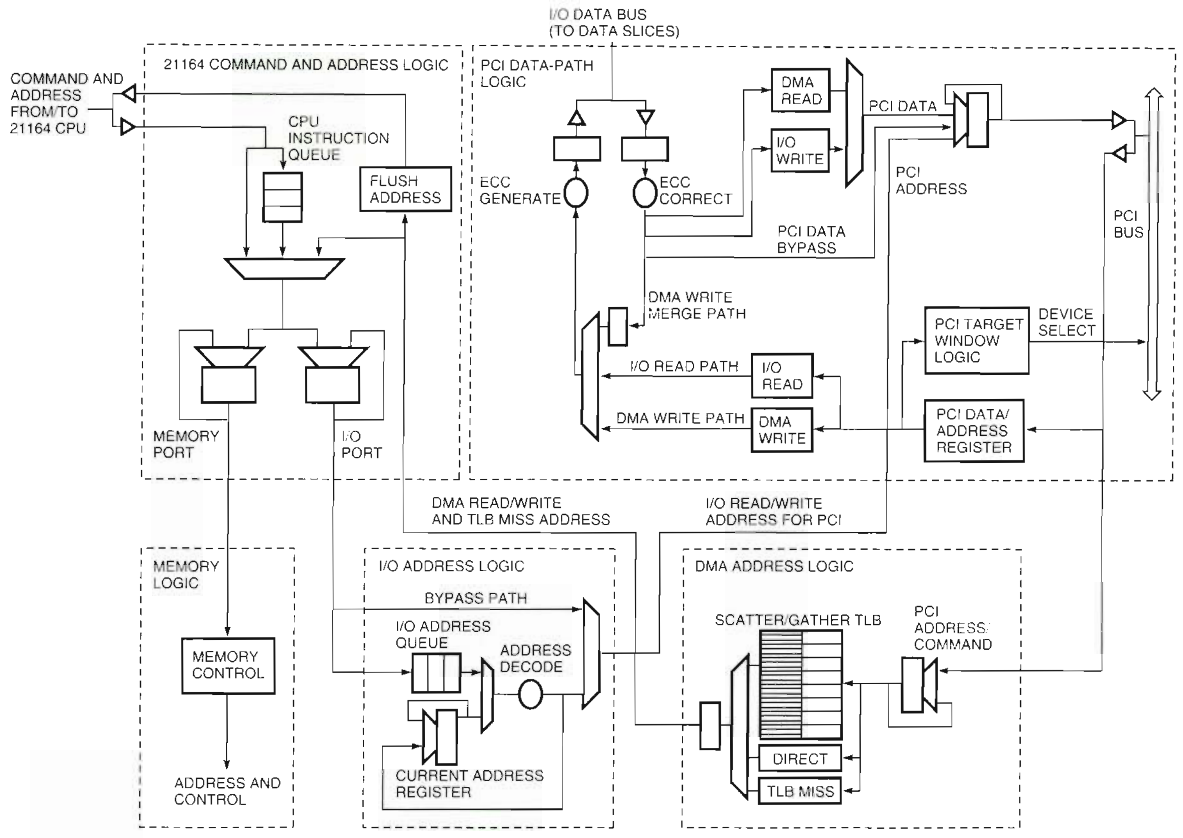


Figure 3
Control Chip Block Diagram

CPU Interface A three-deep queue can hold two outstanding read requests, together with the address of a victim block associated with one of these read requests. During a DMA write, the Flush Address register holds the address of the cache block that the CPU must move to memory (and invalidate in the cache). In this manner, the system maintains cache coherency during DMA write operations.

PCI Address Space Windows PCI devices use address space windows to access main memory. During discussions with the developers of the operating system, we determined that four PCI address space windows would be desirable. EISA devices use one window. S/G mapping uses a second. The third window directly maps a contiguous PCI address region to a contiguous region of main memory. The fourth window supports 64-bit PCI addresses. Future system designs may provide more than 4 GB of main memory, thus requiring the 64-bit address window.

DMA Write Buffering The control chip provides a single-entry, 64-byte DMA write buffer. Once the buffer is full, the data is transferred to the DMA buffers in the data slices. The design can support 97-MB/s DMA write bandwidth from a 32-bit PCI device.

DMA Read Buffering In addition to the two 64-byte buffers inside the data slice, the control chip has two 32-byte DMA read buffers. These buffers prefetch DMA read data when the initiating PCI read command so indicates. This arrangement provides data to a 64-bit PCI device at a rate of more than 260 MB/s.

Scatter/Gather Address Map The S/G mapping address table translates contiguous PCI addresses to any arbitrary memory address on an 8-kilobyte (KB) granularity. For software compatibility with other Alpha system designs, the S/G map uses a translation lookaside buffer (TLB).[3] The designers enhanced the TLB: First, each of the eight TLB entries holds four consecutive page table entries (PTEs). This is useful when addressing large 32-KB contiguous regions on the PCI bus. For instance, the NCR810 PCI-to-SCSI device requires nearly 24 KB of script space.[4] Second, software can lock as many as one half of the TLB entries to prevent the hardware-controlled replacement algorithm from displacing them. This feature reduces TLB thrashing.

Programmed I/O (PIO) Writes The designers focused on improving the performance of the functionality that allows a processor to copy from memory to I/O space. High-end graphics device drivers use this functionality to load the graphics command into the device's first-in, first-out (FIFO) buffer. The data slice has four buffers, and the control chip contains the corresponding four-entry address queue. Four buffers hold enough I/O write transactions to mask the latency of the processor's read of memory. The control chip provides two additional 32-byte data buffers. While one drives data on the PCI bus, the other accepts the next 32 bytes of data from the data slices.

Memory Controller The memory controller logic in the control chip supports as many as eight banks of dynamic random-access memory (DRAM). The current memory backplane, however, provides for only 4 banks, allowing from 32 MB to 1 GB of memory. The memory controller supports a wide range of DRAM sizes and speeds across multiple banks in a system. Registers program the DRAM timing parameters, the DRAM configuration, and the base address and size for each memory bank. The memory timing uses a 15-ns granularity and supports SIMM speeds ranging from 80 ns down to 50 ns.

Cache Design

The Alpha 21164 microprocessor contains significant on-chip caching: an 8-KB virtual instruction cache; an 8-KB data cache; and a 96-KB, 3-way, set-associative, write-back, second-level mixed instruction and data cache. The system allows for an external cache as a plug-in option. This cache is typically 2 MB to 4 MB in size, and the block size is 64 bytes. The access time for the external cache depends on the CPU frequency and the speed variant of the cache. Typically, the first data requires 7 to 8 CPU cycles; subsequent data items require 1 or 2 fewer cycles. The actual value depends on both the minimum propagation time through the cache loop and on the CPU cycle time. The external cache data bus is 16 bytes wide, providing almost 1 GB/s of bandwidth with a 333-MHz CPU and a 5-cycle cache access.

The processor always controls the external cache, but during a cache miss, the system and the processor work together to update the cache or displace the cache victim. For an external cache miss, the system performs four 16-byte loads at 30 ns. Any dirty cache block is sent to the victim buffer in the data slices, in parallel with the read of memory. Fast page-mode memory writes are used to write the victim into memory quickly. (This is discussed in the section Memory Addressing Scheme.)

During DMA transactions, the system interrogates the CPU for relevant cache data. There is no duplicate tag in the system. DMA reads cause main memory to be read in parallel with probes of the CPU's caches. If a cache probe hits, the cache data is used for the DMA read; otherwise main memory data is used. Each DMA write to memory results in a FLUSH command to the CPU. If the block is present in any of the caches, then the data is sent to the DMA buffers in the data slice and the cache blocks are invalidated. This cache data is discarded if the DMA write is sent to a complete block. In the case of a DMA write to a partial block, the DMA write data is merged with cache data or the memory data as appropriate. In this manner, the system maintains cache coherency, removing this burden from the software.

Memory Bandwidth

The memory bandwidth realized by the CPU depends on a number of factors. These include the cache block size, the latency of the memory system, and the data bandwidth into the CPU.

Cache Block Size

The Alpha 21164 microprocessor supports either a 32- or 64-byte cache block size. The AlphaStation 600 workstation uses the 64-byte size, which is ideal for many applications, but suffers on certain vector-type programs with contiguous memory references.[5] An example of a larger block size design is the RISC System/6000 Model 590 workstation from International Business Machines Corporation.[6] This design supports a 256-byte cache block size, allowing it to amortize a long memory latency by a large memory fetch. For certain vector programs, the Model 590 performs well; but in other applications, the large block size wastes bandwidth by fetching more data than the CPU requires.

The AlphaStation 600 provides a hardware feature to gain the benefit of a larger block size when appropriate. The Alpha 21164 microprocessor can issue a pair of read requests to memory. If these two reads reside in the same memory page, the control chip treats them as a single 128-byte memory read. In this way, the system approximates the benefit of a larger block and achieves 284 MB/s of memory read bandwidth.

Memory Latency

The 180-ns memory latency consists of five parts. First, the address is transferred from the microprocessor to the control chip in 15 ns. The control chip sends the memory row-address pulse 15 ns later, and the data is received by the data slices 105 ns later. The data slices require 15 ns to merge the wider memory data onto the narrower SysData bus, and the last 30 ns are spent updating the external cache and loading the Alpha 21164 microprocessor.

Although the 105 ns to access the memory may appear to be generous, the designers had to meet the significant challenge of implementing the required

1 GB of memory with inexpensive 36-bit SIMMs. The JEDEC standard for these SIMMs only specifies the pinning and dimensions. It does not specify the etch lengths, which can vary by many inches from vendor to vendor. Neither does it specify the electrical loading distribution, nor the DRAM type or location (1-bit parts have 2 data loads whereas 4-bit parts have a single, bidirectional load). With a 1-GB memory system, the loading variation between a lightly loaded memory and a fully loaded memory is significant. All these factors contributed to significant signal-integrity problems with severe signal reflections. The memory mother-board etch was carefully placed and balanced, and numerous termination schemes were investigated to dampen the signal reflections.

Data Bandwidth

The SysData bus transfers data between the processor, the tertiary cache, and the data slices. This 128-bit bus is clocked every 30 ns to satisfy the write timing of the external cache and to be synchronous with the PCI bus. Typical memory DRAM parts cycle at 60 ns, thus requiring a 32-byte-wide memory bus to match the bandwidth of the SysData bus. The data slice chips reduce each 32-byte-wide memory data transfer to two 16-byte transfers on the SysData bus. Consequently, the memory system is logically equivalent to a 2-way interleaved memory design.

New memory technologies with superior data bandwidths are becoming available. Synchronous DRAMs are an exciting technology, but they lack a firm standard and are subject to a significant price premium over plain 5-volt DRAM parts. Extended-data-out (EDO) DRAMs allow greater burst memory bandwidth, but the latency to the first data is not reduced. Consequently, the memory bandwidth to the CPU is not significantly improved. The major advantage of using EDO parts is their easier memory timing: The output data of EDO parts is valid for a longer period than standard DRAMs. In addition, an EDO memory can be cycled at 30 ns, which allows a 128-bit memory width instead of the 256-bit width. The designers would have used EDO parts had they been available earlier.

Memory Addressing Scheme

The adopted addressing scheme helps improve memory bandwidth. Whenever the CPU requests a new block of data, the write-back cache may have to displace current data (the victim block) to allow space for the incoming data. The writing of the victim block to memory should occur quickly, otherwise it will impede the CPU's request for new data.

Figure 4 shows the method used to address the external cache and memory. The CPU address <31:6> directly accesses the cache: the low-order bits <19:6> form the index for a 1-MB cache, and the remaining bits <31:20> form the cache tag. The CPU address does not directly address memory. Instead, the memory address interchanges the index portion of the address field with the tag portion. The number of address bits interchanged depends on the row and column dimensions of the DRAM used.

For the sake of discussion, assume a 4-megabit (Mb) DRAM configured with 11 row address bits and 11 column address bits. Hence, bits <30:20> interchange with bits <16:6>, and the remaining bits select the memory bank. This addressing scheme has the following effect: a CPU address that is incrementing by units of 1 MB now accesses consecutive memory locations. DRAM memory provides a fast addressing mode, called page mode, whenever accessing consecutive locations. For a 1-MB cache, objects separated by a multiple of 1 MB correspond to cache victim blocks. Consequently, a CPU read request of memory that involves a victim write to memory gains the benefit of page mode and proceeds faster than it would with a traditionally addressed memory.

Although this address scheme is ideal for CPU memory accesses, it creates the converse effect for DMA transactions. It scatters consecutive DMA blocks

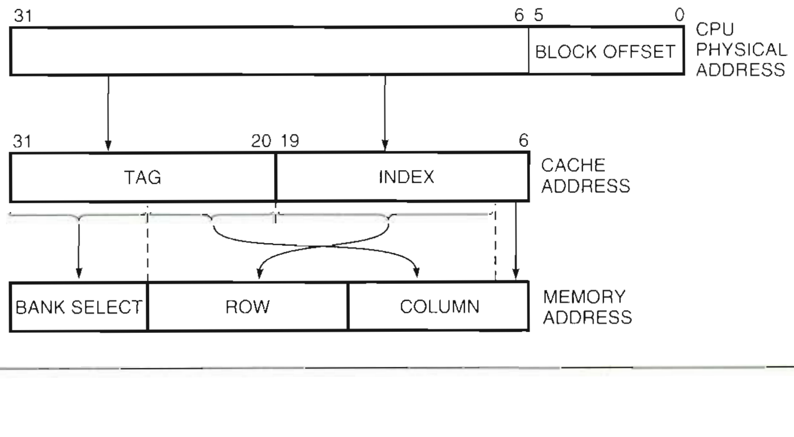


Figure 4
Memory Address Scheme

by 1 MB in memory. These locations fall outside the DRAM page-mode region, resulting in lower performance. The solution is to enlarge the memory blocks; for example, start the memory interchange at bit <8> instead of bit <6>. This compromise allows 256-byte DMA bursts to run at full speed. Slightly fewer victim blocks, however, gain the benefit of page mode.

The bit assignment for this address scheme depends on the row and column structure of the DRAM part and on the external cache size. Power-on software automatically configures the address-interchange hardware in the system.

Design Considerations

In this section, we discuss the design choices made for system clocking, timing verification, and the application-specific integrated circuit (ASIC) design.

System Clocking

The chip set is a synchronous design: The system clock is an integer multiple of the CPU cycle time. Consequently, the PCI clock, the memory clock, and the cache loop are all synchronous to each other. The designers avoided an asynchronous design for two reasons. It suffers from longer latencies due to the synchronizers, and it is more difficult to verify its timing.

Unlike the memory controller, which uses a double-frequency clock to provide a finer 15-ns resolution for the memory timing pulses, the synchronous design of the chip set uses a single-phase clock. This simplified clocking scheme eased the timing verification work. Phase-locked-loop (PLL) devices control the clock skew on the system board and in the ASICs. The PLL in the ASICs also generates the double-frequency clock.

Timing Verification

The complete system was verified for static timing. A signal-integrity tool similar to SPICE was used to analyze all the module etch and to feed the delays into the module timing verification effort. The final ASIC timing verification used the actual ASIC etch delays. This process was so successful that the actual hardware was free of any timing-related bug or signal-integrity problem.

ASIC Design

The chip designers chose to implement the gate array using the 300K technology from LSI Logic Corporation. The control chip uses over 100K gates, and each data slice consumes 24K gates. Originally, the designers considered the slower 100K technology, but it proved unable to satisfy the timing requirements for a 64-bit-wide PCI bus.

The designers used the VERILOG hardware description language to define all the logic within the ASICs. Schematics were not used. The SYNOPSIS gate-synthesizer tool generated the gates. The designers had to partition the logic into small 3,000 to 8,000 gate segments to allow SYNOPSIS to complete within 12 to 15 hours on a DECstation 5000 workstation. Currently, the same synthesis requires 1 hour on the AlphaStation 600 5/260. The designers developed a custom program that helped balance the timing constraints across these small gate segments. This allowed the SYNOPSIS tool to focus its attention on the segments with the greatest potential for improvement.

Performance

Table 1 gives the bandwidths of the workstation for the 32-bit and 64-bit PCI options. A structural simulation model verified this data, using a 180-ns memory latency and a 30-ns system clock. The 285-MB/s read bandwidth of the CPU memory is impressive considering that the memory system is 1 GB. Eventually, the memory size will reach 4 GB when 64-Mb memory chips become available.

The I/O write bandwidth is important for certain 3D graphics options that rely on PIO to fill the command queue. Current high-end graphics devices require approximately 80 MB/s to 100 MB/s. The 213 MB/s of I/O write bandwidth on the 64-bit PCI can support a double-headed 3D graphics configuration without saturating the PCI bus. Other 3D graphics options use large DMA reads to fill their command queue. This approach offers additional bandwidth at 263 MB/s. The system did not optimize DMA writes to the same extent as DMA reads. Most options are amply satisfied with 100 MB/s of bandwidth.

Table 1
Bandwidth Data

Transaction Type	32-bit PCI	64-bit PCI
CPU memory read:		
64 bytes	284	284
I/O write:		
Contiguous 32 bytes	119	213
Random 4 bytes	44	44
I/O read:		
4 bytes	12	12
32 bytes	56	56
DMA read:		
64 bytes	79	112
8 KB	132	263
DMA write:		
64 bytes	97	102

Table 2 gives the performance for several benchmarks. The data is for a system with a 300-MHz processor and a 4-MB cache built out of 12-ns SRAM parts. The SPECmark data is preliminary and clearly world-class. The LINPACK data is for double-precision operands. Even greater performance is possible with faster cache options (for instance, a cache using 8-ns parts) and faster speed variants of the Alpha 21164 microprocessor.

Functional Verification

The functional verification is an ongoing effort. Three factors contribute to the need for greater, more efficient verification. First, the design complexity of each new project increases with the quest for more performance. Next, the quality expectations are rising—the prototype hardware must boot an operating system with no hardware problems. Finally, time to market is decreasing, providing less time for functional verification.

A number of projects at Digital have successfully used the SEGUE high-level language for functional verification.[3,7] SEGUE allows simple handling of randomness and percentage weightings. As an example, a code sequence may express that 30 percent of the DMA tests should target the scatter/gather TLB, and that the DMA length should be selected at random from a specified range. Each evocation of SEGUE generates a test sequence with different random variations. These test sequences are run across many workstations to achieve a high throughput. The project used 20 workstations for 12 months.

The test suite focused on the ASIC verification in the context of the complete system. It was not a goal to verify the Alpha 21164 microprocessor; neither was the EISA logic verified (this logic was copied from other projects). The test environment used the VERILOG simulator and included the Alpha 21164 behavioral model, a PCI transactor (a bus functional model), and a memory and cache model. The SEGUE code generated C-language test programs for CPU-to-memory and CPU-to-I/O transactions, as well as DMA scripts for the PCI transactor.

The goal of verification went beyond ensuring that the prototype hardware functioned correctly. The major objective was to ensure that the hardware is reliable many years hence, when new, as yet undeveloped, PCI options populate the system. Today, the PCI bus uses only a small number of expansion option cards. It is quite probable that a perfunctory verification of the PCI logic would result in a working system at the time of hardware power-on and for many months thereafter. It is only as more option cards become available that the likelihood of system failure grows. Consequently, the verification team developed a detailed PCI transactor and subjected the PCI interface in the control chip to heavy stressors. The complexity of the PCI transactor far exceeds that of the PCI interface logic within the ASIC. The reason is that the ASIC design implements only the subset of the PCI architecture appropriate to its design. The PCI transactor, however, has to emulate any possible PCI device and thus must implement all possible cases. Furthermore, it must model poorly designed PCI option cards (the word "should" is common in the PCI specification).

The verification experience included the following:

- Directed tests. Specific, directed tests are needed to supplement pseudorandom testing. For example, a certain intricate sequence of events is best verified with a specific test, rather than relying on the random process to generate the sequence by chance.
- Staff hours. In prior projects, the hardware team exceeded the verification team in size. Over the years, the importance of verification has grown. On this project, twice as much time was spent on the verification effort as on the hardware coding.
- Degree of randomness. Pure randomness is not always desirable. For instance, an interesting test can be conducted when a DMA write and a CPU read target the same block in memory (although, for coherency reasons, not the same data). Random addresses are unlikely to create this interaction; instead careful address selection is necessary.
- Error tests. The pseudorandom test process added a different error condition, such as a PCI addressing error, within each test. The hardware logic, upon detecting the error, would vector by sending an interrupt to the error-handling code. The handler would check if the hardware had captured the correct error status and, if it had, would resume the execution of the test program. This strategy uncovered bugs when the hardware continued functioning after an error condition, only to fail many cycles later.
- Hardware simulation accelerator. The project team did not use a hardware simulation accelerator for a number of reasons. In the early phase of

Table 2
Benchmark Performance

Benchmark	Performance
SPECint92	331
SPECfp92	503
LINPACK 100 X 100	144
LINPACK 1000 X 1000	380

verification, bugs are so frequent that there is no value in finding more bugs. The limiting resource is the isolation and fixing of the bugs. Second, porting the code onto the hardware simulator uses resources that are better spent improving the test suite: running poor tests faster is of no value. Finally, the hardware-based verification technique offers far greater performance.

- Bug curve. The project team maintained a bug curve. The first-pass ASIC was released when the bug curve was falling but was still above zero. The tests were structured to test the important functionality first. This allowed verification to continue while the operating system developers debugged their code on the prototype. To help this strategy, any performance-enhancement logic in the ASICs could be disabled in case an error was discovered in that logic. Experience on prior projects had shown that such logic has a predilection toward bugs.

Hardware-based Verification

The hardware-based verification was developed to achieve a significant, five-orders-of-magnitude improvement in test throughput. The CPU performs pseudorandom memory and I/O-space transactions, and a number of PCI graphics options emulate generic PCI devices. The hardware-based verification has so far uncovered three bugs. To further improve this technique, a hardware PCI demon is under development. This device has the capability to mimic any PCI device.

The random nature of the test suite means that the bug curve has a long tail: The probability of finding the next bug decreases as each bug is discovered. For example, an earlier project team discovered the last bug after six months but needed only one week to find the penultimate bug. Greater test throughput helps uncover the final bug(s) sooner. Our project team achieved greater throughput by migrating the test strategy onto the actual hardware.

A self-checking, pseudorandom, test-generating program runs on the hardware, testing the memory, the cache, and the PCI. On detecting a mismatch, the software triggers a digital analyzer connected to visibility points on the hardware. Currently, a number of PCI graphics cards are emulating different DMA devices. Eventually, a custom PCI test device, the PCI demon, will replace the graphics cards and provide greater flexibility and functionality (especially with regard to error cases).

The software-based verification, running across 20 workstations, averaged approximately 100 DMA transactions per minute (with concurrent memory and PIO activity). The hardware-based verification runs 60 million comparable DMA transactions per minute per workstation. This 5-orders-of-magnitude improvement suggests that all the tests performed in the past 12 months of software-based verification can be completed during the hardware-based debugging in 5 minutes.

A secondary, but very useful, advantage of hardware-based testing is the ability to stress the chips electrically. For instance, by selecting a data pattern of 1's and 0's for the DMA, memory, and PIO tests, verification engineers can severely test the capability of the chips to switch simultaneously.

Hardware Test Strategy

The SEGUE software proved not to be useful for the hardware-based verification effort. Instead new software was written in the C language for the following reasons:

- Verification must have full control of the hardware and thus cannot run on top of an operating system. Consequently, SEGUE and the operating system functionality are not available.
- Unlike the software environment, visibility into the logic signals is restricted in the hardware environment. The test software has to be written to make debugging simpler.
- One possible strategy is to download the SEGUE tests onto the hardware and thus treat the hardware as a simulation accelerator. However, the resultant performance improvement is small: The SEGUE code takes 2 minutes to generate a 1-hour software-simulation run. These tests run across 20 workstations with a resultant throughput of 1 test every 3 minutes. Assuming the same test executed in zero time on the hardware, the total test time would equal 1 test every 2 minutes—a minor improvement.

The hardware-based verification software relies on the following rationale: The hardware is almost totally bug free, and any remaining bugs are most likely to be due to a rare interaction of events. Indeed, one of the bugs discovered was a special-case DMA prefetch coinciding with a memory refresh. Consequently, no test is likely to detect more than one bug. For instance, if a DMA operation suffers an error, then it is unlikely that a subsequent, identical DMA operation will suffer an error. The second DMA will experience a different set of interactions inside the chip set.

The adopted test environment has two graphics cards, each performing identical DMA operations to two different regions of memory. Because of the serial nature of the PCI bus, however, these cards will perform the DMA operations at different times. Furthermore, other traffic on the PCI bus (for instance, the CPU will be executing random PIO) will further randomize the cards' behavior. While the

DMA transactions run, self-checking, random CPU traffic to memory and I/O will also run. These events provide the random mix of interacting instructions. At the completion of the test, a miscomparsion of the two DMA write regions indicates an error.

Graphics Demon

A number of PCI option cards were investigated as potential PCI demon cards. The requirements for a PCI demon card are twofold: it must be able to perform DMA of various lengths, and it must have memory for the storage of DMA and PIO data. The DEC ZLXp_E1 graphics card was selected because it offers the following advantages:

- Independent DMA. Most PCI options start a DMA operation instantly after the CPU has written to a specific register in the option. This is undesirable because it makes it impossible to emulate options that start DMA operations autonomously (e.g., a network card). To break this coupling, the test program should first make the graphics card paint a portion of the screen. While the graphics device is busy, the graphics command FIFO buffer is filled with the DMA commands. The graphics device will not start the DMA until it has finished painting. Furthermore, the delay is programmable by varying the number of pixels painted.
- Programmable DMA. The graphics card allows the DMA to be any size, whereas most PCI options are constrained to a fixed length. Moreover, it is possible to arrange for PCI disconnects on a DMA read. The graphics card modifies incoming data with the contents of the frame buffer (e.g., frame buffer = frame buffer XOR data). This feature throttles the internal bandwidth of the graphics card, which disconnects it from the PCI.
- Frame buffer. The graphics frame buffer is the target of the DMA and PIO operations. A useful software debugging feature was to observe the frame buffer while running the tests.

PCI Demon

The PCI demon is designed to mimic any possible PCI device. Software has total control of the behavior of the device, including the assertion of error conditions (e.g., parity errors on any specified data word). The architecture of the PCI demon is very simple so that the debugging of the PCI demon is straightforward. (The objective is to find bugs in the chip set and not in the PCI demon.) Consequently, the complexity in using the PCI demon is completely in the software.

The ideal architecture of a PCI demon is a large memory whose output drives the PCI data and control signals directly; the software programs the desired PCI operation by loading the appropriate pattern into this memory. In reality, the architecture of the PCI demon has to diverge from this ideal model for at least two reasons. First, the PCI demon has to be able to emulate the fastest possible PCI device, and this forces the use of an ASIC. However, ASICs have limited memory capacity. It is desirable to store the scripts for many thousands of DMAs in this memory. The scripts are approximately 100-bits wide (64-bit PCI data and control) and require several megabytes of memory. This memory requirement forces the design to use external memory. Second, the PCI architecture has a few handshake control signals that require the use of a fast state machine.

The PCI demon has the functionality to act as a histogram unit (a PCI event counter). Internal counters measure timing information such as DMA latency and the frequency of specified PCI transactions. The PCI demon observes these states by snooping the PCI bus.

Summary

The AlphaStation 600 5-series workstation offers high compute performance, together with substantial I/O subsystem performance. The project team designed a low-cost, 1-GB memory system with a 180-ns memory latency. Timing verification and placement of the plug-in, external cache resulted in a workstation with considerable flexibility in memory expansion, cache variants, and I/O option slots.

The most time-consuming portion of the project was the functional verification. To date, different test programs have run concurrently across 20 high-performance workstations, day and night, for over 12 months. The release of the prototype chip set occurred after 5 months of verification; this chip set successfully booted the operating system. The remaining 7 months of verification were focused on the lower priority functionality (e.g., error cases and slow memory configurations).

The hardware-based verification approach proved its value by uncovering three bugs. The most significant bug involved the interaction of a number of events, including an optimized, prefetching DMA read and a memory refresh. The verification process helped create a very high quality product.

Acknowledgments

Many individuals contributed to the success of this project. The design was a team effort involving far more people in many diverse groups than can be acknowledged here. However, recognition is due to the core hardware and verification team: Ed Arthur, Connie Bielawski, Ernie Crocker, Tracey Gustafson, J. Grady, John Hackenberg, Rick Hagen, Randy Hinrichs, Laura Mendyke, Sudhin Mishra, Sandy

McPherson, Jim Nietupski, Sub Pal, Nick Paluzzi, Rick Rudman, Jim Reilley, Manoo Siarkowski, Bob Stewart, Hugh Kurth, Tony Camuso, Jim Hamel, Rick Calcagni, Carl Mower, Peter Spacek, and Ned Utzig.

References

1. J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," *Digital Technical Journal*, vol. 7, no. 1 (1995, this issue): 119–135.
2. *2420/82430 PCI ISA and EISA Bridges* (Santa Clara, Calif.: Intel Corporation, 1993).
3. S. Nadkarni et al., "Development of Digital's PCI Chip Sets and Evaluation Kit for the DECchip 21064 Microprocessor," *Digital Technical Journal*, vol. 6, no. 2 (Spring 1994): 49–61.
4. *NCR 53C810 Data Manual* (Dayton, Ohio: NCR Corporation, 1992).
5. A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming* (Boston: Kluwer Academic Publishers, 1989).
6. S. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM Journal of Research and Development*, vol. 38, no. 5 (September 1994).
7. W. Anderson, "Logical Verification of the NVAX CPU Chip Design," *Digital Technical Journal*, vol. 4, no. 3 (Summer 1992): 38–46.

Biographies

John H. Zurawski
John Zurawski was the system architect for the AlphaStation 600 5-series workstation. Prior to this project, John was the system architect for the DECstation 5000 series of MIPS R4000 workstations. He has also led the verification effort for the DEC 3000 workstation and led the team that designed the floating-point unit for the VAX 8800 family. John holds a B.Sc. degree in physics (1976), and M.Sc. (1977) and Ph.D. (1980) degrees in computer science, all from Manchester University. John is a member of IEEE. He holds seven patents and has published six papers on computer technology. He joined Digital in 1982 after completing post-doctoral research at Manchester University.

John E. Murray
A consulting engineer in the Alpha Personal Systems Group, John Murray was the logic design architect for the AlphaStation 600 5-series. In previous work, John led the design team for the instruction fetch and decode unit on the VAX 9000 system. Prior to joining Digital in 1982, he was with ICL in the United Kingdom. He holds eleven patents.

Paul J. Lemmon
Paul Lemmon joined Digital in 1987; he is a principal engineer. Paul was the ASIC team leader and the architect of the control ASIC for the AlphaStation 600 5-series. He was previously employed at Datapoint, where he was a design engineer/project engineer. Paul received a B.S. in electrical engineering from Ohio State University in 1980. He holds two patents.

Circuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU

A 300-MHz, custom 64-bit VLSI, second-generation Alpha CPU chip has been developed. The chip was designed in a 0.5-μm CMOS technology using four levels of metal. The die size is 16.5 mm by 18.1 mm, contains 9.3 million transistors, operates at 3.3 V, and supports 3.3-V/5.0-V interfaces. Power dissipation is 50 W. It contains an 8-KB instruction cache; an 8-KB data cache; and a 96-KB unified second-level cache. The chip can issue four instructions per cycle and delivers 1,200 mips/600 MFLOPS (peak). Several noteworthy circuit and implementation techniques were used to attain the target operating frequency.

William J. Bowhill, Shane L. Bell, Bradley J. Benschneider, Andrew J. Black, Sharon M. Britton, Ruben W. Castelino, Dale R. Donchin, John H. Edmondson, Harry R. Fair, III, Paul E. Gronowski, Anil K. Jain, Patricia L. Kroesen, Marc E. Lamere, Bruce J. Loughlin, Shekhar Mehta, Robert O. Mueller, Ronald P. Preston, Sribalan Santhanam, Timothy A. Shedd, Michael J. Smith, Stephen C. Thierauf

The Alpha 21164 chip is a 300-megahertz (MHz), quad-issue, custom very large-scale integration (VLSI) implementation of the Alpha architecture that delivers peak performance of 1,200 million instructions per second (mips)/600 million floating-point operations per second (MFLOPS). The chip is designed in a 0.5-micrometer (μm) complementary metal-oxide semiconductor (CMOS) technology using four levels of metal. The die measures 16.5 millimeters (mm) by 18.1 mm and contains 9.3 million transistors. It operates at 3.3 volts (V) and supports 3.3-V and 5.0-V interfaces. The chip dissipates 50 watts (W) at 300 MHz (internal clock frequency). Switching noise on the power supplies is controlled by an on-chip distributed coupling capacitance between power and ground of 160 nanofarads (nF). The chip contains an 8-kilobyte (KB), first-level (L1) instruction cache; an 8-KB L1 data cache; and a 96-KB second-level (L2) unified data and instruction cache.

This paper focuses on the circuit implementation of the Alpha 21164 CPU. Space does not permit a description of the complete design process utilized throughout the project. Instead, some of the significant circuit design challenges encountered during the project are discussed. The paper begins with an introductory overview of the chip microarchitecture. It continues with a description of the floorplan and the physical layout of the chip. The next section discusses the clock distribution and latch design. This is followed by an overview of the circuit design strategy and some specific circuit design examples. The paper concludes with information about design (physical and electrical) verification and CAD tools.

Microarchitecture Overview

The Alpha 21164 chip is a completely new implementation of the Alpha architecture. Figure 1 shows a block diagram of the Alpha 21164 chip. The microprocessor consists of five functional units: the instruction fetch, decode, and branch unit (I-box); the integer execution unit (E-box); the memory management unit (M-box); the cache control and bus interface unit (C-box); and the floating point unit (F-box).

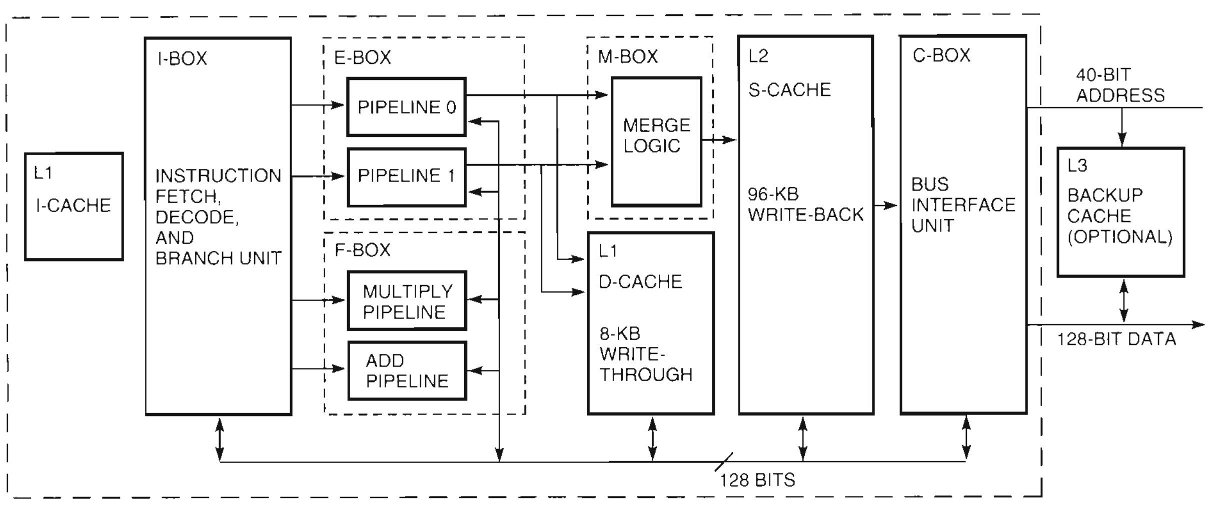


Figure 1
Block Diagram of the Alpha 21164 Functional Units

The chip contains three on-chip caches: the instruction cache (I-cache); the data cache (D-cache); and the second-level (unified data and instruction) cache (S-cache). The microprocessor uses a seven-stage pipeline for integer and memory operations and a nine-stage pipeline for floating-point operations. Pipeline stages are referred to as S0 to S8 for the remainder of this paper.

The I-box fetches instructions from the 8-KB virtual, direct-mapped, physical L1 I-cache. It decodes and issues instructions to the E-box, M-box, and F-box. It maintains state for all pipeline stages to track outstanding register writes. The I-box can decode and issue up to four instructions per cycle. The E-box contains two 64-bit pipelines. The first pipeline contains an adder and a Boolean logic unit. The second pipeline contains an adder, a Boolean logic unit, a shifter, and a multiplier. Most integer instructions execute in one cycle. The F-box contains a floating-point multiply pipeline and a floating-point add pipeline. The M-box executes all load and store instructions. The 8-KB, direct-mapped, write-through, L1 D-cache has two read ports, which allow two load instructions to execute in parallel. The C-box processes memory accesses sent by the M-box and manages all cache coherence protocol functions. It also controls the L2 unified cache and an optional external (off-chip) backup cache. A detailed discussion of the chip microarchitecture is presented in another paper in this issue.[1]

Floorplan Overview

With 9.3 million transistors and a die size very close to the manufacturing limit, floorplanning of the Alpha 21164 chip was a critical activity during development. Power and ground distribution, as well as clock routing and signal busing, were also major factors in developing a viable floorplan. The floorplan was developed early in the project and was regularly monitored and kept up-to-date. Chip area estimates were made using several different methods. The area estimates of the caches were determined by creating trial layout structures. Major sections of the integer and floating-point execution units were estimated using data from previous designs and trial layout. The control logic areas were estimated using data from logic and layout synthesis CAD tools.

Figure 2 is a photomicrograph of the chip with overlays showing the major functional units and pin locations. The major functional units were positioned to minimize critical signal propagation delays and the length of global buses. The I-box, E-box, and M-box were aligned to allow the routing of the data between these sections in a common data path. Since the C-box interfaces with most of the other functional units, it was situated along the bottom of the chip to accommodate different signal routing needs. The address pins for the memory interface were placed at the bottom of the chip to keep them in close proximity to the C-box and M-box, thereby reducing routing and latency. Due to the significant data routing between the S-cache, C-box, and pins, the placement of these sections was critical. Both the 128-bit-wide S-cache data array and the 128-bit-wide C-box data path were split into two halves, each containing the upper and lower 64 bits. This minimized the routing between the C-box, the S-cache, and the data pins along the two sides of the chip.

The D-cache and the I-cache were arranged to allow 128-bit-wide data access to the S-cache. The I-cache and D-cache were placed above and below the S-cache, respectively, and centered within the chip as

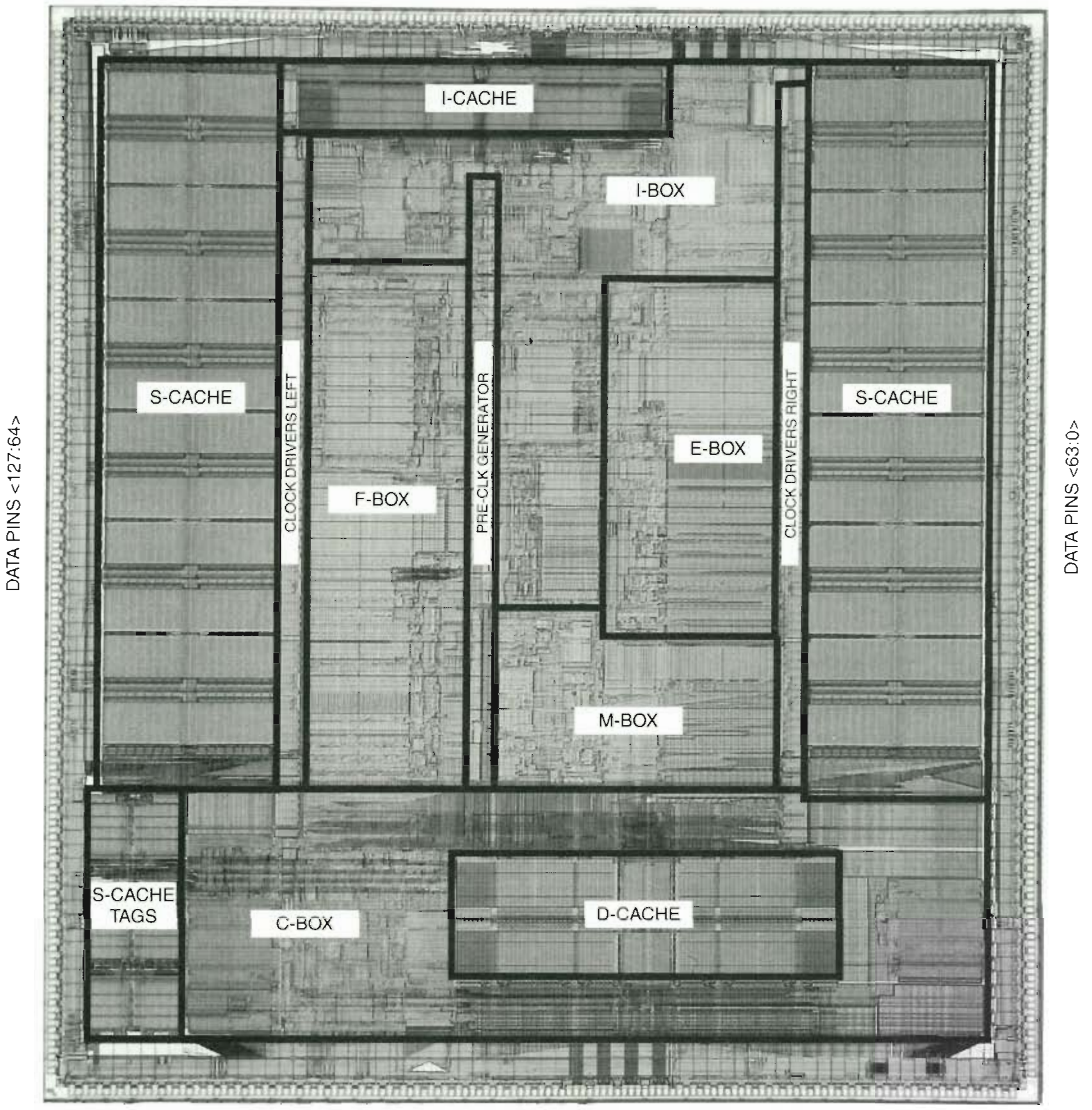


Figure 2
Floorplan of the Alpha 21164 Chip

much as possible. The S-cache was implemented with metal 4 read and write buses spanning the entire height. This provided the access needed to both the top and bottom of the S-cache for routing to the I-cache and D-cache.

The D-cache supports two loads per cycle, requiring a dual-ported read design. The D-cache was implemented as two single-ported caches containing identical data instead of one dual-ported cache. The major consideration that led to this decision was the ability to share the single-ported design with the I-cache. Sharing the design also reduced the overall analysis and verification required.

Interconnect routing was another important part of the floorplanning process. Four metal layers were available for routing. The lower metal layers, metal 1 and metal 2, were used for local transistor hookup and signal routing. The upper metal layers, metal 3 and metal 4, were used primarily for clock, power, and ground distribution. When necessary, the upper metal layers were also used to route critical signals and long buses. The metal orientations were chosen to accommodate both the cache structures and the data paths of the functional units. With reference to Figure 2, metal 2 and metal 4 lines were arranged to run vertically and metal 1 and metal 3 were arranged to run horizontally. Most of the global routing was done by hand. Local cell routing was done by hand with some assistance from auto-routing CAD tools.

The upper metal layers are organized as a fine-pitched regular grid structure placed over the entire chip. The typical drawn line width used in this grid is 12 μm. Power and ground lines are alternated with a single clock line interspersed every few pairs. A limited number of critical signals and buses are also routed in metal 3 and metal 4. In the pad ring, metal 4 is used to route power and ground into the chip from the pads, and input and output signal wires are routed circumferentially in metal 3.

The fine pitch of the power grid allowed its placement and hookup to be deferred until most of the cell layout was complete. Therefore, individual cell layout was done independently of the main grid. The local power, ground, and clock lines in cells were connected to vertically routed metal 2 lines. These metal 2 lines were designed to be long enough to span two or more of the metal 3 grid lines, which allowed the eventual connection to the grid.

Cell hookup to the grid was automatically generated using an in-house CAD tool that accepted nodal layout extracts of the power, ground, and clock nodes. It connected the grids by placing a maximum number of contacts between metal 4 and metal 3, and metal 3 and metal 2. All contacts placed by the tool met the process design rules.

Clock Distribution Scheme and Latch Design

The Alpha 21164 chip enhanced the clock distribution mechanism and verification techniques that were developed during the design of the first-generation Alpha microprocessor chip, the DECchip 21064.[2] A global, single-wire clock (CLK) is distributed over the chip. The design quality of the clock signal was critical to meeting the fast cycle time goal of the Alpha 21164 chip. (Note that the high state of the clock signal is identified as the A-phase, and the low state of the clock signal is considered the B-phase.)

Figure 3 is a circuit schematic of the clock generation circuitry. The chip receives a 2× differential emitter-coupled logic (ECL) oscillator clock signal (up to 600 MHz) that is divided by two and then routed to the center of the chip. It is then buffered up through 6 inverter levels and fanned-out to 24 inverters through a balanced tree to drive the PRE_CLK signal in metal 3 to the right and left banks of the final stage clock drivers. These final buffers are located between the second-level caches and the outside edge of the execution core. The final clock driver has 4 more levels of inverter buffering before it drives the single-wire CLK signal that is formed using a metal 3 and metal 4 grid. Figure 4 is a schematic showing the placement of the CLK grid.

There are 44 final clock drivers per side. The placement of these drivers was based on local clock loading. There are also 12 conditional clock drivers per side, with each activating a single S-cache subarray. Each clock buffer is surrounded by decoupling capacitance to reduce switching noise. The last clock driver inverter has a total gate width of 58 centimeters (cm). It drives a load of 3.75 nF of gate and interconnect capacitance. The clock distribution system consumes about 20 W, approximately 40 percent of the total chip power.

The two banks of final clock drivers, coupled with the size of the clock network, complicated the clock skew analysis. The precise method of calculating clock skew requires a SPICE model of the entire network from input oscillator signal through the drivers and the entire metal grid.[3] This approach was not possible due to the size of the network. Instead, the three main components of clock skew were quantified: (1) PRE_CLK driver and PRE_CLK RC delay variations, (2) final CLK driver transistor delay variations, and (3) RC delay through the metal grid. A SPICE model of the PRE_CLK drivers and network was simulated to identify the differences in arrival time of the PRE_CLK signal at the input to the final drivers.[3] The simulations showed that any delay variations developed in this portion were equalized by strapping the inputs and outputs of the final driver inverters. The net result was that the PRE_CLK network contributes only 10 picoseconds (ps) to the total skew.

The next source of variability affecting clock skew was the difference in transistor characteristics among the final stage drivers. There are actually two causes of this variation: layout-related effects and systemic intra-die process variations. Although the layout-related effects can be controlled in the design, the process variations cannot. To limit channel length variations that result from layout differences, a joint effort was undertaken with the process development group to define a modular block for the driver layout. This block was then repeated as many times as necessary to achieve the required total driver size while keeping the polysilicon spacing, density, and orientation constant.

Since it was not possible to model the metal 3 and metal 4 grid with the drivers in SPICE, extensive RC delay simulation of the clock grid alone was done based on resistance and capacitance values extracted from layout.[3] The drive impedance of each clock driver was modeled as an equivalent resistance in the network. The skew between the first and last latch receiving the CLK signal is 90 ps. The instruction and execution units all see the CLK signal within 65 ps, which is well within the design goal. Figure 5 shows the clock RC delay as a function of the *X* and *Y* locations on the chip.

The development of standard latches was an important aspect of the Alpha 21164 implementation process. The primary goal of the latch design was to produce a fast circuit that would use the device area efficiently and that could be used in a wide variety of instances. To minimize the chip verification effort, a standard latch library was developed early in the design process. This library set the standards for latch

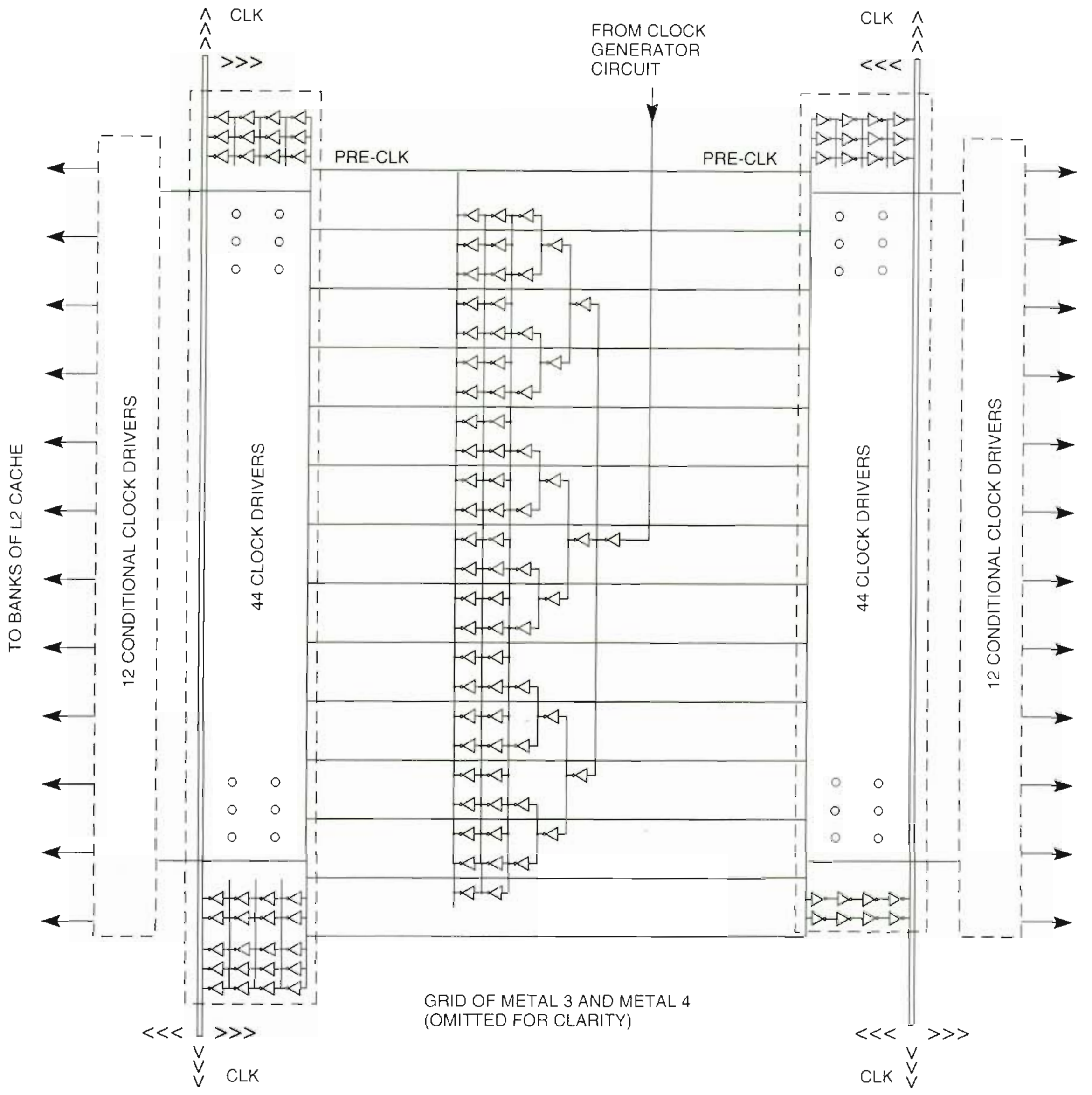

CLK
FROM CLOCK GENERATOR CIRCUIT
CLK
PRE-CLK
PRE-CLK
TO BANKS OF L2 CACHE
12 CONDITIONAL CLOCK DRIVERS
44 CLOCK DRIVERS
44 CLOCK DRIVERS
12 CONDITIONAL CLOCK DRIVERS
TO BANKS OF L2 CACHE
GRID OF METAL 3 AND METAL 4 (OMITTED FOR CLARITY)
CLK
CLK

Figure 3
Schematic of Clock Distribution System

usage and allowed designers to utilize latches that had already been verified over a range of operating conditions and process corners.

The Alpha 21164 chip uses level-sensitive, transmission gate latches as shown in Figure 6. Two basic types of latches were developed: A-latches (Figures 6a and 6c) and B-latches (Figures 6b and 6d). The A-latches are open when CLK is high, and the B-latches are open when CLK is low. The latch input inverter can be replaced by a logic gate (shown in Figures 6c and 6d), thus reducing gate delays in other logic. This style of latch is very fast and area-efficient, yet it does have an inherent race-through problem. It was estimated that the use of this latch style yields a 10 percent improvement in speed over the 21064 microprocessor.

The additional skew in the clock, resulting from the local clock buffer delay, increases the possibility that data could race through a pair of latches during the transition of the clock. Although the overall skew of the internal clock is low, this was not considered sufficient to avoid race conditions. Two significant steps were taken to guarantee that no race could exist between latches. First, the buffered clock inside the latch was sized to minimize the additional skew resulting from its delay. Second, rules and verification tools were developed to make sure that the design includes

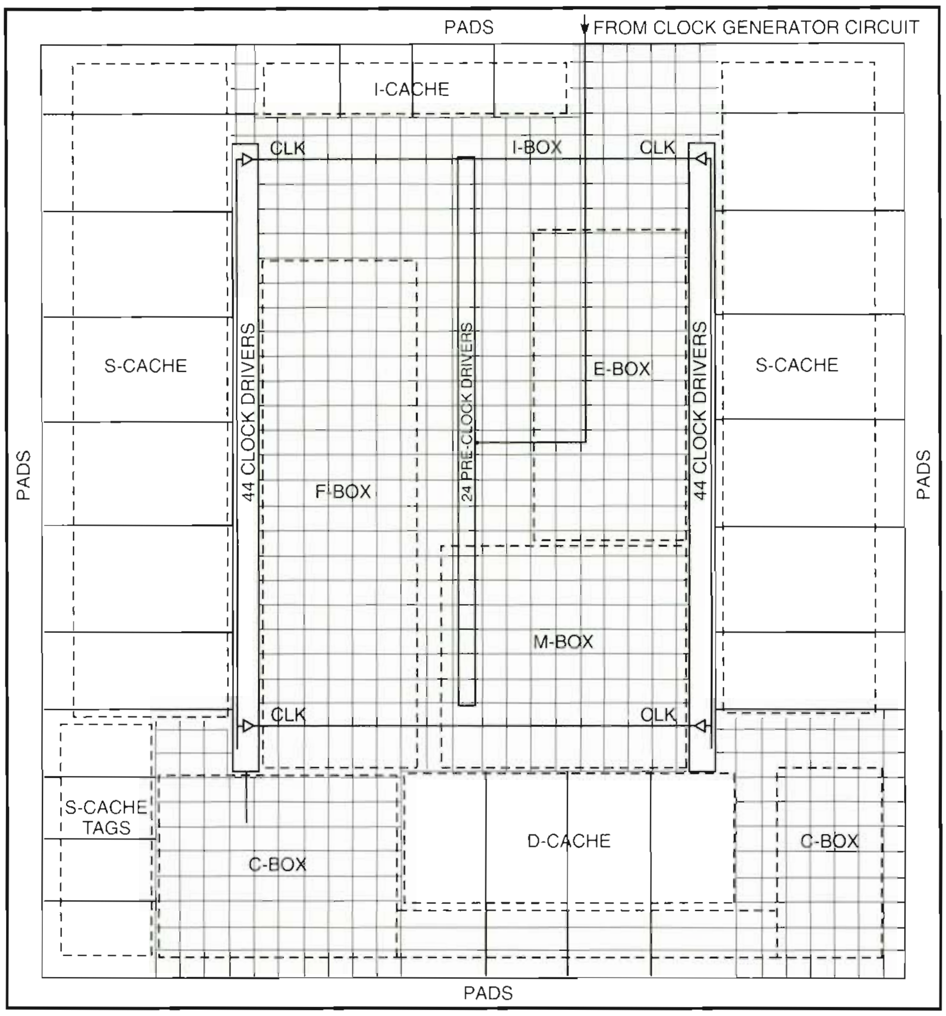


Figure 4
Metal 3 and Metal 4 Clock Grid

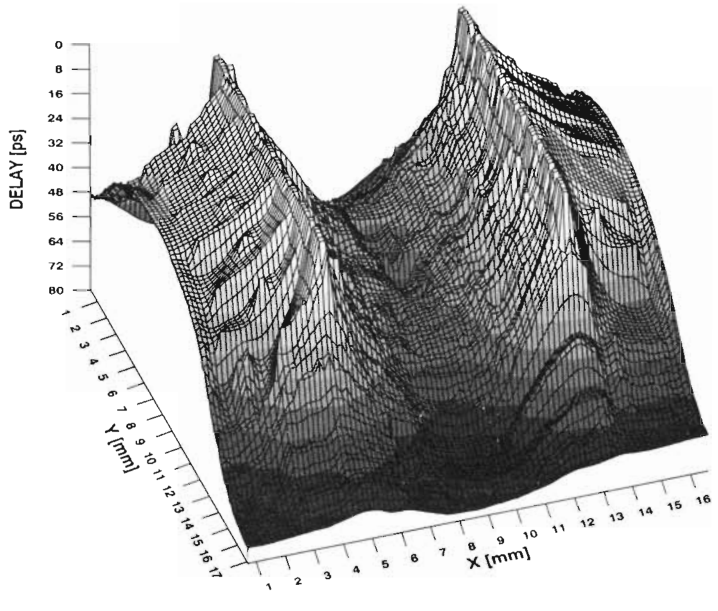


Figure 5
Clock RC Delay

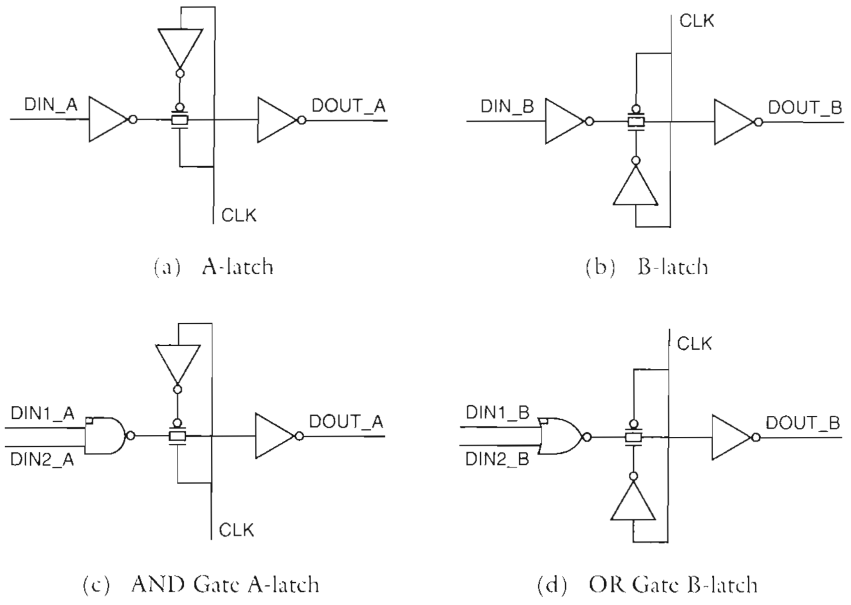


Figure 6
Alpha 21164 Standard Latch Examples

at least one additional gate delay between all latches, thus guaranteeing a race-free design. Designers had the option of designating these gates as logic functions or simple inverters. The delay did not affect critical speed paths, since critical paths tended to have more than one delay between latches.

Circuit Design Strategy

Due to the complexity of the Alpha 21164 chip and the large size of the design team, a comprehensive design methodology was developed. A design guide was created to provide a consistent set of rules and methods for the development of circuit schematics and layout. This document helped ensure that all designers worked under the same design assumptions. In addition, it relieved time-consuming analysis of each circuit by providing guidelines and "rules of thumb" that guaranteed correct operation and minimized the possibility of reliability problems.

Guidelines for common circuit structures such as complementary, cascode, dynamic, and static circuits were created by characterizing their behavior over all process corners. Adequate noise margins were ensured by specifying operating envelopes for such design parameters as device size, stack height, and beta ratio. Reliability guidelines were specified for electromigration, hot carrier effects, and substrate charge injection. Most circuits were designed within the rules specified in the guide; however, a few circuit designs violated the rules. These designs were allowed only when performance and area advantages would be gained. These exceptions were carefully verified for functionality and reliability.

An extensive suite of in-house CAD tools was used to aid and structure the design process. In all cases, the tools supplemented the design process and automated repetitive work. Engineering judgment and iterative use of the software were required to create the final production schematics. Tools that aided schematic generation included a schematic editor, a logic synthesis tool, and a device-sizing tool. Post-schematic tools included a latching methodology checker, a circuit verifier that highlighted design methodology violations, and a timing verifier that analyzed potential critical speed paths. The use of the design tools varied across the chip, based on the degree of customized logic required. For example, the I-box did not rely heavily on the synthesis tools because of the need for optimized circuit structures. However, the C-box used the synthesis tools extensively to produce baseline schematics, which were then modified by hand as necessary.

Circuit Design Examples

The designers of the Alpha 21164 chip were faced with a number of implementation challenges. The most significant challenge was to design a chip that could run at 300 MHz, 50 percent faster than the previous Alpha implementation.[2] Device scaling, process development, and architectural improvements delivered part, but not all, of the required speedup. The additional improvement had to be obtained using circuit design techniques. Other challenges included a much more complicated microarchitecture and the reduction in latency of a number of instructions from the previous implementation. Finally, the large physical size of the chip also led to challenges in circuit design and power management.

The following sections describe several circuit design challenges encountered during the implementation of the Alpha 21164 chip.

I-box Design—Issue Stage Dynamic Dirty/Bypass Logic

The issue stage of the I-box coordinates the release of instructions into the E-box, F-box, and M-box pipelines. The deep pipelines and sophisticated memory management unit along with the high clock frequency presented significant challenges to the implementation team. The Alpha 21164 microarchitecture allows up to 37 instructions to be in progress at the same time (7 integer operates, 9 floating operates, and 21 loads that missed). Superscalar issue of 4 instructions requires that 8 operands and 4 new destinations must be checked against these 37 outstanding instructions in every cycle. In addition, 44 bypass paths are built into the E-box and F-box pipelines in the Alpha 21164 chip. Each of the 8 operands must be checked against several of these bypass paths to ensure that the most up-to-date data is forwarded to the issuing instruction.

The register comparisons were implemented using domino logic. As each instruction is issued, its destination register address is decoded into a 31-bit mask that is entered into a shift register that mimics the appropriate execution pipeline. Checks are performed for stalls and bypasses by selecting the appropriate masks from each level of the shift register and comparing them to the register addresses of the new instructions. Integer and floating-point instructions are handled in separate 31-bit-wide data paths.

Decoding the register addresses allows a logical OR of several destinations to create "dirty" bit masks, greatly reducing the required number of comparators. This reduction in comparators more than compensates for the additional logic involved in carrying the decoded register addresses for all pipe stages (31 bits versus 6 bits for encoded register numbers). With this scheme, all stall calculations are performed using only 38 comparators. Bypass detection is performed in a manner similar to the stall generation using an additional 44 comparators, one for each E-box and F-box bypass path.

The implementation of the comparators requires three domino stages (see Figure 7). The first stage is a two-input dynamic multiplexer that selects the operand/destination decode field for the new instruction or the field of the previous cycle's instruction if a stall was detected. The dirty bit mask is created in a similar dynamic OR structure. The second domino stage is a bit-wise AND function of the operand/destination decode mask and the dirty bit mask followed by a zero detector (logical OR of the 31 bits). A transmission gate forms a second AND function in this stage that qualifies the detected register conflict with an instruction valid signal. The third domino stage is used to further qualify the detected conflict with instruction type decode information and to start a logical OR of the 38 conflict outputs into a single stall wire. In the case of bypasses, the third domino stage is used to priority-encode the bypasses so that only the most up-to-date data is bypassed.

Special attention was given to several circuit design issues when the domino logic was implemented. Careful preplanning of the routing provided large lateral spacing on the dynamic lines to reduce coupling. Noise margins were protected by ensuring that all dynamic inputs were driven from local inverters with a common ground reference. Charge-share problems in the large second domino stage (31-bit-wide AND-OR function) were minimized due to the fact that only a single bit will be set in the new instruction's operand decode bit mask, which is used as the upper input in the 31 AND stacks. Therefore, only a single internal node may charge-share with the large output capacitance.

Another critical concern in such a large dynamic structure was power consumption. The logic was implemented in such a way as to minimize the number of nodes that discharge each phase. To minimize short-circuit currents, the second and third domino stages are precharged by means of matched delay signals. These self-timed precharged lines also help to minimize clock loading since CLK is used to precharge only the first stage.

E-box Design—Bypass Logic

The E-box presented a number of interesting circuit challenges. The Alpha 21164 implementation contains two integer pipelines, as compared to one in the 21064. This significantly increased the circuit design complexity associated with providing result bypassing from all functional units.

The E-box bypass logic is responsible for supplying input operand data to the functional units in both integer pipelines. Input operand data can be supplied from the register file or bypassed from the output of any pipeline stage in the E-box (Figure 8). Functional operations are performed in pipeline stage 4 (S4), and register file writes occur in stage 6 (S6). Without bypass logic, instructions that require data from the pipeline would have to be stalled until the data reaches S6 and is written into the register file. These stalls would impact the integer performance severely. Therefore, the ability to bypass operand data from pipeline stages S4 through S6 was critical to obtaining high integer performance.

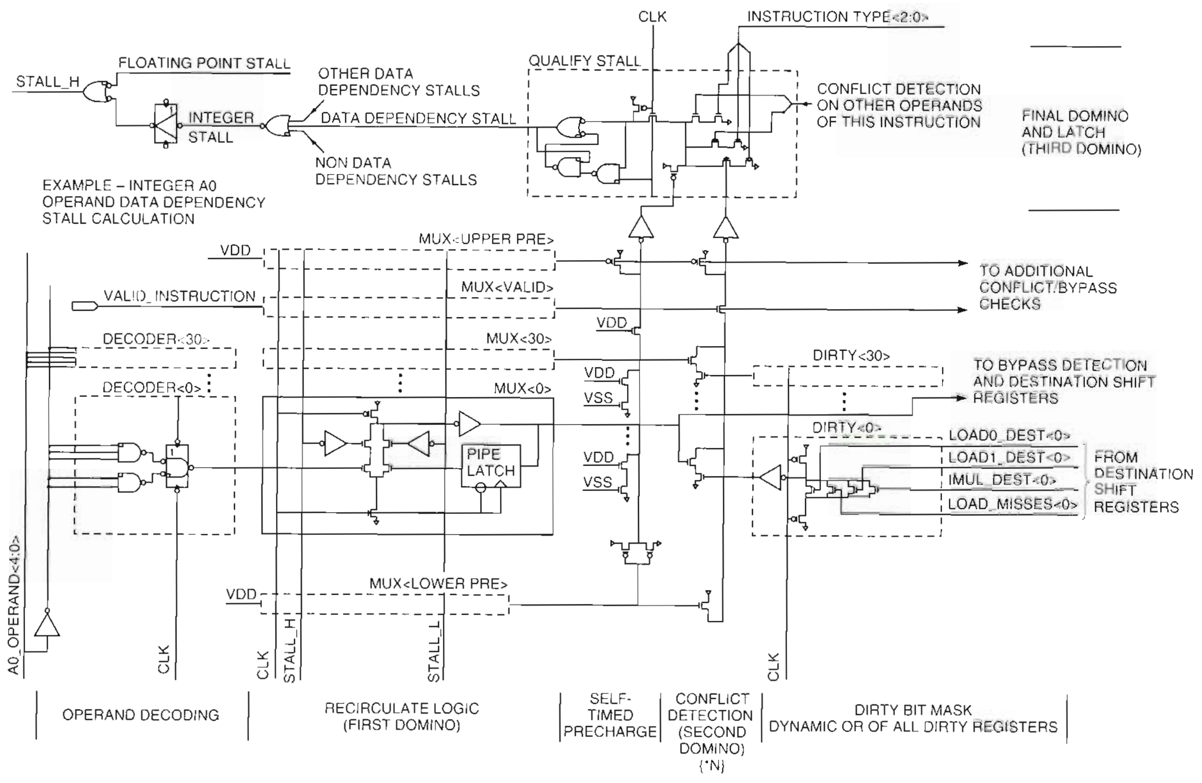

CLK
INSTRUCTION TYPE<2:0>
FLOATING POINT STALL
STALL_H
QUALIFY STALL
OTHER DATA DEPENDENCY STALLS
INTEGER STALL
DATA DEPENDENCY STALL
CONFLICT DETECTION ON OTHER OPERANDS OF THIS INSTRUCTION
FINAL DOMINO AND LATCH (THIRD DOMINO)
NON DATA DEPENDENCY STALLS
EXAMPLE – INTEGER A0 OPERAND DATA DEPENDENCY STALL CALCULATION
VDD
MUX<UPPER PRE>
VALID_INSTRUCTION
MUX<VALID>
TO ADDITIONAL CONFLICT/BYPASS CHECKS
DECODER<30>
MUX<30>
DIRTY<30>
TO BYPASS DETECTION AND DESTINATION SHIFT REGISTERS
DECODER<0>
MUX<0>
VSS
DIRTY<0>
LOAD0_DEST<0>
LOAD1_DEST<0>
IMUL_DEST<0>
LOAD_MISSES<0>
FROM DESTINATION SHIFT REGISTERS
PIPE LATCH
MUX<LOWER PRE>
A0_OPERAND<4:0>
CLK
STALL_H
STALL_L
OPERAND DECODING
RECIRCULATE LOGIC (FIRST DOMINO)
SELF-TIMED PRECHARGE
CONFLICT DETECTION (SECOND DOMINO) (*N)
DIRTY BIT MASK DYNAMIC OR OF ALL DIRTY REGISTERS

Figure 7
Domino Logic for Issue Scoreboard

Four 64-bit dual-rail operand buses are used to bypass data. Two buses in each pipeline are used to supply A and B operand data to the functional units. The buses are controlled by the BYPASS_ENABLE_L signals generated in the I-box and are driven during the B-phase (see Figure 9). A typical operand bus driver is shown as well as the shifter operand bus driver. The shifter driver is unique because it has byte zap (set byte to zero) logic capability.

Data is read from the operand buses during the early portion of the A-phase by operand bus receivers located at the input of each functional unit. The receiver is a dynamic gate structure that can be configured to receive one or more inputs and generate a logical function output. The adder uses the logical function capability to generate propagate and kill signals.

The operand bus is precharged by a delayed A-phase clock. This delay allows the dynamic bus receiver gate to act as a latch and eliminates the need for a true B-latch (see Figure 9). During the beginning of the A-phase, operand data propagates through the receiver and is captured by the receiver gate output latch node before the delayed A-phase clock precharges the operand bus. Once the operand bus is precharged, the latch node is decoupled from the operand bus.

E-box Design—64-bit Shifter

The E-box shifter executes all 64-bit shift, extract, insert, and zap (set to zero) instructions on both little and big endian data types using a 128-bit right-only shifter. All shift instructions take one cycle to execute, an improvement of one cycle relative to the 21064 design.

The data path portion of the shifter logic uses dynamic and cascode circuitry to read the operand buses, to present the data to either the low or high 64 bits of data, and to sign-extend the high 64 bits, when necessary, in the A-phase. In the B-phase, the input data is shifted, a byte zap is performed when necessary, and the result is driven onto the result bus. The result can be bypassed onto an operand bus. Right shifts are performed by loading the A-operand data into the low 64 bits and shifting based on the value of the B-operand; left shifts are performed by loading the A-operand data into the high 64 bits and shifting based on the two's complement of the value of the B-operand. The shifter array is implemented as a differential dynamic gate. The layout uses metal 1 for the input data, metal 2 for the output value, and metal 3 for the shift amount.

The chief improvement in this design over the 21064 design is the single-phase generation of the 65 shift enable signals and byte zap mask. The shift enable

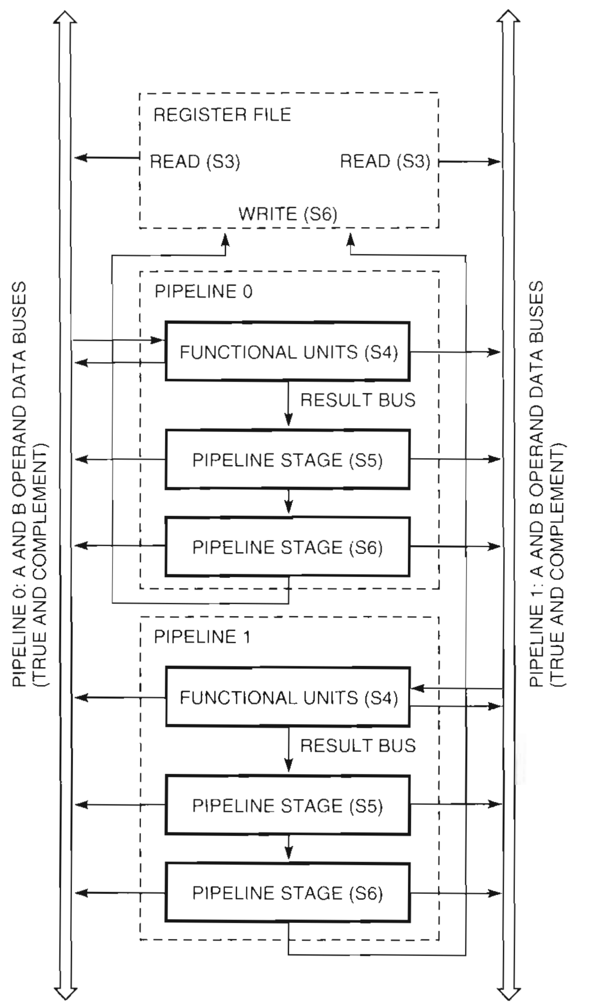


Figure 8
E-box Pipeline

generation is accomplished by combining the shift requirements of the extract and insert instructions with the B-operand decode logic for normal shifts. An 8-bit shifter is used to implement the byte zap mask to achieve the single-phase goal. The 8-bit zap mask shifter is built using differential dynamic logic. Its control resembles that of the 64-bit shifter, employing cascode data input circuitry and dynamic decode logic. The shift amount is determined from the B-operand or bits in the instruction based on the opcode.

Cache Design—Power Savings

Special design considerations were given to the three caches on the Alpha 21164 chip because they comprise, by far, the largest number of devices and have the greatest impact on yield. Since the caches are accessed frequently, the power consumption of the caches was also a cause for concern.

The 8-KB I-cache includes two pairs of fuse-programmable redundant rows to offset any yield loss. The D-cache leverages the I-cache design by combining two of these caches to form a single, dual-read-ported, 8-KB data cache. The D-cache employs the same row redundancy scheme as the I-cache. The Alpha 21164 chip also contains the S-cache, which is a large, second-level cache for both data and instructions. The S-cache data array is organized into 24 banks of 4 KB each. Twelve banks are placed on the left and right sides of the chip. Figure 10 shows the arrangement of the banks on the right side. Each bank of both the tag and data arrays implements row redundancy. The S-cache data array also implements column redundancy.

Pipeline processing of the S-cache allows the inclusion of power-saving features. The S-cache operates in a four-stage pipeline: two stages for tag lookup and modification, and two for data access and transfer. Address decoding during the tag lookup results in the clocking of only 2 of the 8 banks in each of the 3 sets (6 of 24 in the whole cache). The bit lines and sense amplifiers in the disabled 18 banks are frozen in the precharge mode, consuming minimal power.

Hit signals from the tag-lookup logic control the word lines and sense amplifiers of the six enabled banks. Therefore, of the six banks enabled, only the two banks for the set that hit are activated and discharged. This design results in an estimated power savings of 10 W.

System Clock Design—Synchronization

The Alpha 21164 chip is designed to accommodate multiprocessor systems using a synchronous bus. This requires the synchronization of the Alpha 21164 chip's generated reference clock (SYS_CLK) to the systems-generated reference clock (REF_CLK). To achieve the maximum system performance, this must be done with as little error as possible.

In other designs, this synchronization is achieved using an on-chip phase-locked loop (PLL).[4] However, the on-chip noise environment of the Alpha 21164 could cause excessive PLL jitter. Jitter can reduce the width of a clock phase and create a pulse too narrow to clock on-chip logic. This uncertainty would dictate slowing the clock frequency, thus reducing system performance.

The design challenge was to find a low-risk digital solution that would meet the high-frequency performance requirements of the Alpha 21164. To meet this challenge, a state machine PLL (SMPLL) was designed. This all-digital approach has much better noise immunity than a traditional PLL, but it does introduce a quantizing error, or skew, into the system clock timing. This skew can complicate system timing but has minimal impact on CPU performance, since it allows the Alpha 21164 chip to run at the highest possible clock frequency.

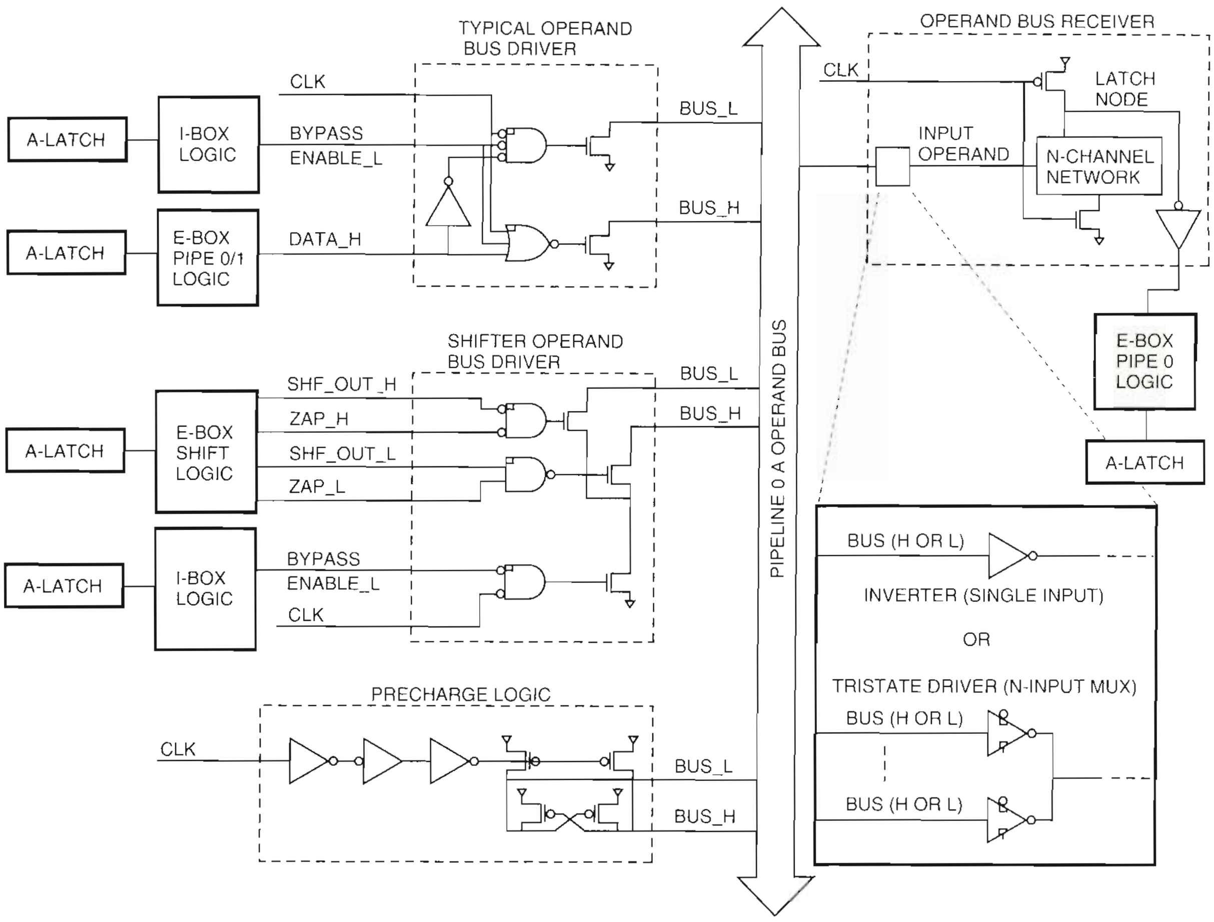


Figure 9
E-box Bypass Bus

Figure 11 shows a functional block diagram of the SMPLL. The Alpha 21164 generates a system bus clock (SYS_CLK) by dividing the internal CLK by a preprogrammed amount. This SYS_CLK is then aligned to the system-generated reference clock (REF_CLK). To do this, the frequency of REF_CLK must be slightly lower than that of SYS_CLK. A phase detector compares the arrival of the rising edge of REF_CLK with the rising edge of SYS_CLK. If the edges are coincident, the SMPLL stretches SYS_CLK by the period of the chip oscillator. Thus, the rising edge of REF_CLK always leads the rising edge of SYS_CLK. However, because SYS_CLK is slightly faster than REF_CLK, the rising edge of SYS_CLK will eventually catch up to REF_CLK. When this happens, the phase detector once again stretches SYS_CLK, and the process of catching up starts anew.

The SMPLL design takes advantage of the on-chip clock divider circuitry by suppressing the divide for a single count whenever a phase alignment is required. This scheme adjusts the phase alignment in increments of 1.67 nanoseconds (ns) (assuming a 600-MHz input clock) and allows the rising edge of the REF_CLK, measured at the input pin of the Alpha 21164, to coincide with the SYS_CLK to within 1.67 ns.

Physical and Electrical Verification

The ability to verify the layout of a 9.3-million-transistor VLSI chip, both physically and electrically, without hampering its performance or impacting its development schedule, was a primary concern from the outset of the project. Many new techniques were developed to accomplish this task. Some of the more significant advances are discussed in the following section.

Physical Layout Verification

The size and complexity of the Alpha 21164 dictated the use of physical assembly methods that did not require the CAD tool suite to verify the complete chip layout database in one pass.

Full-custom designs like the Alpha 21164 chip are composed of large blocks of random logic that are not easily divisible into highly repetitive instantiations of common cells. Because of the relatively few instances of repetitive structures, there was no need to design using a deep cell hierarchy. Similar to previous, large, full-custom designs, the Alpha 21164 floorplan divided the chip along major box boundaries.[5] This partitioning reduced the device count per partition, allowing each to be verified independently.

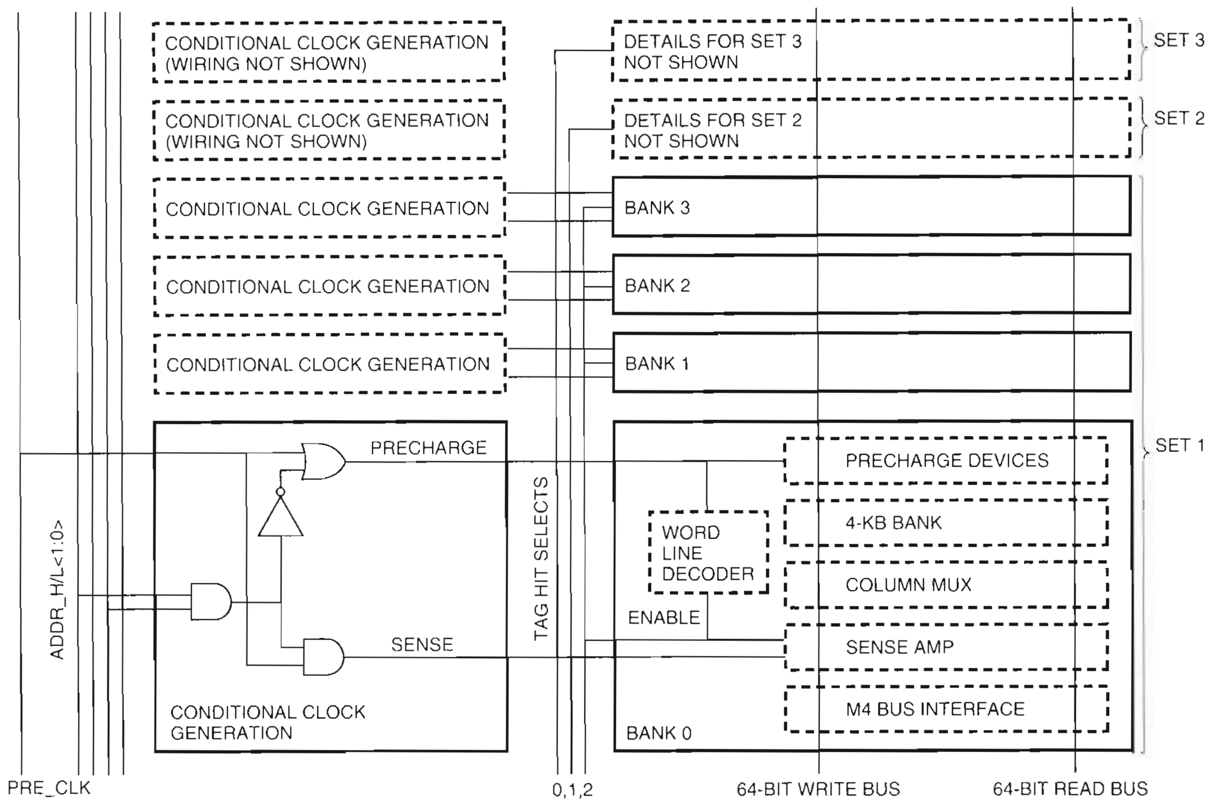

CONDITIONAL CLOCK GENERATION (WIRING NOT SHOWN)
CONDITIONAL CLOCK GENERATION (WIRING NOT SHOWN)
CONDITIONAL CLOCK GENERATION
CONDITIONAL CLOCK GENERATION
CONDITIONAL CLOCK GENERATION
DETAILS FOR SET 3 NOT SHOWN
SET 3
DETAILS FOR SET 2 NOT SHOWN
SET 2
BANK 3
BANK 2
BANK 1
PRECHARGE
SENSE
CONDITIONAL CLOCK GENERATION
ADDR_H/L<1:0>
TAG HIT SELECTS
PRECHARGE DEVICES
4-KB BANK
WORD LINE DECODER
COLUMN MUX
ENABLE
SENSE AMP
M4 BUS INTERFACE
BANK 0
SET 1
PRE_CLK
0,1,2
64-BIT WRITE BUS
64-BIT READ BUS

Figure 10
Schematic of Right Half of L2 Cache Data Arrays

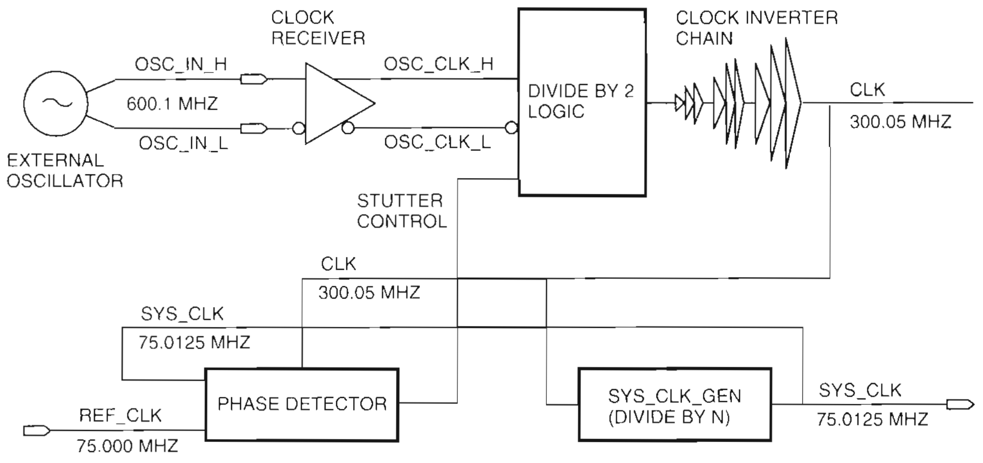


Figure 11
SMPLL Block Diagram

The Alpha 21164 cache partitions, containing 7.2 million of the 9.3 million total devices, are, in themselves, very large and difficult to verify. Accordingly, all three caches were designed and assembled hierarchically. Specifically, each cache bank contains several references to the same precharge, decoder, control, and random access memory (RAM) array logic and layout, which are then instantiated to form the overall cache. Subdividing the cache partitions into major hierarchical blocks reduced the device count per block. In addition, since each bank was identical, only the devices within one bank needed to be verified.

Although the hierarchical method is typically used in semicustom designs, it was new for Digital's full-custom microprocessors. Prior to the development of the Alpha 21164, the caches were designed and verified without an established hierarchy, as was the rest of the chip. Digital's CAD tools handled large databases without hierarchy; the layout verification methods were trusted; and the percentage of duplicated circuitry was small. Consequently, there had been no prior compelling need to design with deep hierarchies.

Toward the end of the chip developmemt, using a considerable amount of computer resources, all three Alpha 21164 cache layouts were also verified without hierarchy to prove the new hierarchical method and CAD tools. The large size of the Alpha 21164 made it the prime candidate for verifying new hierarchical verification tools (which were run concurrently with the traditional ones). Table 1 compares the processing time of the S-cache for both the nonhierarchical and the hierarchical verification methods. The hierarchical approach resulted in a significant improvement in CPU time.

Table 1
Alpha 21164 S-cache Verification Compute Time

Operation	Nonhierarchical Processing	Hierarchical Processing
Netlist extraction	11 hours	6 hours
Netlist comparison	6 hours	30 minutes
Geometric verification	18 hours	10 minutes

Capacitive Coupling and Carrier Injection Verification

Since capacitive coupling between adjacent signals can have a disastrous effect on the logical functionality and long-term reliability of a design, it was a major concern throughout the project. When adjacent nodes switch, coupling between them can result in their logic state being degraded or lost by adding or removing charge to or from the coupled node. For static cases, coupling results in a loss in performance, since the node recovers state if the chip cycle time is slowed. For dynamic nodes, however, state may be lost, leading to a logic failure that occurs regardless of cycle time.

Interconnect coupling capacitance can also lead to voltage excursions above the power supply voltage (V_{DD}) and below ground (V_{SS}) on signals in the chip. For the case of an excursion below V_{SS}, the n-type source/drains connected to the signal become forward biased, injecting minority carriers (electrons) into the substrate. If these minority carriers are collected by *N*-diffusions connected to dynamic nodes, the charge stored on the dynamic node can be corrupted, as shown in Figure 12. Similarly, excursions above V_{DD} forward bias p-type source/drains, which can also lead to data corruption.

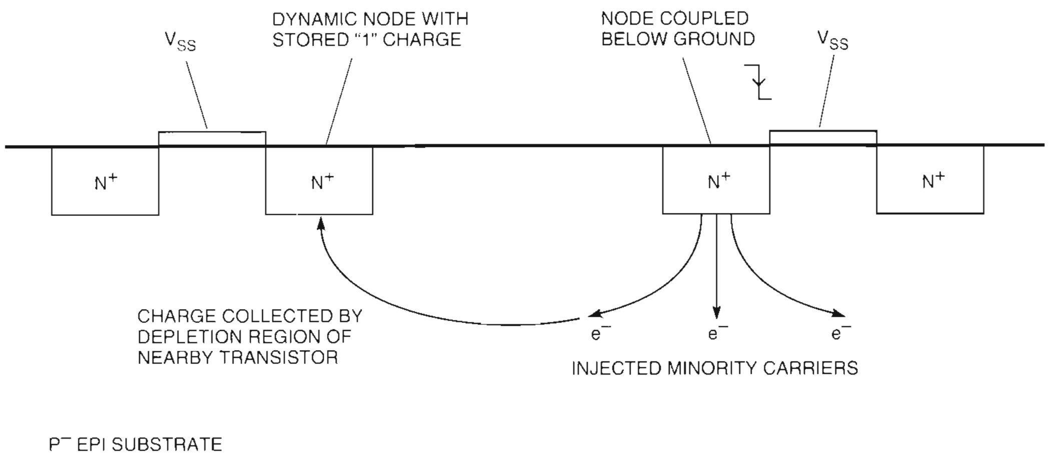
An extensive set of CAD tools was used to identify potential coupling and charge injection problems. In the case of injection checks, a circuit wirelist of the chip was extracted from the layout that included *X-Y* location coordinates for all transistors. An electrical analysis, using capacitances extracted from layout, was then run to identify all nodes that made voltage excursions outside the power supply voltages and that were potential minority carrier injectors. Once these nodes were identified, the CAD tool, which referenced the coordinates from the extracted wirelist, checked all circuitry in the vicinity of the injectors to ensure that there were no dynamic nodes present that could be corrupted. When a potential corruption problem was found, a layout fix was implemented to eliminate the coupling causing the injection. If the coupling could not be reduced or eliminated, a diffusion collector tied to a power rail was placed between the injector and the dynamic node (Figure 13).

Antenna-induced Device Damage Analysis

During the metal etch process, when interconnect is being formed from a blanket layer of metal, stray charge from the etch plasma can be captured by the visible metal. The charge is collected on any polysilicon gate capacitors attached to the node. If enough charge is collected, the gate voltage may rise high enough for tunneling into the gate oxide to occur. This new concern, called antenna-induced device damage, can cause breakdown of the gate oxide, transistor threshold voltage shifts, and long-term reliability problems.

Antenna-induced device damage can be prevented if an alternate path is provided for the collected charge. A diode connection on the antenna node, such as a diffusion connection in either the well or the substrate, acts as such a path. Although all nodes in the Alpha 21164 chip have a diode connection, this connection may not be present at the first or second metalization steps, thereby allowing damage to occur. The magnitude of the damage is dependent on the antenna ratio, defined as the ratio between the area of the visible metal layer being processed and the area of the gates attached to that node through lower-level connecting layers.

To analyze the chip, a special computer-based layout design rule check was developed. This check extracted partial node layout as it would appear during each metalization-patterning etch step and filtered all nodes that did not have a diode shunt connection. For these nodes, antenna ratios were computed and compared to their corresponding ratio limit. To reduce the antenna ratio of a failing node, the antenna metal was broken into sections and metal jumpers, which were placed in the next-higher adjacent metal layer, to connect the sections into a single node. This reduced the charge-collecting area for the section of interconnect



V_{SS}
DYNAMIC NODE WITH STORED "1" CHARGE
NODE COUPLED BELOW GROUND
V_{SS}
N^+
N^+
N^+
N^+
CHARGE COLLECTED BY DEPLETION REGION OF NEARBY TRANSISTOR
e^-
e^-
e^-
INJECTED MINORITY CARRIERS
P^- EPI SUBSTRATE

Figure 12
Dynamic Node Corruption Caused by Minority Carrier Injection

that had the polysilicon gate attached and, as a result, reduced the antenna ratio. If this approach was not feasible or did not reduce the antenna ratio adequately, a diffusion diode was attached to the offending antenna to shunt the charge away.

Electromigration Reliability Analysis

The methods and algorithms used to perform the electromigration (EM) analysis on the Alpha 21164 chip have greatly improved since previously reported.[5] The chief enhancements are the analysis of unidirectional and bidirectional current flow, the addition of thermal heating models, and the introduction of statistical electromigration budgeting.

From a design perspective, one of the main improvements in EM analysis was the introduction of unidirectional and bidirectional current flow limits. Unidirectional current is the flow of current in one direction, for example in wires connecting devices to power or ground. The segment of wire connecting a complementary logic gate to its load is considered bidirectional since the current flows toward the load to charge its capacitance and flows back to the driver as the capacitance is discharged. The bidirectional behavior of current has been shown to improve EM reliability by at least a factor of two. This is a tremendous benefit as nearly all on-chip signal wiring is bidirectional. (Power supply metal is not and must be treated accordingly.)

The most stringent EM requirement is meeting the traditional average current density limit of 2.0 milliampere/μm^2. Statistical electromigration budgeting (SEB) was used for the first time during design verification to assess the impact of allowing small portions of the Alpha 21164 design to exceed the fixed EM average current limits.[6] Statistical parameters characterizing EM risk for the 0.5-μm CMOS interconnect process were combined with the average node currents and layout geometry to compute the magnitude of the EM risk of all design rule violations taken

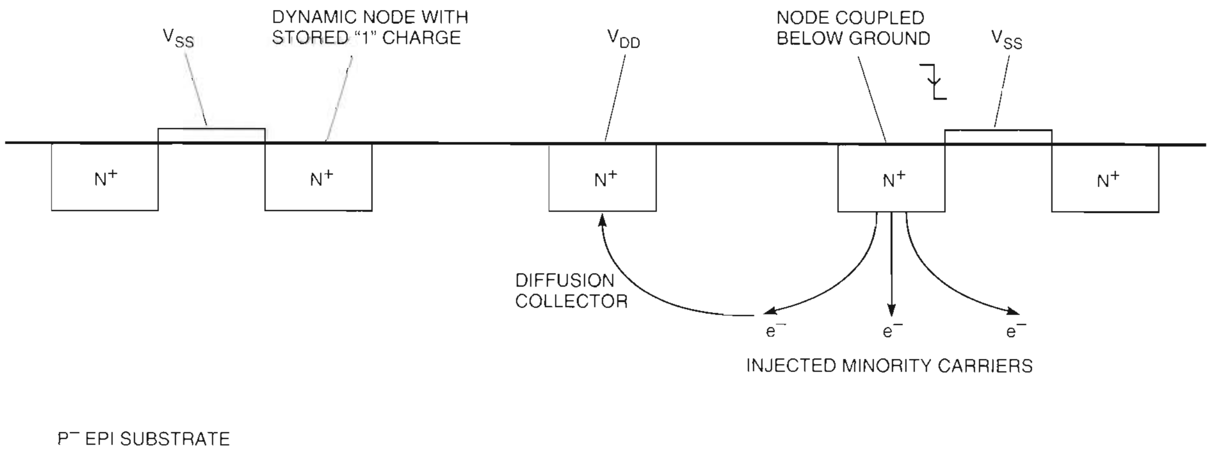


Figure 13
Dynamic Node Protected from Minority Carrier Injection

together. Only those violations that added significant risk were required to be fixed. This reduced design verification time and retained performance advantages while ensuring that the Alpha 21164 design met its chip-level reliability goals.

Conclusion

The implementation details of the Alpha 21164 microprocessor have been described. The custom VLSI chip contains 9.3 million transistors, including a 96-KB second-level cache, in an area of 299 mm^2. The chip implements the Alpha instruction set architecture and can issue up to four instructions at a time. It reaches a peak execution rate of 1.2 billion instructions per second (bips) and 600 MFLOPS. The Alpha 21164 is the fastest and highest-performance microprocessor designed to date in the industry.[7]

The chip achieved its performance goal of 300-MHz operation in a 0.5-μm CMOS technology by employing a fine-pitch, low-resistance power grid; a low-skew clock distribution network; fast latches; and high-speed circuit techniques. Extensive verification of the functionality, electrical circuits, and physical layout was performed to ensure the functionality and reliability of the design. The chip operates from a 3.3-V supply and dissipates 50 W. It is easily air-cooled using conventional technology. First-pass silicon was functional and booted three operating systems running on a number of different system platforms.

Acknowledgments

We would like to acknowledge the contributions of many people who helped make this chip possible. These include William Herrick and Paul Rubinfeld for management and project support; Alan Cave and Radenko Cvijetic for invaluable CAD assistance; and Larry Bair, Narain Arora, Len Gruber, and Bjorn Zetterlund for device and technology modeling. Designers include Randy Allmon, Roy Badeau, Pete Bannon, Todd Benninghoff, Randel Blake-Campos, Derek Brasili, Kevin Broch, Todd Broch, Mike Charnoky, Beth Cooper, Dan Dever, Rob Dupcak, Tim Fischer, Frank Fox, Rich Fromm, Bruce Gieseke, Mary Gowan, Charles Hightower, Jim Keller, John Kowaleski, Tim Mast, Anthony Murphy, John Mylius, Andy Olesin, Tung Pham, Nate Raughley, Don Priore, Vidya Rajagopalan, Steve Strickland, Chandra Somanathan, Jon White, Gil Wolrich, and the authors of this paper. Custom layout was done by Picco Aires, Sandy Carroll, Jeff Ceparski, Danielle DeMarse, Gina Franceschi-Bean, Mark Gaetz, Natasha Geagan, Jerry Heath, Susan Lowell, Tom McDermott, Karen McFadden, Rich Matthew, Stephanie Miller, Sue Moore, Brian Mulhollen, Dave Olson, Marie Riley, Avraham Shenvald, Chad Stark, Marc Tareila, Lang Tran, and Greg Williams.

References and Note

1. J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300 MHz 64-bit Quad-issue CMOS RISC Microprocessor," *Digital Technical Journal*, vol. 7, no. 1 (1995, this issue): 119–135.

2. D. Dobberpuhl et al., "A 200-MHz 64-bit Dual issue CMOS Microprocessor," *Digital Technical Journal*, vol. 4, no. 4 (Special Issue 1992): 35–50.

3. SPICE is a general-purpose circuit simulator program developed by Lawrence Nagel and Ellis Cohen of the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley.

4. K. Kurita et al., "PLL-Based BiCMOS On-Chip Clock Generator for Very High-Speed Microprocessor," *IEEE Journal of Solid State Circuits*, vol. 26, no. 4 (April 1991): 585–589.

5. D. Donchin et al., "The NVAX CPU Chip: Design Challenges, Methods, and CAD Tools," *Digital Technical Journal*, vol. 4, no. 3 (Summer 1992): 24–37.

6. J. Kitchin, "Statistical Electromigration Risk Budgeting for Reliable Design and Verification in a 300MHz Microprocessor," *Digest of Technical Papers, VLSI Circuits Symposium*, 1995.

7. "Digital Leads the Pack with 21164," *Microprocessor Report*, vol. 8, no. 12 (September 1994): 1, 6–10.

Biographies

William J. Bowhill
William Bowhill is a consultant engineer in Digital Semiconductor's High Performance CPU Group. He co-led the implementation of the Alpha 21164 CPU and represented the design organization during the development of the 0.5-μm CMOS process in which the chip is fabricated. His previous responsibilities include technical contributions to both the VAX 6000 Model 400 and Model 600 chip sets. Before joining Digital in 1985, he worked for Standard Telecommunications Laboratories, Harlow, England, where he designed VLSI chips for telecommunication applications. Bill received a B.Eng. (honors) in electronic engineering from the University of Liverpool in 1981.

Shane L. Bell
Shane Bell joined Digital after receiving a B.S. in computer systems engineering from the University of Massachusetts at Amherst in 1993. As a hardware engineer in Digital Semiconductor, he worked on the integer execution unit of the Alpha 21164 CPU. He is currently involved in the design of another high-performance microprocessor. Shane is a member of Eta Kappa Nu, Tau Beta Pi, and IEEE.

Bradley J. Benschneider
Brad Benschneider is a principal hardware engineer in Digital Semiconductor. He was responsible for designing various sections of the memory management unit on the 21164, as well as defining the latching methodology for the chip. He is currently leading the implementation effort of the memory management unit for the next-generation Alpha CPU. Since joining Digital in 1987, he has contributed to several custom chip designs in the VAX 6000 family and the early Alpha implementations. He received a B.S.E.E. from the University of Cincinnati, has one patent, and has co-authored four papers.

Andrew J. Black
Andy Black is a senior hardware engineer in Digital's Palo Alto Design Center, where he is designing the bus interface unit for the StrongARM PDA microprocessor chip. During his work on the Alpha 21164 CPU, he was a member of the design team for the memory management unit and contributed to the chip's clock design. Andy joined Digital in 1992 after working for International Solar Electric Technology. He received a B.S.E.E. from Pennsylvania State University and an M.S.E.E. from the University of Southern California. Andy is a member of IEEE, Tau Beta Pi, and Eta Kappa Nu.

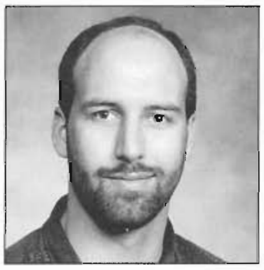
Sharon M. Britton
Sharon Britton is a principal hardware engineer in Digital Semiconductor. She received a B.S.E.E. from Boston University in 1983 and an M.S.E.E. from MIT in 1990. She joined Digital in 1983 to work on the design and development of optical disk drive controllers. Since joining Digital Semiconductor in 1990, Sharon has contributed to the design of the floating-point unit on the 21064 CPU chip and led the implementation of the load/store unit for the Alpha 21164 CPU. She is currently a member of the design team working on the instruction issue unit for the next-generation Alpha chip.

Ruben W. Castelino
Before receiving a B.S.E.E. from the University of Cincinnati in 1988, Ruben Castelino was a co-op student at Digital working on a chip set for the VAX 6000 Model 200. Currently a senior hardware engineer in Digital Semiconductor, he was a codesigner of the cache control and bus interface unit for the Alpha 21164 CPU. Prior to that, he worked on the instruction fetch, decode, and branch unit for the NVAX chip and performed implementation work for the NVAX virtual instruction cache. Ruben is currently a codesigner of the cache control and bus interface unit for a new Alpha microprocessor.

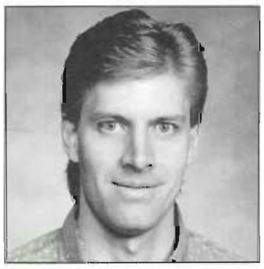
Dale R. Donchin
Dale Donchin is an engineering manager and technical contributor in Digital Semiconductor. He designed several circuits related to the clock and cache and contributed to and led CAD tool use for the Alpha 21164 CPU. He is presently performing these duties for the development of the next-generation Alpha microprocessor. Dale joined

Digital in 1978 and was previously a development manager in the RSX Operating System Group. Dale holds a B.S.E.E. (1976, honors) and an M.S.E.E. (1978) from Rutgers University College of Engineering and is a member of IEEE and ACM.

John H. Edmondson
John Edmondson is a consultant engineer in Digital Semiconductor. He was the architecture leader of the design team for the Alpha 21164 microprocessor. Previous to that work, he was a member of the design team for the VAX 6000 Model 600 microprocessor. Prior to joining Digital in 1987, John worked at Canaan Computer Corporation and Massachusetts General Hospital. John received a B.S.E.E. from the Massachusetts Institute of Technology in 1979.



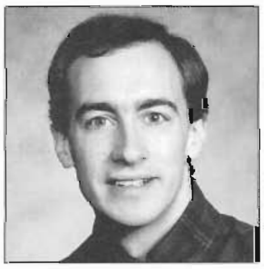
Harry R. Fair, III
Harry Fair is a senior hardware engineer in Digital Semiconductor's Advanced Development Group and is currently working on the design of the instruction issue unit for a high-performance microprocessor. Harry came to Digital in 1985 as a co-op student and worked on the VAX 6000 Model 400 chip set. He joined Digital after receiving a B.S.E.E. from Purdue University in 1989. Since then he has contributed to the NVAX and NVAX+ microprocessor designs and most recently was a member of the design teams for the integer execution unit and memory management unit of the Alpha 21164. Harry is a member of IEEE.



Paul E. Gronowski
Paul Gronowski joined Digital in 1984 after receiving a B.S. degree in electrical engineering from the University of Cincinnati. During the past 10 years with Digital Semiconductor, he has contributed to the design of several high-performance microprocessors. For the Alpha 21164 CPU, he was responsible for the integer execution unit and led the physical chip verification effort. He is currently responsible for the technical design and management of the next-generation processor. He is the co-author of several ISSCC papers and holds one patent.

Anil K. Jain
Anil Jain, a consulting engineer in Digital Semiconductor, led the implementation of the external interface unit on the Alpha 21164 microprocessor. Prior to this, he was the project leader for the floating-point unit on the NVAX microprocessor. He also made technical contributions on the CVAX microprocessor and on device modeling of Digital's first CMOS process. Anil received a B.S.E.E. from Punjab Engineering College (1978) and an M.S.E.E. from the University of Cincinnati (1980). He holds three patents.

Patricia L. Kroesen
A principal engineer in Digital Semiconductor, Patricia Kroesen is currently a circuit designer of the cache controller and I/O interface section for the next-generation Alpha microprocessor. In her work on the Alpha 21164, she was an implementer on the floating-point unit and optimized the clock distribution system for the PASS2 release. Since joining Digital in 1988, Patty has also worked on advanced development efforts of bipolar and GaAs chips. She has a B.S.E.E. from the University of Michigan and an M.S.E.E. from Polytechnic Institute. She holds one patent and has co-authored several papers.



Marc E. Lamere
A principal hardware engineer in Digital Semiconductor, Marc Lamere is currently a CMOS circuit designer for the next-generation Alpha microprocessor. In his work on the

Alpha 21164, he was responsible for the integer execution unit shifter and other circuit designs as well as the physical and electrical verification of the chip. Marc joined Digital in 1984 as an ECL circuit designer on the VAX 9000 project and helped design custom and semicustom bipolar chips. He holds a B.S.E.E. (1983) from Rensselaer Polytechnic Institute and an M.S.E.E. (1988) from Northeastern University.

Bruce J. Loughlin
Consultant engineer Bruce Loughlin was responsible for the signal integrity design for the Alpha 21164 chip. Since joining Digital in 1975, Bruce has contributed to many projects, including the clock design of the DEC 3000 workstation, corporate FCC shielding strategy, design of the FDDI physical interface, and disk servo writing equipment for the R80 Winchester disks. From 1971 to 1975, Bruce was a member of the Eclipse design team at Data General. Prior to that, he was the vice president of engineering for Data Technology Inc., a company he cofounded in 1961. Bruce holds B.S. degrees in electrical engineering and mechanical engineering and an M.S. in aeronautics and astronautics, all from MIT.

Shekhar Mehta
Shekhar Mehta is a senior hardware engineer in Digital Semiconductor's High Performance Computing Group. He designed the miss address file on the memory subsystem of the Alpha 21164 CPU and was responsible for the electromigration checks of the chip. He is currently leading the design of the caches on a future Alpha microprocessor. Before joining Digital in 1988, Shekhar was an engineer at Larsen & Toubro, Bombay, India. He received an M.S.E.E. from the University of Wisconsin at Madison (1988).

Robert O. Mueller
Rob Mueller joined Digital in 1990 after receiving a B.S. in computer and systems engineering from Rensselaer Polytechnic Institute. As a senior hardware engineer in Digital Semiconductor, he is currently involved in the design and implementation of the pad ring for a new Alpha microprocessor. In his work on the Alpha 21164 chip, Rob contributed to the design, implementation, and electrical verification of the pad ring, the cache control, and the bus interface unit.

Ronald P. Preston
Ronald Preston is a principal engineer in Digital Semiconductor. Since joining Digital in 1988, he has worked on the design of several microprocessors and was the implementation leader for the instruction unit on the Alpha 21164. Ron was also responsible for the architecture and implementation of the issue/bypass/scoreboard logic. Ron is the coauthor of several articles on hot carrier analysis of CMOS circuits. He received a B.S.E.E. in 1984 and an M.S.E.E. in 1988, both from Rensselaer Polytechnic Institute. Ron is a member of Eta Kappa Nu and IEEE.

Sribalan Santhanam
Sri Santhanam received a B.E. in electrical engineering from Anna University, Madras, India, in 1987, and an M.S.E. degree in computer science and engineering from the University of Michigan in 1989. He joined Digital as a design engineer for Digital Semiconductor, responsible for the full-custom design and development of high-performance CMOS VLSI processors. Sri worked on the design of the floating-point unit of the 21064 CPU and the design of the cache control unit of the Alpha 21164 CPU. He is currently a member of the Low Power Alpha Group where he is involved in the design of a low-power microprocessor.

Timothy A. Shedd
Before receiving a B.S.E.E. from Purdue University in 1992, Tim Shedd was a co-op student at Digital working on several VAX CPUs as well as the floating-point unit of the 21064 CPU. He is now a hardware engineer in Digital Semiconductor's Advanced Development Group. Tim contributed to the circuit design of the Alpha 21164 microprocessor's instruction issue unit and is currently working on the memory management unit of the next-generation Alpha CPU. Tim is a member of Tau Beta Pi and Eta Kappa Nu.

Michael J. Smith
A principal engineer in Digital Semiconductor, Michael Smith was a member of the instruction unit design team for the Alpha 21164 microprocessor, responsible for floorplanning, logic, and circuit design. Prior to this, he was involved in the design of two memory controller/bus adapter chips for the VAX 4000 Models 300 and 600. Currently he is a member of the bus interface and instruction unit teams of the next-generation Alpha microprocessor. Michael joined Digital in 1986 after receiving a B.S.E.E. from the Rochester Institute of Technology.

Stephen C. Thierauf
Stephen Thierauf joined Digital in 1976. As a consulting hardware engineer in Digital Semiconductor, Stephen is currently responsible for I/O circuit design, on- and off-chip signal integrity, and I/O modeling for Alpha microprocessors and PCI peripherals. He has designed or led the circuit design for a number of high-performance telecommunication and peripheral chips. Previous responsibilities include I/O circuit design and system-level signal integrity analysis, micropackaging analysis, and micropackaging design for numerous high-performance microprocessors and peripherals. He is a member of IEEE.

Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor

John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, Gilbert M. Wolrich

A new CMOS microprocessor, the Alpha 21164, reaches 1,200 mips/600 MFLOPS (peak performance). This new implementation of the Alpha architecture achieves SPECint92/SPECfp92 performance of 345/505 (estimated). At these performance levels, the Alpha 21164 has delivered the highest performance of any commercially available microprocessor in the world as of January 1995. It contains a quad-issue, superscalar instruction unit; two 64-bit integer execution pipelines; two 64-bit floating-point execution pipelines; and a high-performance memory subsystem with multiprocessor-coherent write-back caches.

Overview of the Alpha 21164

The Alpha 21164 microprocessor is now a product of Digital Semiconductor. The chip is the second completely new microprocessor to implement the Alpha instruction set architecture. It was designed in Digital's 0.5-micrometer (μm) complementary metal-oxide semiconductor (CMOS) process. First silicon was powered on in February 1994; the part has been commercially available since January 1995. At SPECint92/SPECfp92 ratings of 345/505 (estimated), the Alpha 21164 achieved new heights of performance.

The performance of this new implementation results from aggressive circuit design using the latest 0.5-μm CMOS technology and significant architectural improvements over the first Alpha implementation.[1] The chip is designed to operate at 300 MHz, an operating frequency 10 percent faster than the previous implementation (the DECchip 21064 chip) would have if it were scaled into the new 0.5-μm CMOS technology.[2] Relative to the previous implementation, the key improvements in machine organization are a doubling of the superscalar dimension to four-way superscalar instruction issue; reduction of many operational latencies, including the latency in the primary data cache; a memory subsystem that does not block other operations after a cache miss; and a large, on-chip, second-level, write-back cache.

The 21164 microprocessor implements the Alpha instruction set architecture. It runs existing Alpha programs without modification. It supports a 43-bit virtual address and a 40-bit physical address. The page size is 8 kilobytes (KB).

In the following sections, we describe the five functional units of the Alpha 21164 microprocessor and relate some of the design decisions that improved the performance of the microprocessor. First, we give an overview of the chip's internal organization and pipeline layout.

Internal Organization

Figure 1 shows a block diagram of the chip's five functional units: the instruction unit, the integer function unit, the floating-point unit, the memory unit, and the cache control and bus interface unit (called the

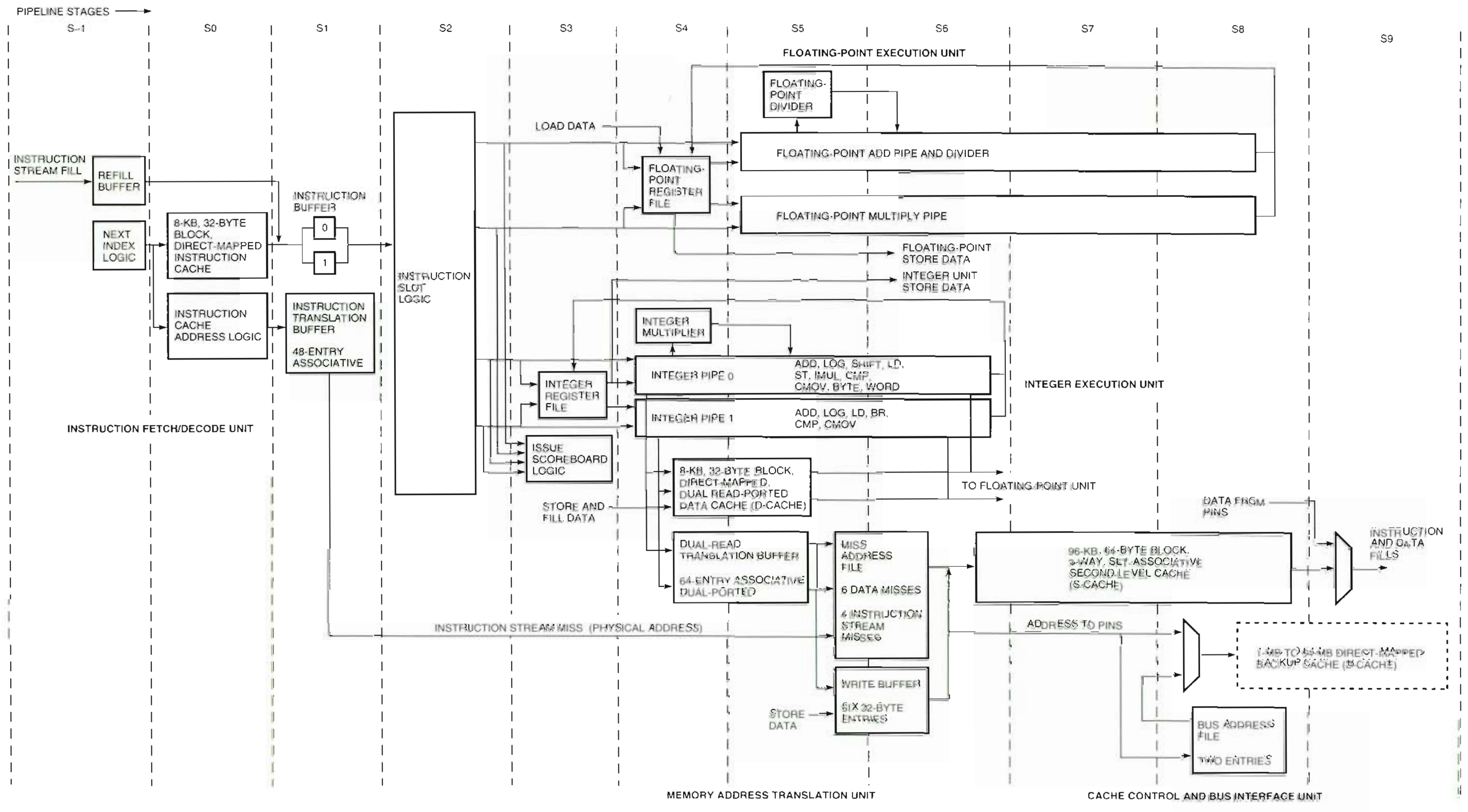

PIPELINE STAGES
S-1
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
FLOATING-POINT EXECUTION UNIT
FLOATING-POINT DIVIDER
LOAD DATA
FLOATING-POINT ADD PIPE AND DIVIDER
FLOATING-POINT REGISTER FILE
FLOATING-POINT MULTIPLY PIPE
FLOATING-POINT STORE DATA
INTEGER UNIT STORE DATA
INSTRUCTION STREAM FILL
REFILL BUFFER
NEXT INDEX LOGIC
8-KB, 32-BYTE BLOCK, DIRECT-MAPPED INSTRUCTION CACHE
INSTRUCTION BUFFER
0
1
INSTRUCTION CACHE ADDRESS LOGIC
INSTRUCTION TRANSLATION BUFFER
48-ENTRY ASSOCIATIVE
INSTRUCTION SLOT LOGIC
INTEGER MULTIPLIER
INTEGER REGISTER FILE
INTEGER PIPE 0
ADD, LOG, SHIFT, LD, ST, IMUL, CMP, CMOV, BYTE, WORD
INTEGER PIPE 1
ADD, LOG, LD, BR, CMP, CMOV
INTEGER EXECUTION UNIT
INSTRUCTION FETCH/DECODE UNIT
ISSUE SCOREBOARD LOGIC
8-KB, 32-BYTE BLOCK, DIRECT-MAPPED, DUAL READ-PORTED DATA CACHE (D-CACHE)
TO FLOATING-POINT UNIT
STORE AND FILL DATA
DATA FROM PINS
INSTRUCTION AND DATA FILLS
DUAL-READ TRANSLATION BUFFER
64-ENTRY ASSOCIATIVE DUAL-PORTED
MISS ADDRESS FILE
6 DATA MISSES
4 INSTRUCTION STREAM MISSES
96-KB, 64-BYTE BLOCK, 3-WAY, SET-ASSOCIATIVE SECOND-LEVEL CACHE (S-CACHE)
INSTRUCTION STREAM MISS (PHYSICAL ADDRESS)
ADDRESS TO PINS
1-MB TO 64-MB DIRECT-MAPPED BACKUP CACHE (B-CACHE)
WRITE BUFFER
SIX 32-BYTE ENTRIES
STORE DATA
BUS ADDRESS FILE
TWO ENTRIES
MEMORY ADDRESS TRANSLATION UNIT
CACHE CONTROL AND BUS INTERFACE UNIT

Figure 1
Five Functional Units on the Alpha 21164 Microprocessor

C-box). The three on-chip caches are also shown. The instruction cache and data cache are primary, direct-mapped caches. They are backed by the second-level cache, which is a set-associative cache that holds instructions and data.

Alpha 21164 Pipeline

The Alpha 21164 pipeline length is 7 stages for integer execution, 9 stages for floating-point execution, and as many as 12 stages for on-chip memory instruction execution. Additional stages are required for off-chip memory instruction execution. Figure 2 depicts the pipeline for integer, floating-point, and memory operations.

Instruction Unit

The instruction unit contains an 8-KB, direct-mapped instruction cache, an instruction prefetcher and associated refill buffer, branch prediction logic, and an instruction translation buffer (ITB).

The instruction unit fetches and decodes instructions from the instruction cache and dispatches them to the appropriate function units after resolving all register and function-unit conflicts. It controls program flow and all aspects of exception, trap, and interrupt handling. In addition, it manages pipeline control for the integer and floating-point units, controlling all data bypasses and register file writes.

The instruction cache has 32-byte blocks. The cache tags hold virtual address information. Its tags also support PALcode through a bit which indicates that the tag contains a physical address. (PAL stands for privileged architecture library and refers to physically addressed code executed in a privileged hardware mode that implements an architecturally defined interface between the operating system and the hardware.)

Instruction Pipeline

The first four pipeline stages of the Alpha 21164 microprocessor are the instruction unit pipeline stages, stage 0 through stage 3. The logic in the stage before stage 0 is normally considered to operate in stage 1 of the pipeline. In that stage, the new instruction cache address is calculated either by incrementing the previous address or by selecting a new address in response to a predicted or actual flow change operation.

During stage 0, the 8-KB instruction cache is accessed. It returns a naturally aligned block of four instructions (16 bytes) with 20 bits of previously decoded instruction information (5 bits per instruction). The precalculated decode information is used in stage 1 for branch and jump processing and in stage 2 for instruction slotting.

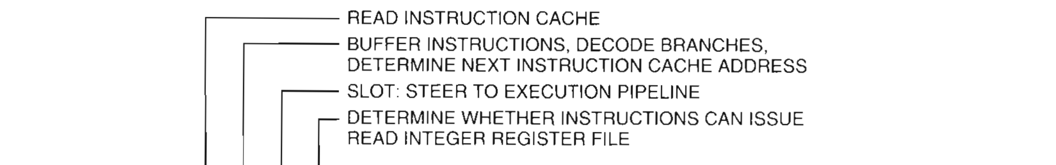

READ INSTRUCTION CACHE
BUFFER INSTRUCTIONS, DECODE BRANCHES, DETERMINE NEXT INSTRUCTION CACHE ADDRESS
SLOT: STEER TO EXECUTION PIPELINE
DETERMINE WHETHER INSTRUCTIONS CAN ISSUE READ INTEGER REGISTER FILE
S0 S1 S2 S3
FIRST INTEGER PIPELINE STAGE
SECOND INTEGER PIPELINE STAGE
WRITE INTEGER REGISTER FILE
S4 S5 S6
INTEGER PIPELINE
READ FLOATING-POINT REGISTER FILE
FIRST FLOATING-POINT PIPELINE STAGE
LAST FLOATING-POINT OPERATE STAGE, WRITE FLOATING-POINT REGISTER FILE
S4 S5 S6 S7 S8
FLOATING-POINT PIPELINE
CALCULATE VIRTUAL ADDRESS, BEGIN DATA CACHE READ
END DATA CACHE READ, TRANSLATE VIRTUAL ADDRESS IN DTB
S4 S5 S6 S7 S8 S9 S10 S11 S12
MEMORY ACCESS PIPELINE
USE DATA CACHE DATA, WRITE STORE DATA TO DATA CACHE, BEGIN SECOND-LEVEL CACHE TAG ACCESS
END SECOND-LEVEL CACHE TAG ACCESS
BEGIN SECOND-LEVEL CACHE DATA ACCESS
END SECOND-LEVEL CACHE DATA ACCESS
BEGIN DATA CACHE FILL
END DATA CACHE FILL
USE SECOND-LEVEL CACHE DATA

Figure 2
Alpha 21164 Pipeline Stages

In stage 1, the four-instruction block is copied into one entry of the two-entry instruction buffer (IB). Also in stage 1, the instruction cache and ITB each check for hits, and the branch-and-jump prediction logic determines new fetch addresses.

The main function of stage 2 is steering each instruction to an appropriate function unit. This process, called instruction slotting, resolves all static execution conflicts. The instruction slotter accepts the next four-instruction block from the IB into a staging register at the beginning of stage 2 and routes the individual instructions to the appropriate functional pipelines as it advances them to stage 3. If the block contains certain mixes of instruction types, it is able to slot all four instructions in a single cycle. Otherwise, it advances as many instructions as possible in the first cycle. The remaining instructions in the block are slotted during subsequent cycles. Instructions are slotted strictly in program order. A new four-instruction block enters stage 2 when every instruction in the prior block has been slotted and advanced to stage 3.

The issue stage operates in stage 3. It performs all dynamic conflict checks on the set of instructions advanced from stage 2. The issue stage contains a complex register scoreboard to check for read-after-write and write-after-write register conflicts. This stage also detects function-unit-busy conflicts, which can occur because the integer multiplier and floating-point divider are not fully pipelined. The register scoreboard logic detects all integer and floating-point operand bypass cases and sends the necessary bypass control signals.

The issue stage issues instructions to the appropriate function units unless it encounters a dynamic conflict. If a conflict occurs, the instruction and logically subsequent instructions are stalled (not issued). A stall in stage 3 also stalls the advance of the next set of slotted instructions from stage 2. This stall ends when all instructions in stage 3 have been issued.

To perform conflict checking and to handle exceptions (including traps and interrupts), the instruction unit tracks the instructions issued during stage 4 through stage 8. The instruction unit sends register file write strobes and addresses to the integer and floating-point register files for instructions that reach the retire point (stage 6) without an exception. In the event of an exception, write strobes are withheld (gated) to prevent incomplete instructions from updating the register file. These instructions do not complete either because they caused an exception or because they are in the "shadow" of an exception. The shadow of an exception includes all instructions that are in the pipeline when an exception is recognized but are logically subsequent to the instruction taking the exception.

The issue stage stalls for a single cycle to permit the integer multiplier or floating-point divider to return a result into its associated pipeline. This is necessary because the register files do not have extra write ports dedicated to receiving these results. The issue stage also stalls for one cycle in similar cases to permit data fills for load instructions that missed in the data cache to write to the register file and data cache. The issue stage stalls indefinitely when necessary to execute the trap barrier and memory barrier instructions.

No-op Instructions

New instructions are shifted into the slotting and issue stages when a given stage becomes completely empty. Compared to an ideal design in which instructions are shifted to fill a given stage partially, this design has a slightly increased average cycles-per-instruction ratio. We considered the alternative in which instructions are shifted in as slots become available. This alternative would have created critical paths that would increase the CPU cycle time by approximately 10 percent. An evaluation of our trace-driven performance model showed that the alternative did not reduce the cycles-per-instruction ratio enough to compensate for the reduction in cycle time. As a result, we chose the simpler and faster design.

Compilers and assembly language programmers can insert no-op instructions to minimize and, in most cases, to eliminate any negative performance effect. To facilitate this process, the Alpha 21164 microprocessor handles three different kinds of no-op instruction.

The first two kinds of no-op instruction are the integer no-op (NOP) and the floating-point no-op (FNOP). NOP (BIS R31,R31,R31) can issue in either integer execution pipeline. FNOP (CPYS F31,F31,F31) can issue in either floating-point execution pipeline. The compiler uses these to improve performance when two instructions would be slotted together even though they cannot issue in the same cycle. If one instruction in a pair is dependent on the other, issuing them together guarantees the second will stall in the issue stage and prevent later instructions from entering that stage. The compiler inserts a NOP or FNOP to delay the issue of the second instruction. With this improvement, the second instruction can be issued with later instructions.

The third kind of no-op instruction, the universal no-op (UNOP), is detected in stage 2. UNOP [LDQ_U R31,0(Rnn)] is discarded in stage 2 so that it does not require an issue slot in either pipeline. UNOP allows compilers to align instructions without the unnecessary use of pipeline issue slots. For example, the compiler can align the target of a branch without necessarily slowing execution of the fall-through path to that branch.

Instruction Prefetcher and Refill Buffer

The instruction prefetcher operates in parallel with the instruction cache. When an instruction is not in either the instruction cache or refill buffer, the prefetcher generates a stream of 32-byte instruction block fetch requests to fill the 4-entry refill buffer with instruction data. Each instruction block contains 8 instructions. Fetched instruction data is stored in the refill buffer when it is returned. Four-instruction subblocks of instruction data are moved from the refill buffer to the IB when needed. At that time, the instruction cache is also updated. If this data movement empties an entry in the refill buffer, an additional fetch request is initiated. Fetched instruction data is buffered in the refill buffer rather than the instruction cache to avoid evicting valid cache blocks unnecessarily.

The refill buffer is a type of stream buffer. Each entry stores a virtual address and has a comparator so the refill buffer can be probed for instruction data on a cache miss. Instruction fetching begins only if an access misses in both the instruction cache and the refill buffer. Fetching stops when any instruction flow change occurs (i.e., branch, jump, exception, etc.). It also stops if at any time the instructions needed in stage 1 are found in the instruction cache.

The combination of the on-chip, 96-KB second-level cache and the instruction prefetcher significantly reduces the benefit of enlarging the instruction cache beyond its current size of 8 KB. The prefetcher generates requests at a high rate. Because it is on-chip, the second-level cache has the bandwidth to handle requests quickly and with relatively little effect on data-stream requests. In general, the performance benefit from making the instruction cache larger is very small. This is one of the benefits of the two-level on-chip cache hierarchy.

Instruction Stream Address Translation and the Instruction Translation Buffer

The instruction unit contains a 48-entry, fully associative instruction translation buffer (ITB) that holds instruction stream address translations and protection information. Each entry in the ITB can map 1, 8, 64, or 512 contiguous 8-KB pages.

During stage 1, the ITB entries are checked for a match with the program counter (PC). If the page is found, its protection bits are checked against the current operating mode. If the page is not found, an ITB miss trap occurs. If the page is found in the ITB and the access is an instruction cache miss, the ITB supplies the physical page address to the prefetcher.

Branch and Jump Prediction

The branch prediction logic examines the block of instructions coming from the instruction cache or refill buffer during stage 1. It checks the block for control instructions (taken conditional branches, jumps, subroutine return instructions, and other flow-change instructions) and calculates the new fetch address. Since the new fetch address is available at the end of stage 1, the read of the instruction cache for the target instruction occurs in the next cycle. This means the control instruction is in stage 2 at the same time as the target instruction is in stage 0, resulting in a one-cycle branch delay that creates an empty cycle in the pipeline. The IB quashes this empty cycle if any stall occurs ahead of it in the pipeline.

The branch prediction logic predicts conditional branch instructions using a branch history table with 2K entries addressed by low-order bits of the PC. Each is a two-bit counter that increments when branches are taken and decrements when branches are not taken. The counter saturates at the top and bottom counts. A branch is predicted to be taken if the current counter value is one of the two highest counts; otherwise, it is predicted to be not-taken. This method is more effective than the method used in the first Alpha microprocessor (which had only one bit of history per entry), partly because it reduces the misprediction rate for typical loop branches by half.

A 12-entry return address stack is used to predict the target address on subroutine returns (i.e., RET, JSR_COROUTINE) and returns from PALcode. Each entry stores 11 bits of address, which is sufficient to address the 8-KB instruction cache. The upper 32 bits of the target address are predicted by using the value in the instruction cache tag that is addressed by the return address stack. The same basic mechanism is used to predict the full target address of jump and jump-type subroutine call instructions since the Alpha architecture provides a hint field in these instructions that indicates the target cache address.

The Alpha 21164 microprocessor recovers from incorrect branch and PC predictions by taking a mispredict trap when the incorrectly predicted branch or jump-type instruction executes in the execution unit. For a typical branch misprediction, the execution time is five cycles longer.

Replay Traps

In a replay trap, the instruction unit prevents completion of a given instruction by trapping the instruction and then restarting execution immediately with that instruction. The trap mechanism prevents completion of subsequent instructions. This mechanism replays the instruction from the beginning of the Alpha 21164 pipeline. It is used when a stall after stage 3 would otherwise be required.

There are three main reasons stalls are not implemented for stages later than stage 3. The ability to stall adds complexity to clocking circuits, particularly in execution unit data paths. In addition, it adds control complexity. An example of this is a stalled two-input function unit in which one input operand is invalid. To end the stall, certain latches must be enabled while others are not, because the valid data must be held in one pipeline latch while the invalid data is replaced in another. Finally, adding stall logic would create additional critical paths. The elimination of stalls beyond stage 3 and the use of the replay trap mechanism avoid these complexities.

The replay trap mechanism is used for a number of unusual memory instruction conflicts and memory unit resource overruns. For example, the load-miss-and-use replay trap is used when a load misses in the data cache and a dependent instruction issues exactly two cycles after the load. The issue decision for such a dependent instruction is made prior to the actual determination of cache hit, so a hit is predicted. If this prediction is wrong, the dependent instruction is restarted from the front of the pipeline and will arrive at the issue stage one cycle before data arrives from the second-level cache. Because the instruction arrives before the data, there is no performance loss due to the trap mechanism.

Integer Function Unit

The integer function unit executes integer operate instructions, calculates virtual addresses for all load and store instructions, and executes all control instructions except floating-point conditional branches. It includes the register file and several integer functional subunits, most of which are contained in two parallel four-stage pipelines. Both pipelines contain an adder and a Boolean logic unit. The first pipeline contains the shifter, and the second pipeline contains the control instruction execution unit. The first pipeline also attaches to the partially pipelined integer multiplier, which operates in the background. Except for the issue cycle and a cycle to return the result, the first pipeline and integer multiplier operate in parallel.

Integer Register File and Bypasses

The integer register file is read during stage 3 and written in stage 6. Bypass paths are implemented to allow all subunits other than the multiplier to receive and use the result of a previous instruction from stage 4, 5, or 6 of either pipeline. Due to implementation constraints, the multiplier can only receive bypassed data from stage 6 of the pipeline. This increases multiply latency by as many as two cycles when multiply input operands are produced by preceding integer operate instructions.

The integer register file contains 40 registers: the 32 integer registers specified by the architecture (R0 through R31) with R31 always reading as 0; and 8 shadow registers available to PALcode as scratch space. The register file is accessed by 4 read ports (2 for each pipeline) and 2 write ports (1 for each pipeline).

Instruction Latencies

Most instructions executed in the integer function unit have a latency of 1 cycle. These instructions execute in stage 4. The conditional move instruction has a latency of 2 cycles. It executes in stage 4 and stage 5.

Multiply latency depends on the data size and the operation being performed. Thirty-two–bit multiplies have an 8-cycle latency, and the multiplier can start a second multiply after 4 cycles, provided that the second multiply has no data dependency on the first. Sixty-four–bit signed multiplies have a 12-cycle latency; the 64-bit multiply unsigned high instruction has a 14-cycle latency; and for both of these 64-bit multiplies, the multiplier can start a nondependent multiply after 8 cycles.

Because of a special bypass, compare and Boolean logic instructions can have a latency of 0 cycles when a conditional move or a branch test input operand is the result of an immediately preceding compare or Boolean logic instruction. The integer unit uses the bypass to allow dual issue of the producer and consumer in this case.

To realize the full benefit from the increased issue width relative to the first Alpha microprocessor, the DECchip 21064, it is critical to reduce operational latencies. As the issue width increases, the cost in instruction execution opportunities for a given latency increases. In the integer unit, the following latencies are reduced relative to the 21064: the shifter latency (from 2 cycles to 1), the byte and word operation latencies (from 2 cycles to 1), and the multiplier latency (from 19 to 23 cycles in the 21064 to 8 to 16 cycles in the Alpha 21164). Also the special bypass for conditional instructions reduces that latency from 1 cycle in the 21064 to 0 cycles in the Alpha 21164. For the most part, these latency reductions are achieved by circuit design improvements.

Integer Load and Store Instructions

Integer load instructions issue in either pipeline and as many as two can issue per cycle. Integer store instructions issue in the first pipeline only. For integer load instructions that hit in the data cache, the data is multiplexed into the output of stage 5 of the pipeline in which the load issued; the data is then written to the register file through the write port associated with that pipeline. For integer load instructions that miss in the data cache, the data is returned later by the memory subsystem. The data is then multiplexed into the output of stage 5 as before, and the instruction unit inserts a properly timed NOP cycle by stalling the issue stage for one cycle to make the pipeline's register write port available.

Floating-point Unit

The floating-point unit consists of the floating-point register file and two pipelined functional subunits: an add pipeline that executes all floating-point instructions except for multiply, and a multiply pipeline that executes floating-point multiplies. All IEEE and VAX rounding modes are done in hardware, including IEEE round to plus and minus infinity.

Pipeline Structure and Operation Latencies

Each floating-point subunit on the Alpha 21164 CPU chip contains three functional stages implemented in four pipeline stages, stage 5 through stage 8. The floating-point register file is read in stage 4 and written at the end of stage 8. Figure 3 depicts the physical layout of the floating-point unit. Figure 4 shows the pipelining of instructions executed in the floating-point unit.

As in the integer unit, latency is reduced in the floating-point unit relative to the previous Alpha implementation. The latency of all floating-point operate instructions, except floating-point divide, is 4 cycles. In the DECchip 21064, most floating-point operations take 6 cycles. The floating-point divide latency varies depending on the input data values. For a single-precision divide, the latency is reduced from 34 cycles in the 21064 to an average of 19 in the 21164; and for a double-precision divide, it is reduced from 63 cycles to an average of 31. As discussed previously, reducing latency is important as issue width increases. As in the integer unit, the reduced latency is achieved mostly by circuit design improvements.

Register File and Bypasses

The floating-point register file has nine ports: two read ports and one write port per functional unit for source and destination operand accesses, one read port for floating-point stores, and two write ports to support two floating-point loads per cycle. Bypass paths forward data from each of the four write buses in the floating-point register file to each of the five read buses.

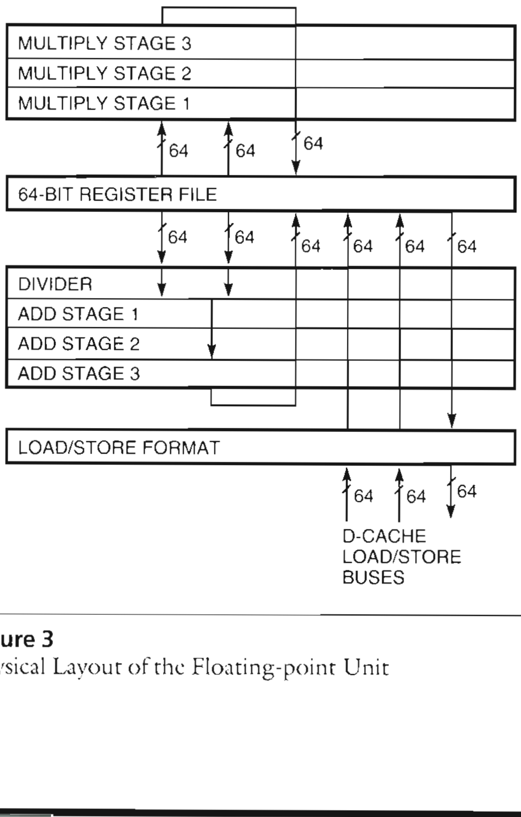


Figure 3
Physical Layout of the Floating-point Unit

Floating-point Load and Store Instructions

In Alpha microprocessors, floating-point numbers are stored in one format in memory and in another format in the floating-point registers. Floating-point load and store instructions convert from one format to the other as they move the data. In the Alpha 21164 pipeline, floating-point input operands are read from the floating-point register file one cycle later than integer input operands are read from the integer register file. This skew provides an extra cycle for floating-point load data format conversion.

Floating-point load and store instructions first issue to the integer unit for address calculation. The issue restrictions are exactly the same as for integer load or store instructions. For floating-point load instructions, the data is written to the register file using one of the two write ports reserved for that purpose. When a conflict for these write ports occurs between a write due to a new load that hit in the data cache and a write due to a previous load that missed, the conflict is resolved by forcing the new load to miss in the data cache.

Add Pipeline

The key components of the add pipeline design are the fast fraction adder, operand data-path alignment, normalization shift detection, sticky-bit calculation, and round-adder design. The fast-adder design operates in a single phase (one phase equals one-half of a CPU cycle). It is used in the function stage 1 and stage 3 fraction adders. To reduce formatting and rounding complexity, the least significant bits in fractions are aligned to one of two different bit positions: one for single-precision data (IEEE S and VAX F) and 4-byte integers, and one for double-precision data (IEEE T, and VAX G and D) and 8-byte integers.

For effective subtracts with exponent differences of -1, 0, or 1, a new normalization shift detect algorithm uses three leading bit chains to examine stage 1 input operands to determine the required normalization shift. The normalization shift amount is chosen by comparing the least significant bit of one exponent to the least significant bit of the other.

The sticky bit for adds and subtracts is determined by comparing the exponent difference with an encoded value for the number of trailing zeros in the fraction being aligned.

The stage 3 round adder operates in one cycle and consists of a fraction adder and an output selector. The fraction adder takes one phase and adds two operands plus rounding bits based on the round mode. The selector assembles the fraction result based on global carry-and-propagate information from the adder. It also examines the adder result alignment and performs a final normalization shift of as much as one bit left or right. The exponent result is also selected in stage 3 before the complete result is sent to the register file write bus and bypass logic.

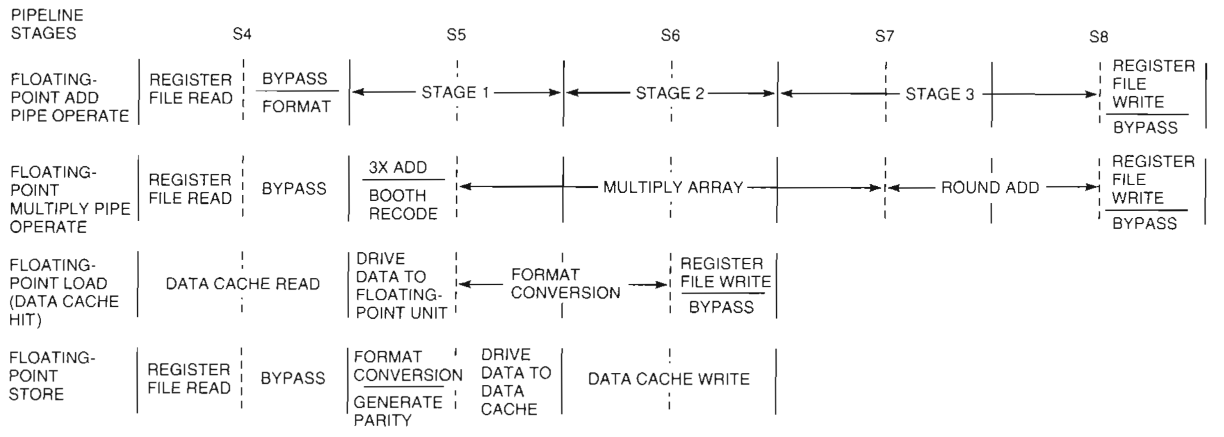


Figure 4
Floating-point Unit Pipeline

Multiply Pipeline

Multiplication is done using radix-eight Booth encoding, which requires 18 partial products to be summed.[3] The first stage of the multiply pipeline is used to create three times the multiplicand and to determine the Booth encodings. The multiplier array is composed of 14 rows of carry-save adders that perform the addition of multiplicands. The carry and sum outputs of the array are reduced by combining carry-save adders and then are passed through a half adder to facilitate rounding.

The sticky bit for multiplication is determined by summing the number of trailing zeros in both operands. The carry output from the less significant product bits is used by the round selector of the multiply pipeline to determine the correct final product.

Divider

Floating-point divide instructions issue into the add pipeline. The operands are immediately passed to the divider. Instruction issue to the add pipeline continues while a divide is in progress until the result is ready. At that point, the issue stage in the instruction unit stalls one cycle to allow the quotient to be sent to the round adder and then be written into the register file.

The divider uses a normalizing nonrestoring algorithm that determines 1 to 4 bits of quotient per cycle, averaging 2.4 quotient bits per cycle.[4] Implementation of this algorithm requires that an exact partial remainder be produced every cycle. The implementation uses a fast adder that produces its result in half of a cycle.

Memory Unit

The memory unit contains a fully associative, 64-entry, data translation buffer (DTB); an 8-KB, direct-mapped, primary data cache; a structure called the miss address file (MAF); and a write buffer. It processes load, store, and memory barrier instructions.

The write-through data cache has 32-byte blocks and 2 read ports. Its tags hold physical address data.

The memory unit receives as many as 2 virtual addresses from the integer unit each cycle. Because it has 2 read ports, the DTB can translate both virtual addresses to physical addresses and detect memory management faults. (Like the ITB, each entry in the DTB can map 1, 8, 64, or 512 contiguous 8-KB pages.)

Load instructions access the data cache and return data to the register file if there is a hit. The latency for loads that hit in the data cache is two cycles. Again, latency is reduced relative to the DECchip 21064 microprocessor where the latency is three cycles for loads that hit. The reduced latency was achieved by circuit design improvements. Reducing this latency is particularly important as issue width increases because of the frequent use of loads in programs.

For loads that miss, the physical addresses are sent to the MAF, where they wait to be sent to the C-box. Store instructions write the data cache if there is a hit; they are always placed in the write buffer, where they wait to be sent to the C-box.

Memory Unit Pipeline Structure

Virtual address calculation begins in the integer unit early in stage 4. The data cache access begins later in stage 4 and completes early in stage 5. Address translation is done in parallel with data cache access. Data cache hit is determined late in stage 5. If the access hits, the data is written to the register file (for a load access) or the cache (for a store access) in stage 6. In the case of a data cache miss, the memory access advances to pipeline stages in the C-box.

Miss Address File

The MAF consists of two sections that store data. The first section holds load misses (called DREADs) in six entries, and the other section holds instruction fetch addresses (called IREFs) in four entries. For DREADs, the MAF stores the physical address, destination register, and instruction type (integer/floating-point, 4-byte/8-byte/IEEE-S-Type/VAX-G-Type, etc.). For IREFs, the MAF stores only the physical address.

Buffered accesses in the MAF and write buffer are sent to the C-box at a peak rate of one every other cycle. DREADs have highest priority, writes have the next highest priority, and IREFs have lowest priority.

When the C-box returns data for a DREAD, the memory unit provides the destination register and instruction type information from the MAF. This information is then used to convert the data to its in-register format, to determine which registers to write, and to update the register scoreboard in the instruction unit. The DREAD entry is removed from the MAF when the second half of the data fill arrives.

The C-box returns IREF data directly to the instruction unit's cache and refill buffer. The IREF entry is removed from the MAF as soon as the command has been accepted by the C-box.

Merging Capability One key performance feature of the MAF is that it merges multiple load misses that access the same 32-byte block of memory into a single C-box DREAD request. One load instruction requests at most 8 bytes of a 32-byte memory block. As many as 4 load misses can be merged into 1 DREAD request. This improves latency and reduces unnecessary bandwidth consumption in the second-level cache.

To implement merging, the MAF merge logic detects any load miss address to a block that has already been queued in the DREAD section of the MAF. The logic then adds the new destination register to the existing request. Merging is limited to 1 load miss per naturally aligned 8-byte portion of the 32-byte block. Also, merging is permitted only for load misses with identical instruction types. The memory unit allocates a new DREAD entry in the MAF only for load misses that do not merge. The merge logic supports the peak load instruction issue rate. It can merge as many as 2 load misses per cycle into the DREAD section and can merge loads that issue together.

The MAF merge capability is an integral part of the two-level cache hierarchy design. It can reduce the rate of memory read operations from two loads per cycle in the integer pipelines to one read every other cycle in the second-level cache pipeline. By doing so, the MAF makes the full bandwidth of the second-level cache available to the program.

The MAF can hold as many as 6 DREADs that can represent as many as 21 loads. (The theoretical maximum is 24 loads; this limit is a by-product of the overflow prevention algorithm.) Requests are sent to the C-box in the order in which they were allocated in the MAF. Accesses in the second-level cache can hit underneath (behind) second-level cache misses, allowing data fills to be returned in a different order than they were sent to the C-box.

Two-level Data Cache Many workloads benefit more from a reduced latency in the data cache than from a large data cache. We considered a single-level design for a large data cache. For circuit reasons, physically large caches are slower than small caches. To achieve a reduced latency, we chose a fast primary cache backed by a large second-level cache. As a result, the effective latency of reads is better in the Alpha 21164 CPU chip than it would have been in a single-level design.

The two-level data cache has other benefits. The two-level design makes it reasonable to implement set associativity in the second-level cache. Set associativity enables power reduction by making data set access conditional on a hit in that set. The two-level design also allows the second-level cache to hold instructions, which makes a larger instruction cache unnecessary.

In addition, the two-level design was simpler. Because performance studies showed that the Alpha 21164 CPU chip should have write-back caching on-chip, the data cache in the single-level design would have been write-back. Also, because of its larger size, it would have been virtually addressed, which would have required a solution to the synonym problem. Finally, it would have been difficult to make the single large cache set-associative without adding latency. The two-level design eliminated all these issues.

Write Buffer

The write buffer contains 6 entries; each entry holds as many as 32 bytes of data and one physical address. It accumulates store instructions written to the same 32-byte block by merging them into 1 entry. It can merge 1 store instruction per cycle, matching the peak store instruction issue rate. The write buffer places no restrictions on merging until a write is sent to the second-level cache. At that time, the write buffer stops merging to that entry.

Once an entry from the write buffer has been sent to the C-box, several steps may be required to complete the write, depending on the presence of the memory block in the second-level cache and its cache coherence state. The C-box signals the memory unit upon completion of a store operation, and then the memory unit removes the corresponding entry from the write buffer.

Access Ordering

The memory unit guarantees that all memory accesses to the same address are processed in the order given by the instruction stream. This is a design problem in any nonblocking memory subsystem design. Load misses that conflict with a store, and stores that conflict with a load miss, set conflict bits that prevent the issue of the DREAD or write until all conflicts have been cleared. If a store matches a valid entry in the write buffer and cannot merge with that entry, it is allocated a new entry that is prevented from being sent to the C-box until the earlier write is completed.

Memory Barrier Instructions

The memory unit implements the memory barrier (MB) instruction by retiring all previous load misses and writes before sending the MB to the bus interface unit. The instruction unit stalls new memory instructions until the MB has been completed.

The memory unit implements the write memory barrier (WMB) instruction as follows: When the WMB is executed, the memory unit marks the last write that is pending at that time. Writes added after that time are added behind the WMB mark. They are not sent to the C-box until all writes ahead of the WMB mark are completed. Unlike the MB instruction, execution of the WMB instruction does not require any stalls in the instruction unit.

Replay Traps in the Memory Unit

The memory unit forces a replay trap if a new load or write would cause the buffer to overflow. It also forces a replay trap when a store that hits in the data cache is followed by a load to exactly the same location in the next cycle. In this case, because the store writes the data cache in stage 6, the data from the store would not yet be available to the load.

Cache Control and Bus Interface Unit

The cache control and bus interface unit or C-box contains the second-level cache and the following sub-units: the second-level cache arbiter unit (SAU), the bus interface unit sequencer (BSQ), the victim address file (VAF), the bus interface unit address file (BAF), the write buffer unit (WBU), and the system probe arbiter (SPA). Figure 5 shows the functional units of the C-box.

The C-box provides the interface to the system for access to memory and I/O. It provides full support for multiprocessor systems using a cache coherence protocol (described later in this section). It manages the second-level cache and an optional off-chip third-level cache, both of which are multiprocessor-coherent write-back caches.

The SAU arbitrates the requests for access to the second-level cache. The BSQ requests to write data fill (due to previous second-level cache misses). The VAF requests read accesses of deallocated second-level cache blocks that have been modified (called victims). The SPA requests access for external cache coherence

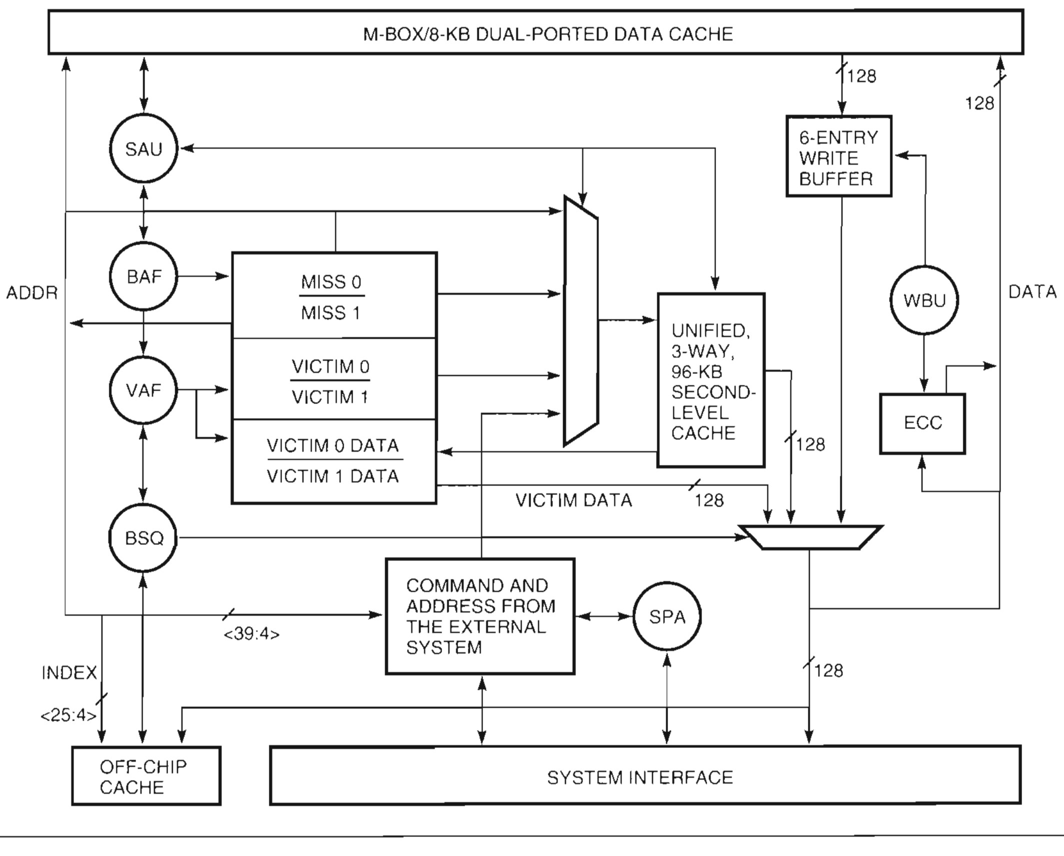


Figure 5
Functional Units of the C-box

transactions. The memory unit requests access for DREAD, IREF, and write requests. Highest priority is given to the BSQ, followed by the VAF, and then the SPA; lowest priority is given to the memory unit.

The BSQ controls data movement to and from the Alpha 21164 microprocessor. It accesses the optional off-chip third-level cache. It communicates with the system to request data that is not cached, to write back deallocated cache blocks that have been modified, to carry out coherence transactions, and to perform I/O accesses.

The VAF reads and holds victims from the second-level cache and data for memory broadcast writes, I/O writes, and external cache coherence commands that require data from the second-level cache. It has two entries for victims, each of which holds the address and data for a victim. These victims are written back to third-level cache or memory when the BSQ is idle or sooner if necessary to maintain cache coherence. These entries also hold data for memory broadcast writes and I/O writes. A separate buffer holds data for external cache coherence commands that require data from the second-level cache.

The WBU handles second-level cache writes and cooperates with other C-box subunits to maintain cache coherence.

The SPA receives cache coherence requests from the external system environment. To fulfill these coherence requests, it accesses the second-level cache and, if the off-chip cache is present, cooperates with the BSQ to access the off-chip cache. It then sends an appropriate response to the external system.

Second-level Cache and Optional Off-chip Cache

The C-box manages the on-chip second-level cache and the optional off-chip cache. Both are write-back, and both are mixed instruction and data caches. If it is present, the off-chip cache is a third-level cache. The second-level cache is 96 KB in size and is 3-way set-associative. The off-chip cache is direct-mapped and can be configured to sizes ranging from 1 megabyte (MB) to 64 MB. The off-chip cache is not set-associative because it is not feasible given pin-count constraints. The tags in both caches hold physical address data and coherence state bits for each block.

The block size for the off-chip cache is configurable to 32 bytes or 64 bytes. The second-level cache has 1 tag per 64-byte block. It can be configured to operate with 64-byte blocks or with 32-byte subblocks.

The second-level cache tags contain bits to record which 16-byte data words within the block or subblock have been modified since the block was brought on-chip. When a block or subblock is copied back to the off-chip cache, only modified 16-byte data words are transferred. This reduces the time required to write back second-level cache victims in many cases.

Transaction Handling

A maximum of 2 second-level cache misses can be queued in the BAF for external access in the off-chip cache and memory. The BAF merges read requests to 32-byte blocks within the same 64-byte block.

For simplicity, only one operation to a given second-level cache address is allowed in the BAF at a time, except when the two requests merge. A new request with a second-level cache address that matches an existing request in the BAF is aborted. Similarly, requests that require VAF entries when the VAF is full are aborted, and new requests are aborted when the BAF is full. If a request is aborted, the memory unit retries the request repeatedly until it is accepted. Accesses to second-level blocks that are partially valid because they are being filled are aborted repeatedly until the data fill completes.

Maintaining Cache Coherence

The Alpha 21164 CPU chip uses a cache coherence protocol implemented in hardware to provide full support for multiprocessor systems. The instruction cache is virtual and is not kept coherent by the hardware. (The Alpha architecture requires software to manage instruction cache coherence.) The data cache is a subset of the second-level cache. If the off-chip cache is present, then the second-level cache is a subset of the off-chip cache.

Three state bits record the coherence state of each block or subblock in the second-level cache and the off-chip cache: the valid bit, the shared bit, and the dirty bit. The valid bit indicates that the block contains valid data. The shared bit indicates that the block may be cached in more than one CPU's cache. The dirty bit indicates that the memory copy of the block is not correct and the cache block must eventually be written back. These state bits allow the following states to be encoded for a given cache block or subblock: invalid, exclusive-unmodified, exclusive-modified, shared-unmodified, and shared-modified.

The system bus interface is the coherence reference point in the system. Any request to modify the state of a block is arbitrated at this bus before the block is changed. For example, when the Alpha 21164 CPU chip must write to a block in the second-level cache that is in the exclusive-unmodified state, the BSQ sends a request to the system to change the state of the block to the exclusive-modified state. The C-box waits for the system to acknowledge the request, and then retries the write. If another processor reads the same block before the request is acknowledged, the block is instead changed to the shared-unmodified state. In that situation, the Alpha 21164 CPU chip subsequently sends a full-block memory write on the system bus that causes all other processors to invalidate their copy of the block and leaves the block in the exclusive-unmodified state in this processor.

Second-level Cache Transaction Flows

DREADs, IREFs, and writes from the memory unit access the second-level cache after winning arbitration in the memory unit and the SAU. The second-level cache is fully pipelined. Figure 6 shows an example of a read that is followed by a write as both hit in the cache.

For the read access shown in Figure 6, the pipeline stages are the following. The SAU arbitrates in stage 5; the second-level cache tag store is read in stage 6; the hit is determined in stage 7; and the requested data is read from the cache data store in stage 8 and sent on the 128-bit-wide read data bus (R-bus) in stage 9. The second half of the 32-byte block is read and sent in the next pipeline cycle. The R-bus data is received by the integer unit, the floating-point unit, or the instruction unit, depending on the access type.

For data returned to the integer unit or the floating-point unit, the data cache fill begins in stage 10 and completes in stage 11. The register file write occurs in stage 11. An instruction that is dependent on the load can begin execution in the next cycle. In this case, the load latency is eight cycles.

For the write access shown in Figure 6, the pipeline stages are the following. The SAU arbitrates in stage 5; the tag store is read in stage 6; the hit is determined, and data is sent on the 128-bit write data bus (W-bus) in stage 7; and the cache is written in stage 8. As before, the second half of the 32-byte write occurs in the next pipeline cycle.

A second-level cache miss that results in a victim provides an interesting case for discussion. Here, we must determine which set to fill and then remove the victim before data can be returned from the off-chip cache. Figure 7 shows an example of a DREAD that misses in the second-level cache, creating a victim, and then hits in the off-chip cache. The example shown is the fastest possible. In this case, the BSQ is idle so the BAF is bypassed and the address is sent immediately to the off-chip cache. The access time for the off-chip cache is four CPU cycles.

As shown in Figure 7, the DREAD wins arbitration in stage 5, and the miss is detected in stage 7. The set picked by the random replacement algorithm contains modified data (a victim). Since the block size in the second-level cache is 64 bytes, two 32-byte victim read sequences are needed to copy the entire victim into the on-chip victim buffer. The two victim reads arbitrate at high priority to ensure that the victim is copied before the data fills from the off-chip cache overwrite the locations.

The Alpha 21164 CPU chip begins sending the off-chip cache address in stage 8 (because of BAF bypass, as described above). The tag and data are clocked into the Alpha 21164 chip at the beginning of stage 12. The BSQ arbitrates speculatively for a single cycle on the second-level cache pipeline to reserve a cycle on the R-bus. That cycle is used to send the data from the off-chip cache to the execution units and data cache.

If the access hits in the off-chip cache, the BSQ arbitrates to fill the second-level cache. The fill transaction takes a single cycle in the pipeline to write the tag store in stage 6 and the data store in stage 8.

The second victim read sequence occurs after the first data fill. Because of this, the first victim read sequence always reads the data location overwritten by the first data fill.

PALcode

The Alpha architecture defines the privileged architecture library code (PALcode) as a set of software routines that interface an operating system to a specific Alpha implementation. PALcode presents the operating system with an architecturally defined interface that is the same in all implementations even though the underlying hardware designs can be very different. PALcode currently exists to interface the Alpha 21164 microprocessor to the Windows NT, Digital UNIX (formerly DEC OSF/1), and OpenVMS operating systems.

When the processor is executing PALcode, it is in PAL mode. PAL mode is entered upon execution of the CALLPAL instruction and upon the occurrence of interrupts, exceptions, and certain kinds of traps. The PALcode entry point is a hardware dispatch to a location that is determined by the entering event. In PAL mode, instructions are fetched from physical memory without address translation. Also, five PAL support instructions are enabled that give access to all hardware registers and special load/store access to virtual and physical memory. PAL mode is exited by executing a PAL instruction called HW_REI.

To meet performance goals, a number of PAL features are included in the Alpha 21164 microprocessor. For example, the integer register file contains eight shadow registers that map over R8 through R14 and R25 in PAL mode. Although this overmapping

PIPELINE STAGE	S5	S6	S7	S8	S9	S10
READ	ARB	TAG	HIT	RAM	R-BUS RAM	R-BUS
WRITE	ARB	TAG	HIT/W-BUS	RAM W-BUS	RAM	

Figure 6
Second-level Cache Read/Write Flow

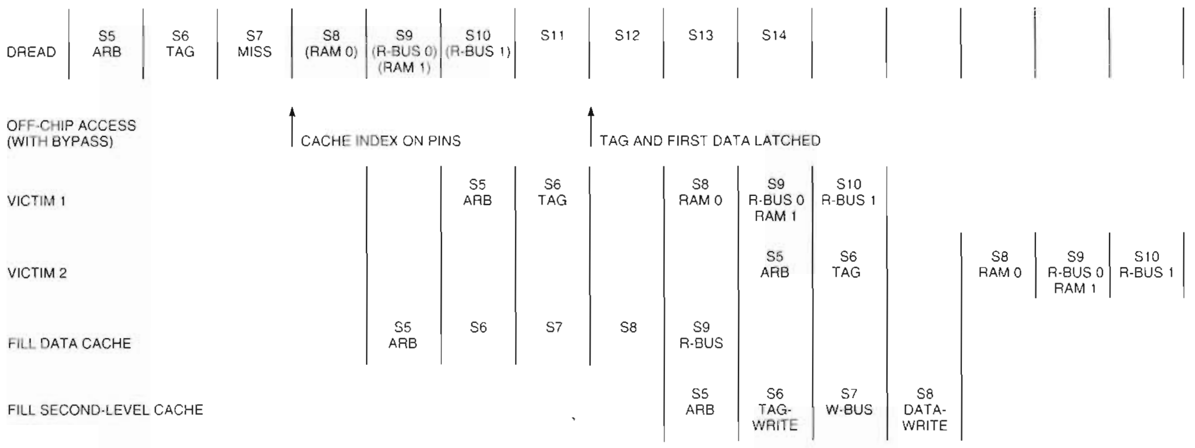


Figure 7
Second-level Cache Miss Sequence with Fastest Fill Possible

is normally enabled in PAL mode, it can be disabled through a hardware control register. This speeds PALcode entry and exit, because PALcode is free to use these registers without saving and restoring state. The shadow register mapping is designed to avoid overmapping any register used to pass data from the operating system to PALcode or vice versa.

Several of the operating systems that run on Alpha systems access memory management page tables through virtual memory.[5] The Alpha 21164 microprocessor contains hardware to speed processing of the PALcode for translation buffer miss. These PALcode routines access virtually mapped page tables. The hardware calculates the virtual address of the page table entry (PTE) based on the miss address and the address of the page table base. This eliminates the instruction sequence required for this calculation. PALcode then executes a load instruction to this virtual address to fetch the required PTE. This load is performed using a PAL instruction that signals a virtual PTE fetch. If this load misses in the DTB, a special PALcode trap routine is dispatched to fill the DTB using a multilevel, physical-address access method. After that, the original virtual PTE read is restarted and will succeed.

Testability Features

The Alpha 21164 microprocessor incorporates several testability features. Some enhance chip test, and some features provide useful module test capability.[6]

Repairable On-chip RAMs

The Alpha 21164 microprocessor requires large random-access memory (RAM) arrays for its on-chip caches. To improve yield, the instruction and data cache arrays have spare rows and the second-level cache has spare rows and spare columns.

A working instruction cache is necessary for most chip test programs. Consequently, it is automatically tested by built-in self-test (BiSt) and automatically repaired by built-in self-repair (BiSr). During wafer probe, the test result is serially shifted off-chip for permanent repair by laser. Upon chip reset, BiSt of the instruction cache occurs automatically, but BiSr is not necessary if the chip has been repaired.

The data cache and second-level caches are tested by programs loaded into the instruction cache during wafer probe. These programs condense the test results and write them off-chip to be captured by the tester for subsequent laser repair.

Chip Logic Testability

To enhance core logic testability, the Alpha 21164 microprocessor contains dual-mode registers that can operate as scan registers or as linear feedback shift registers (LFSRs). The scan mode is used for initialization, for scanning out signatures, and for debugging. The LFSR mode is used for manufacturing test.

Module Manufacturing

The Alpha 21164 microprocessor implements the IEEE 1149.1 standard for supporting testing during module manufacturing. The supported instructions are EXTEST, SAMPLE/PRELOAD, BYPASS, CLAMP, and HIGHZ.

Summary

The internal organization of the Alpha 21164, a new, high-performance Alpha microprocessor, has been presented. Mechanisms designed to enhance the CPU's performance combined with the CPU's clock speed of 300 MHz produce an extremely high-performance microprocessor. First silicon of the Alpha 21164 CPU chip was produced in February 1994, and three different operating systems were successfully booted on the first-pass silicon. The part became commercially available in January 1995. It achieved the performance level of 345 SPECint92 and 505 SPECfp92 (estimated), a performance level unmatched by commercially available microprocessors.

Acknowledgments

The authors wish to acknowledge George Chrysos, Robert Cohn, Zarka Cvetanovic, Kent Glossop, Lucy Hamnett, Steve Hobbs, John Shakshober, and Paula Smith for their work in producing to the estimates of SPEC benchmark performance quoted in this paper.

References

1. D. Dobberpuhl et al., "A 200-MHz 64-bit Dual-issue CMOS Microprocessor," *Digital Technical Journal*, vol. 4, no. 4 (Special Issue 1992): 35–50.

2. W. Bowhill et al., "Circuit Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU," *Digital Technical Journal*, vol. 7, no. 1 (1995, this issue): 100–118.

3. E. Swartzlander, ed., *Computer Arithmetic* (New York: Dowden, Hutchinson, and Ross, 1980).

4. O. MacSorley, "High-speed Arithmetic in Binary Computers," *Proceedings IRE*, vol. 49 (1961): 67–91.

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

6. D. Bhavsar and J. Edmondson, "Testability Strategy of the Alpha 21164 Microprocessor," *International Test Conference* (October 1994): 50–59.

Biographies

John H. Edmondson
John Edmondson is a consultant engineer in Digital Semiconductor. He was the architecture leader of the design team for the Alpha 21164 microprocessor. Previous to that work, he was a member of the design team for the VAX 6000 Model 600 microprocessor. Prior to joining Digital in 1987, John worked at Canaan Computer Corporation and Massachusetts General Hospital. John received a B.S.E.E. from the Massachusetts Institute of Technology in 1979.

Paul I. Rubinfeld
Paul Rubinfeld was the engineering manager on the Alpha 21164 microprocessor project. During the last 16 years at Digital, he has worked on VAX and PDP-11 CPU development projects and a single-instruction, multiple-data, massively parallel processor system. Paul received a B.S. and an M.S. in electrical engineering from Carnegie Mellon University, where he helped build the Cm* multiprocessor. Paul is a senior engineering manager within Digital Semiconductor.

Peter J. Bannon
Pete Bannon is a consulting engineer in Digital Semiconductor. He has participated in the design or verification of several microprocessor chips and was a member of the Alpha 21164 architecture team. He joined Digital in 1984 after receiving a B.S. (special honors) in computer system design from the University of Massachusetts. He holds three patents for VAX CPU design and has filed six patent applications for the Alpha 21164.

Bradley J. Benschneider
Brad Benschneider is a principal hardware engineer in Digital Semiconductor. He was responsible for designing various sections of the memory management unit on the 21164, as well as defining the latching methodology for the chip. He is currently leading the implementation effort of the memory management unit for the next-generation Alpha CPU. Since joining Digital in 1987, he has contributed to several custom chip designs in the VAX 6000 family and the early Alpha implementations. He received a B.S.E.E. from the University of Cincinnati, has one patent, and has coauthored four papers.

Debra Bernstein
Debra Bernstein is a consultant engineer in Digital Semiconductor. Her work has spanned the areas of architecture, performance, simulation, logic design, firmware, PALcode, verification, and hardware debug for four generations of Digital CPUs. Deb is currently working on the hardware and software components of a PC-based multimedia solution. She received a B.S. in computer science (1982, cum laude) from the University of Massachusetts.

Ruben W. Castelino
Before receiving a B.S.E.E. from the University of Cincinnati in 1988, Ruben Castelino was a co-op student at Digital working on a chip set for the VAX 6000 Model 200. Currently a senior hardware engineer in Digital Semiconductor, he was a codesigner of the cache control and bus interface unit for the Alpha 21164 CPU. Prior to that, he worked on the instruction fetch, decode, and branch unit for the NVAX chip and performed implementation work for the NVAX virtual instruction cache. Ruben is currently a codesigner of the cache control and bus interface unit for a new Alpha microprocessor.

Elizabeth M. Cooper
Beth Cooper received B.S. degrees (summa cum laude) in electrical engineering and computer science from Washington University in St. Louis (1985) and an M.S. degree in computer science from Stanford University (1995). She is a member of Eta Kappa Nu. She joined Digital in 1985 and has worked on the implementations of several CMOS VAX and Alpha CPUs since then. Beth was the lead cache designer on the Alpha 21164 microprocessor. She is currently a principal hardware engineer in the Palo Alto Design Center.

Daniel E. Dever
Since joining Digital in 1988, Dan Dever has worked on the design and logic verification of the CMOS VAX microprocessors as well as the 21064 and 21164 Alpha microprocessors. Dan is currently involved in the design of the memory management unit for the next-generation Alpha microprocessor. He received a B.S. in electrical engineering from the University of Cincinnati in 1988.

Dale R. Donchin
Dale Donchin is an engineering manager and technical contributor in Digital Semiconductor. He designed several circuits related to the clock and cache and contributed to and led CAD tool use for the Alpha 21164 CPU. He is presently performing these duties for the development of the next-generation Alpha microprocessor. Dale joined Digital in 1978 and was previously a development manager in the RSX Operating System Group. Dale holds a B.S.E.E. (1976, honors) and an M.S.E.E. (1978) from Rutgers University College of Engineering and is a member of IEEE and ACM.

Timothy C. Fischer
Tim Fischer is a senior hardware engineer in Digital Semiconductor. He is currently working on the instruction issue logic for the next-generation Alpha microprocessor. Prior to this, Tim worked on the design of the Alpha 21164 floating-point unit, the NVAX+ bus interface unit, and the NVAX clocks and patchable control store. He has coauthored several papers. Tim joined Digital in 1989 after receiving an M.S. in computer engineering from the University of Cincinnati.

Anil K. Jain
Anil Jain, a consulting engineer in Digital Semiconductor, led the implementation of the external interface unit on the Alpha 21164 microprocessor. Prior to this, he was the project leader for the floating-point unit on the NVAX microprocessor. He also made technical contributions on the CVAX microprocessor and on device modeling of Digital's first CMOS process. Anil received a B.S.E.E. from Punjab Engineering College (1978) and an M.S.E.E. from the University of Cincinnati (1980). He holds three patents.

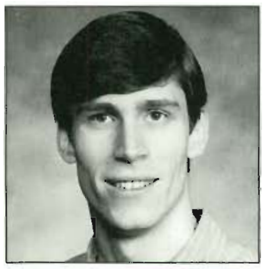
Shekhar Mehta
Shekhar Mehta is a senior hardware engineer in Digital Semiconductor's High Performance Computing Group. He designed the miss address file on the memory subsystem of the Alpha 21164 CPU and was responsible for the electromigration checks of the chip. He is currently leading the design of the caches on a future Alpha microprocessor. Before joining Digital in 1988, Shekhar was an engineer at Larsen & Toubro, Bombay, India. He received an M.S.E.E. from the University of Wisconsin at Madison (1988).

Jeanne E. Meyer
Since joining Digital in 1989, Jeanne Meyer has worked on the implementation, behavioral modeling, and logic verification of several microprocessor chips. In her work on the Alpha 21164 CPU, she was responsible for PALcode verification, maintenance, and support. She also contributed to the microarchitecture definition and behavioral model of the chip's memory management unit. She is currently leading the design of the memory management unit for a new Alpha microprocessor. Jeanne received a B.S.E.E. (summa cum laude, 1982) from the University of Cincinnati. She holds two patents.

Ronald P. Preston
Ronald Preston is a principal engineer in Digital Semiconductor. Since joining Digital in 1988, he has worked on the design of several microprocessors and was the implementation leader for the instruction unit on the Alpha 21164. Ron was also responsible for the architecture and implementation of the issue/bypass/scoreboard logic. Ron is the coauthor of several articles on hot carrier analysis of CMOS circuits. He received a B.S.E.E. in 1984 and an M.S.E.E. in 1988, both from Rensselaer Polytechnic Institute. Ron is a member of Eta Kappa Nu and IEEE.

Vidya Rajagopalan
Vidya Rajagopalan is currently with Quantum Effect Design Inc. Prior to joining QED, she was a member of Digital's Semiconductor Engineering Group, where she worked on the Alpha 21164 and 21064 microprocessor designs. Vidya received an M.S. in electrical engineering from the University of Maryland, College Park, and a B.E. from Visvesvaraya Regional College of Engineering, Nagpur, India.

Chandrasekhara Somanathan
Chandrasekhara Somanathan received an M.B.A. from Northeastern University in 1994, an M.S. in computer science from the Rochester Institute of Technology in 1984, and a B.S. in electrical and electronics engineering from BITS, Pilani, India in 1982. While at Digital from 1985 to 1994, he designed the cache controller unit of the Alpha 21164 RISC microprocessor, and the floating-point and cache controller units of the VAX 6000/400 CISC microprocessor; he also developed Digital's MOS timing analysis CAD software. He is currently with HaL Computer Inc., developing high-performance SPARC RISC microprocessors.



Scott A. Taylor
Scott Taylor joined Digital in 1993 after receiving a B.S. degree in electrical engineering from the University of Illinois. He was involved with the functional verification of the memory, cache control, and external interface units on the Alpha 21164 microprocessor. Scott has also worked on CPU test pattern generation and debug as well as on-chip cache repair strategies. He is currently contributing to the verification of the next generation of Alpha high-performance microprocessors.

Gilbert M. Wolrich
A consultant engineer in Digital Semiconductor, Gil Wolrich was the leader and architect for the floating-point unit on the Alpha 21164 chip. He received a B.S.E.E. from Rensselaer Polytechnic Institute and an M.S.E.E. from Northeastern University.

Michael Kantrowitz
Lisa M. Noack

Functional Verification of a Multiple-issue, Pipelined, Superscalar Alpha Processor — the Alpha 21164 CPU Chip

Digital's Alpha 21164 processor is a complex quad-issue, pipelined, superscalar implementation of the Alpha architecture. Functional verification was performed on the logic design and the PALcode interface. The simulation-based verification effort used implementation-directed, pseudorandom exercisers, supplemented with implementation-specific, hand-generated tests. Extensive coverage analysis was performed to direct the verification effort. Only eight logical bugs, all unobtrusive, were detected in the first prototype design, and multiple operating systems were booted with these chips in a prototype system. All bugs were corrected before any 21164-based systems were shipped to customers.

The Alpha 21164 microprocessor is a quad-issue, superscalar implementation of the Alpha architecture. The CPU chip required a rigorous verification effort to ensure that there were no logical bugs. World-class performance dictated the use of many advanced architectural features, such as on-chip virtual instruction caching with seven-bit address space numbers (ASNs), an on-chip dual-read ported data cache, out-of-order instruction completion, an on-chip three-way set-associative write-back second-level cache, support for an optional third-level write-back cache, branch prediction, a demand-paged memory management unit, a write buffer unit, a miss-address file unit, and a complicated bus interface unit with support for various CPU-system clock ratios, system configurations, and third-level cache parameters.[1]

Functional verification was performed by a team of engineers from Digital Semiconductor whose primary responsibility was to detect and eliminate the logical errors in the Alpha 21164 design. The detection and elimination of timing, electrical, and physical design errors were separate efforts conducted by the chip design team.[2]

Extensive functional verification prior to releasing the first-pass design to the manufacturing process is a common technique used to ensure that time-to-market goals are met for complex processors. Increasingly, these verification efforts are relying on pseudorandom test generation to improve the quality of the verification effort. These techniques have been in use at Digital for more than seven years and are also used elsewhere in the industry and in academia.[3-6] This paper describes a functional verification effort that significantly extended pseudorandom testing with extensive coverage analysis and some hand-generated tests to produce working first-pass parts.

Goals

The verification team had several key goals. Goals for first-pass silicon included ensuring that the first prototypes could boot the operating system and providing a vehicle for debugging of system-related hardware and software. An additional goal was to execute a test to check every block of logic and every function in the chip to ensure that no serious functional bugs remained. The goal for second-pass silicon was to be bugfree so that these chips could be shipped to customers for use in revenue-producing systems. Secondary goals included assisting in the verification of Privileged Architecture Library code (PALcode) and keeping manufacturing test patterns in mind when creating the verification environment and writing tests.

Modeling Methodology

Several different model representations of the Alpha 21164 CPU were developed for testing prior to prototypes. The verification team primarily used a register-transfer-level (RTL) model of the Alpha 21164 CPU chip. This model accurately represented the detailed logic of the design and delivered very high simulation performance.

Modeling Environment

The design team wrote the RTL model in the C programming language. The model represented all latches and combinatorial logic of the design and was accurate to the clock-phase boundary. The C programming language was chosen because C provides the speed and flexibility needed for a large-scale design. Digital's CAD group designed a user interface for access into the RTL model of the Alpha 21164 CPU. The C command line interface (CCLI) allowed access into the variables used to define signals and to the routines that represented the actual design. It provided the ability to create binary traces of signals for postprocessing analysis and debugging. A standard set of macro-instructions simplified bit manipulation of signals with arbitrary widths.

The use of C also allowed the team to simulate portions of the gate-level design in the structural simulator, CHANGO, and to perform cycle-by-cycle comparisons with various states in the RTL model. These simulations, called shadow-mode simulations, were fully utilized for testing the various functional units of the chip.

Pseudosystem Models

The verification team developed several models to interface to the Alpha 21164 CPU RTL model and to allow testing of interactions with pseudosystems to occur. The C language provided a level of flexibility in the creation of these models that was not available on previous verification projects. One area in which this flexibility was fully utilized was in the formation of a sparsely populated memory model. By using a dynamic tree data structure rather than a static array, the cache, duplicate tag store, and memory system models could be written to support the full range of 64-bit addressing. Hence, tests could be created to use any set of addresses without restrictions. In addition, comparisons with the reference model could be drawn from the entire contents of memory. This significantly enhanced the ability to detect possible errors in the design.

The verification engineers created a system model (the X-box) to simulate transactions on the pin bus. The X-box model provided a means to mimic the real system behavior that the Alpha 21164 CPU would encounter when used with a variety of different platforms. The team used C to develop an X-box model that could be connected to every possible configuration and mode setting of the Alpha 21164 CPU chip. This allowed all modes of the Alpha 21164 CPU to be tested with a single, multipurpose system interface model. The X-box also performed many of the checks needed to ensure the proper operation of the system bus.

Strategy

The verification strategy employed multiple techniques to achieve full functional verification of the Alpha 21164 chip. The primary technique used was pseudorandom exercisers. These programs generated pseudorandom instruction sequences, executed the sequences on both the 21164 model and a reference model, and compared the results. A second major technique used focused, hand-generated tests to cover specific areas of logic. Other methods consisted of design reviews, executing existing tests and benchmarks, and a few static analysis techniques. Figure 1 shows the general flow for a single simulation.

This strategy was deployed in three parts: the try-anything phase, the test-planning phase, and the structured completion phase. Devising a test plan was not the first step. During the early stage of the project, the primary goal was to stabilize the design as quickly as possible. Any major bug that would have had an impact on the architectural definition of the chip was uncovered. Circuit design and layout could then commence without fear of major revisions later. If time had been spent structuring detailed test plans, less time would have been given to actual testing, and at this point in the design, careful thought was not needed to find bugs.

The main purpose of the try-anything phase was to exercise as much functionality of the design as possible in the shortest time in order to stabilize the design quickly. This phase began even before the RTL model was ready, with the construction of the pseudorandom exerciser programs. The pseudorandom exercisers and the RTL model were debugged together. This produced an atmosphere of intensity and challenge in which everyone was required to interact constantly to help identify the source of problems. This had a secondary benefit of bringing the design and verification teams closer together.

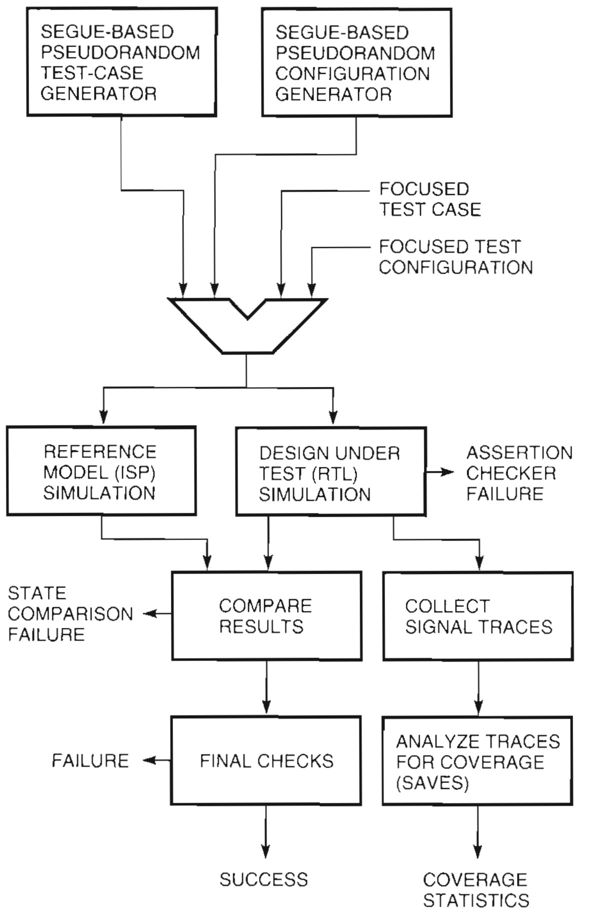

SEGUE-BASED PSEUDORANDOM TEST-CASE GENERATOR
SEGUE-BASED PSEUDORANDOM CONFIGURATION GENERATOR
FOCUSED TEST CASE
FOCUSED TEST CONFIGURATION
REFERENCE MODEL (ISP) SIMULATION
DESIGN UNDER TEST (RTL) SIMULATION
ASSERTION CHECKER FAILURE
STATE COMPARISON FAILURE
COMPARE RESULTS
COLLECT SIGNAL TRACES
FAILURE
FINAL CHECKS
ANALYZE TRACES FOR COVERAGE (SAVES)
SUCCESS
COVERAGE STATISTICS

Figure 1
Design Verification Test Environment

Once the design stabilized and the bug rate declined, the design team began focusing on circuit design and layout, and the verification team took a step back and created a test plan. The purpose of the test plan was to ensure that the verification team understood what needed to be verified. The test plan provided a mechanism for reviewing what would be tested with the design team. The joint review ensured that the verification team did not miss important aspects of the design. The test plan also allowed a way for the design team to raise issues around specific problem areas in the design or areas that employed special logic that were not obvious from the specification. Finally, the test plan provided a means for scheduling and prioritizing the rest of the verification effort.

The test plan consisted of a description of every feature or function of the design that needed to be tested, including any special design features that might require special testing. It did not describe how the test would actually be created. Past experience had indicated that test plans describing the specific sequence of instructions needed to test chip features quickly became outdated. Instead, the test plan focused on the "what," not the "how."

The final verification step was the structured completion phase. During this time, each item from the test plan was analyzed and verified. The analysis consisted of deciding which mechanism was appropriate for covering that particular piece of the design. This might consist of a focused test, a pseudorandom exerciser with coverage analysis, or an assertion checker. As the verification of each item was completed, a review was held with the design and architecture teams to examine what was verified and how it was done. In this way, any problems with the verification coverage were identified.

Test Stimulus

Both focused and pseudorandom exercisers were used during the verification of the Alpha 21164 chip. More than 400 focused tests were created during the verification effort, covering a wide variety of chip functions. Six different pseudorandom exercisers were used. One was a general-purpose exerciser that provided coverage of the entire architecture. Each of the other five exercised a specific section of the chip in a pseudorandom way.

The one general-purpose exerciser used was provided by a separate group and generated pseudorandom streams of instructions, data, and chip state. Its focus was at the architectural level and generated pseudorandom stimulus that would work on any implementation of the Alpha architecture.

Almost all focused design verification tests (DVTs) were written using Alpha assembly code. This provided the right level of abstraction to avoid the need to toggle ones and zeros directly on each pin, yet allowed specific control over the timing of transactions and instruction sequences that would not be possible from a compiled language. The macro-preprocessor feature of the Alpha macro-assembler was used heavily. This allowed the assembly-level programs to be constructed in a modular manner.

Pseudorandom Testing

Pseudorandom testing offers several advantages in the verification of increasingly complex chips. These include producing test cases that would be time-consuming to generate by hand, and providing the ability to generate multiple simultaneous events that would be extremely difficult to think of explicitly.

Exercisers

In support of the pseudorandom testing strategy, various exercisers were created that focused on different aspects of the chip. The following areas were targeted explicitly:

- Branching
- Data-pattern-dependent transactions
- Floating-point unit
- Traps
- Cache and memory transactions

Fundamentally, each exerciser was the same. The exerciser would create pseudorandom assembly-language code, run the code on the model under test and a reference model, collect results from each, and compare the results from both model runs. Any errors or discrepancies were then reported to the user.

The reference model used, called the ISP model, was a very high-level abstraction of the Alpha architecture written in the C language. The core of this model was created during the design of the 21064, the first Alpha processor. It was modified slightly to include Alpha 21164 specific features such as internal register definitions. The ISP model integrated the same sparsely populated memory model used in the pseudosystem model in such a way that the freedom in creating addresses could be duplicated.

SEGUE, a text generation/expansion tool, was used extensively to create pseudorandom code and configurations. Each exerciser used SEGUE templates to generate code. Variables were passed to the SEGUE templates that would determine what percentage of certain events or instructions would occur in the resultant code. Users would vary the percentages and create additional templates to target their exercisers to certain portions of the chip. An exerciser could focus only on loads and stores, or templates could be created that would generate trapping code. The verification engineers had the flexibility to create whatever code was needed. The verification engineers worked closely with the designers to understand the details of the logic. As a result, cases could be generated that would thoroughly test the functions being designed into the Alpha 21164 CPU chip.

Configuration Selection

Each test, either pseudorandom or focused, also made use of a configuration control block (CCB) parameter file. The CCB was used to set up the type of system that would be emulated for a given simulation. The parameter file consisted of variables that could be weighted to make certain system events occur or to cause certain configurations to be chosen. Once again, SEGUE scripts were utilized to create the command files that controlled these events. Examples of the type of events that could be chosen were single-bit error-correcting code (ECC) errors, interrupts, the presence of an external cache, the ratio between the system clock and the CPU internal clock rate, cache size and configuration, and other bus-interface timing events. These and other events were varied throughout the course of the project to ensure that the chip could be run in real systems using any given configuration.

The configuration chosen was guided through the use of a parameter file that contained various parameters and weightings to be utilized by SEGUE. Once a configuration was chosen using the parameter file, it was processed to produce two files used in the simulation. The first was a CCLI control file used to set up state internal to the pseudosystem-level model. The second file was loaded into the memory model to be used by the DVT and to provide information accessible through assembly code regarding the configuration type.

Simulation

Once the pseudorandom code and configuration had been generated, the test was loaded into the model under test or into the ISP model to use as the stimulus. A DVT loader was created for both models that would interpret selected data in the CCB and determine the memory locations where the test should be located. The additional information encoded in the CCB included whether the test ran in I/O, where handlers should be placed, and what page mapping was used.

After a DVT was loaded, the simulation would start. A PALcode reset handler was executed first. It read information from the CCB and loaded various registers with the configurations specified. The DVT was executed after the PALcode completed.

Capturing Random Events

In some cases, pseudorandom exercisers were used to capture events that were unlikely to occur and that would have been difficult to obtain by a focused test. By using a new tool (called FIGS), engineers were able to use the pseudorandom exercisers and postprocessing to look for events that were needed to achieve coverage of the various functions in the F-box. When the event occurred, the event could be saved and re-created for future regression testing.

Correctness Checking

A variety of mechanisms were used for checking whether the model behaved correctly. Some hand-crafted tests had comparisons built-in to verify that they generated the expected answer. This self-checking mechanism, however, is difficult to include with pseudorandom testing. Three categories of checking mechanisms were developed that could work with pseudorandom or focused tests. These were checks performed during simulation of a model, postsimulation checks done automatically every time a model completes executing, and test-specific postsimulation checks. In all cases, adjusting the checking mechanisms to eliminate reporting false errors was important to keep the debugging time low.

The RTL model was augmented with a wide variety of built-in assertion checkers. These were active any time the model was run; they verified that various assertions and rules of behavior were not violated at any time during the test execution. Assertion checkers ranged from the simple to the complex and were added to the model by both the design and verification teams. Some assertion checkers were added as the initial model was coded, and others were added as needed to ensure that certain situations did not occur. Examples of simple assertion checkers include watching for a transition to an illegal state in a state machine, or watching for the select lines of a multiplexer (MUX) to choose an unused MUX input. More complex assertion checkers were used that required explicit knowledge about illegal sequences. For example, the system bus had a complicated set of checkers attached to it that checked for violations of the bus protocol.

When a test completed executing on the model, several end-of-run checks were done automatically. One simple check was to verify that the test reached its normal completion point and had not ended prematurely. Complete cache coherency checks were performed to ensure that all three levels of cache contents were consistent with the memory image.

A variety of very powerful end-of-run checks were used. These compared the results of running a test on the model and on the ISP model. Information about the state of the model was saved while the test was executing and then compared with its equivalent from the ISP model. State that was compared in this way included a trace of the program counter (PC), a trace of the updates made to each architectural register, and the final memory image upon completion of a test.

The main problems encountered with this technique were due to inconsistencies between the ISP model and the Alpha 21164 design. The ISP model was used across multiple Alpha design projects. It provided architecturally correct results but had no concept of timing, pipelining, or caching. Several features of the Alpha 21164 implementation were difficult to verify with this reference machine.

In the Alpha architecture, arithmetic traps are imprecise, in that they might not be reported with the exact PC that caused them. Since the ISP model had no concept of timing, it reported traps at a different time than the real design. Thus, the checking mechanisms needed to be intelligent enough to take this possibility into account. Arithmetic traps also presented a problem because the destination register of certain types of traps is unpredictable after a trap occurs. Combined with the imprecise nature of traps, unpredictable values could propagate to other registers, making comparison against the reference machine difficult. Normally, certain software conventions would be followed to control these aspects of the architecture. To achieve the full benefit from pseudorandom testing, however, no restrictions were placed on which registers or instruction sequences could be used. Instead, an elaborate method was devised for tracking which registers were unpredictable at any given time. This information was then used to filter false mismatches.

Optional checks made on a per-test basis could be viewed as more complicated assertion checks. These were performed by tracing internal signals. The specific signals to trace were selected based on the particular postprocessing to be done. Then, by using a library of routines (called SAVES) to simplify accessing and manipulating these signal traces, particular interactions and protocols were verified. These could be viewed as assertion checks, but they were more complicated than the built-in variety. One example involved representing the behavior of a large section of the design as a single, complicated state machine. The behavior of this state machine could be compared with the I/O behavior of the actual design section. Another example was the representation of the branch-prediction algorithm in a more abstract form than the actual model. The behavior of the abstract algorithm was compared with the behavior of the model itself.

Coverage Analysis

The primary difficulty with functional verification is that it is virtually impossible to know when the verification effort is complete. Completing a predetermined set of tests merely indicates that the tests are complete, not that the design has been fully tested. Monitoring the bug rate provides useful information, but a low bug rate might indicate that the testing is not exercising the problem areas. To alleviate this problem and provide increased visibility into the completeness of the verification effort, extensive coverage analysis of the focused tests and pseudorandom exercisers was done. Two types of coverage checking were used: information gathered while a model was executing, and information gathered by postprocessing signal traces.

While a model was executing, information was being stored about the occurrence of simple events. For example, a record was kept on the number of times the machine issued instructions to four pipes simultaneously, the number of times the translation buffers filled up, or the number of times stalls occurred. Since the chip operated in random configurations, a record was also kept about the configuration information such as the B-cache size and timing selected, the system interface options, and timing. At the end of every model run, this recorded information was written to a database to collect statistics across multiple runs.

In addition to these simple coverage checks, more elaborate coverage analysis was done through postprocessing. By using the SAVES library, signal traces were collected while the model was executing; these were later analyzed for the specific occurrence of predefined events. The events were composed of complicated timing relationships among signals. Often, two-dimensional matrices were created, in which each axis of the matrix represented a list of events. Thus, the occurrence patterns of every event in one list could be visualized happening with every event in the second list. For example, it was verified that every type of system command (read, invalidate, set-shared, etc.) occurred followed by every type of bus response (ACK, NOACK, etc.).

Automatic coverage-checking methods were also used. The most common was a state machine coverage analyzer. It was a goal to verify that every state/arc transition in every state machine was being exercised. Programs were automatically generated to search the trace files for these transitions and record the information about what was and was not covered. This concept was extended to sections of the chip that were not designed as simple state machines. As described above, one large section of the design was represented as a single, monolithic state machine to provide an independent reference for the correct outputs of the section. This conceptual state machine was processed through the coverage analysis tool. Although the transitions that were checked did not map directly to the physical design, they did provide an excellent indication of how well that section of the design had been tested.

The trace analysis tools could accumulate data across multiple simulation runs. The data was analyzed periodically, and areas that were lacking coverage were identified. This allowed the identification of trends in the coverage and provided an understanding as to how well the pseudorandom exercisers were exercising the chip. With this insight, pseudorandom exercisers were modified or new focused tests were created to improve the test coverage. Running pseudorandom exercisers with coverage analysis proved to be a very powerful technique in functional verification.

Bug Trends

During the Alpha 21164 CPU verification effort, more than 600 bugs were logged and tracked before first-pass parts were manufactured. Figure 2 shows the bug rate achieved as a function of time for the duration of the project. To track bugs, an action tracking system was set up. Tracking of bugs started after all the subsections of the RTL-level model had been integrated and a small subset of tests was run successfully. Since many areas of the model were ready before others, the action tracking system does not represent all the issues raised. However, it is interesting to look at the trends presented by the data.

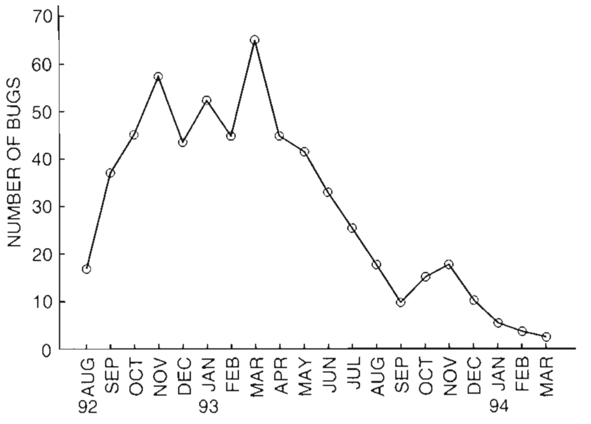


Figure 2
Bug Rate as a Function of Time

The first trend to consider is the effectiveness of the pseudorandom and focused efforts. As shown in Figure 3, more than half the bugs were found using pseudorandom techniques. Furthermore, one-third of the bugs found by the focused effort were in the error-handling functionality of the design, which had poor pseudorandom test coverage.

Bugs were thought to have been introduced in a variety of ways. Figure 4 shows the breakdown of the causes of bugs. The majority occurred in implementing the architectural ideas that were decided upon for the project.

Figure 5 shows the various detection mechanisms that were used to detect bugs. As in the past, assertion checkers placed in the design to quickly detect when something is not correct are the most successful.

Results and Conclusions

As of September 1, 1994, eight logical bugs were found in the first-pass Alpha 21164 CPU design. Only one of these impacted normal system operation, but it did not occur very often. The first two issues were

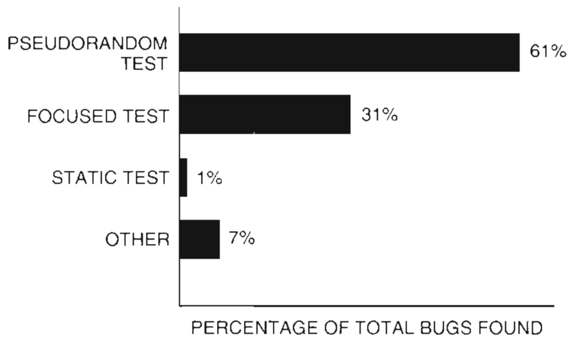


Figure 3
Effectiveness of Class of Test

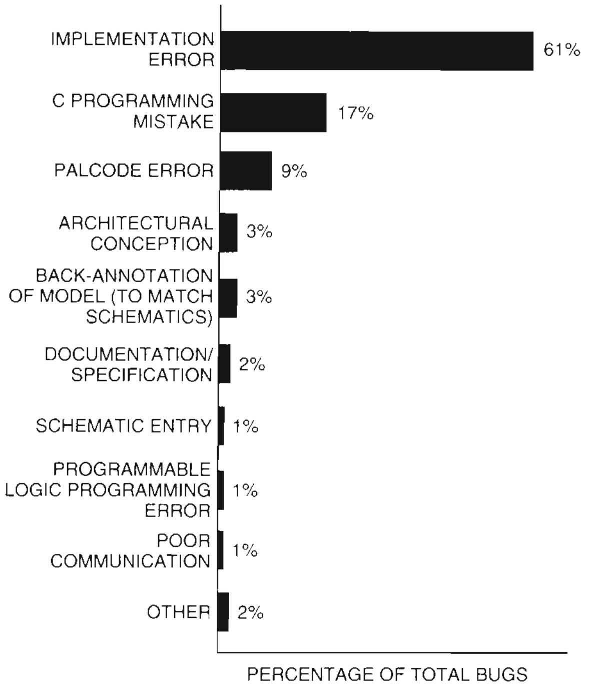


Figure 4
Introduction of Bugs

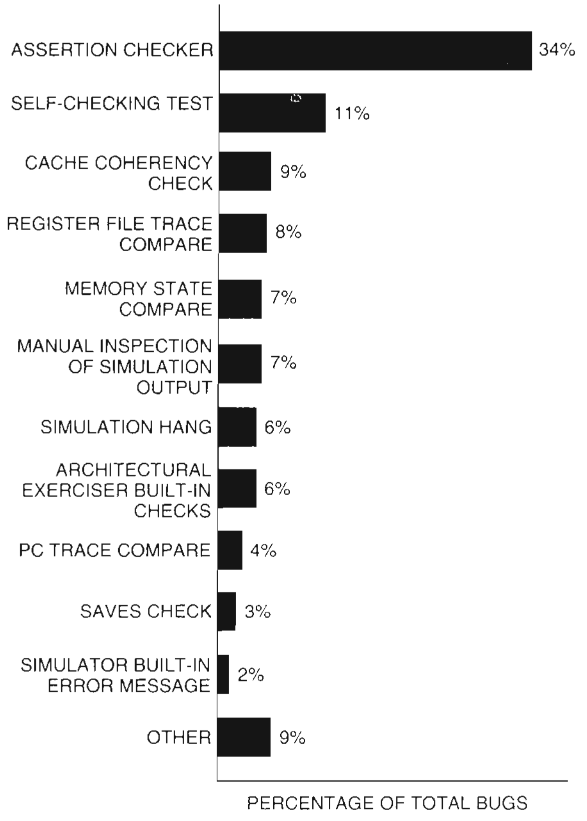


Figure 5
Effectiveness of Bug Detection Mechanisms

found while debugging test patterns on the tester; the third was a variation on a known restriction; the fourth occurred in a rare prototype system configuration that was found through pseudorandom simulation testing (which had continued even after the design was released to manufacturing); the fifth was a race condition between two events that rarely were stimulated in simulation; the sixth was a performance-related issue on the pin interface that was found by thinking about the design; the seventh was a very specific set of events that resulted in a system hang; and the last was related to not responding appropriately to an error condition.

These bugs escaped detection for the following reasons:

- An exerciser running on a simulator was slow to encounter the conditions that would evoke the bug. Many conditions needed to occur concurrently, but all of them occurred infrequently.
- An assertion checker did not work properly.
- Comparisons between the RTL model and the structural model missed the bug.

All bugs were fixed before any systems were shipped to customers.

Details of these bugs follow. Included is information about how the bug was detected, a hypothesis on why the bug eluded detection before first-pass chips were fabricated, and lessons learned from the detection and elimination of the bug.

1. One bug was found by an exerciser running on the second-pass RTL model. A cache line victim failed to write back on a B-cache index match because a bypass occurred at the same time. This bug existed only in 32-byte cache mode and B-cache speed configurations of 4, 5, and 6. This bug could have been found in the first-pass model if this case had been generated pseudorandomly. Running many cases is crucial with a pseudorandom testing strategy. Given unlimited time and computation cycles, this bug might have been found earlier.
2. A second bug was caused by the B-cache read/write timing being off by one cycle. This bug could have caused multiple drivers to drive the data bus at one time. An assertion checker for this bug was in the RTL model, but the checker itself was not working properly. In the future, assertion checkers should be verified by causing the failure to occur and watching to see that it detects the case. In some cases, assertion checkers are written to flag an error for events that should never happen. Forcing an illegal situation to occur can be very difficult.
3. Another bug was found by an exerciser when a WRITE_BLOCK command was preceded by a single-cycle idle_BC signal assertion. This issue was

directly related to a specific B-cache speed and was related to another system configuration restriction. This issue caused a restriction to be added, but the design was not changed.

4. If the B-cache sequencer is performing a bypass immediately after a command loads in the B-cache address file and a reference is coming down the S-cache pipe, the B-cache index could change in back-to-back cycles. The index should change only every other cycle. An assertion checker should have been written to test for this situation and make sure it never occurred.
5. The performance-monitoring logic that counted load merges was not counting these events correctly. This bug was not in the RTL model but only in the actual implementation. Possibly, more RTL-to-CHANGO comparisons needed to be run on this section of logic.
6. Because of an LDxL/STxC bug, an invalidate to a locked address was not detected as a hit against the LDxL address. As a result, an STxC passed when it should have failed. This bug could have been detected if a focused test had been written with very specific timing of a FILL and an LDxL hitting the S-cache in consecutive cycles. Gaining control of this interaction on the system bus was not possible, however, and random simulations were relied upon to achieve this case. This was a rare event in the random simulations, but parameters could have been adjusted to make this occur more often.
7. For one specific system configuration, a READ or FLUSH command sent by the system to the Alpha 21164 chip could cause the system to hang. For this to happen, three specific events, all with very tight timing windows, needed to occur. We could have found this bug during simulation if we had emphasized this type of condition during the pseudorandom testing.
8. When responding to a command, the system had the option of asserting an error signal instead of its normal response. The error signal acted as an interrupt request to the Alpha 21164 chip. Under certain conditions, and for a narrow window of time, this error signal was not properly recognized. Testing of error conditions was a project goal but not a high priority compared to testing normal events. This bug could have been found earlier if additional error-mode tests had been run.

The above issues were fairly minor and all have been fixed in the version of the design that will be released to customers. The use of pseudorandom testing was very successful. Many major, complicated bugs were found over the course of the project that would never have been found using a focused effort. Because of the number of system configurations possible, a verification effort that consisted only of focused testing would have been impossible.

Acknowledgments

The Alpha 21164 functional verification effort was performed by a team of engineers from the SEG microprocessor verification group. Members of this team included Homayoon Akhiani, David Asher, Darren Brown, Rick Calcagni, Erik DeBriae, Jim Ellis, Bill Feaster, Mariano Fernandez, Jim Huggins, Mike Kantrowitz, Ginger Lin, Chris Mikulis, Lisa Noack, Ray Ratchup, Carol Stolicny, Scott Taylor, and Jonathan White. The CCLI user interface would not have been possible without John Pierce. Walker Anderson provided quality guidance through all phases of the project. The Alpha Architecture Group RAX team (Matt Baddeley, Larry Camilli, Ed Freedman, Joe Rantala, Pravin Santiago, Lucy Tancredi, Steve Torchia), once again, provided and supported an effective verification tool. Lastly, the success of the project and the final quality of the Alpha 21164 chip logical design are as much a tribute to the work of the architecture and design teams as they are to the work of the verification team.

References

1. J. Edmondson et al., "Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor," *Digital Technical Journal,* vol. 7, no. 1 (1995, this issue): 119–135.

2. W. Bowhill et al., "Implementation of a 300-MHz 64-bit Second-generation CMOS Alpha CPU," *Digital Technical Journal,* vol. 7, no. 1 (1995, this issue): 100–118.

3. W. Anderson, "Logical Verification of the NVAX CPU Chip Design," *Digital Technical Journal,* vol. 4, no. 3 (Summer 1992): 38–46.

4. A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-random Test Program Generator," *IBM Systems Journal,* vol. 30, no. 4 (1991): 527–538.

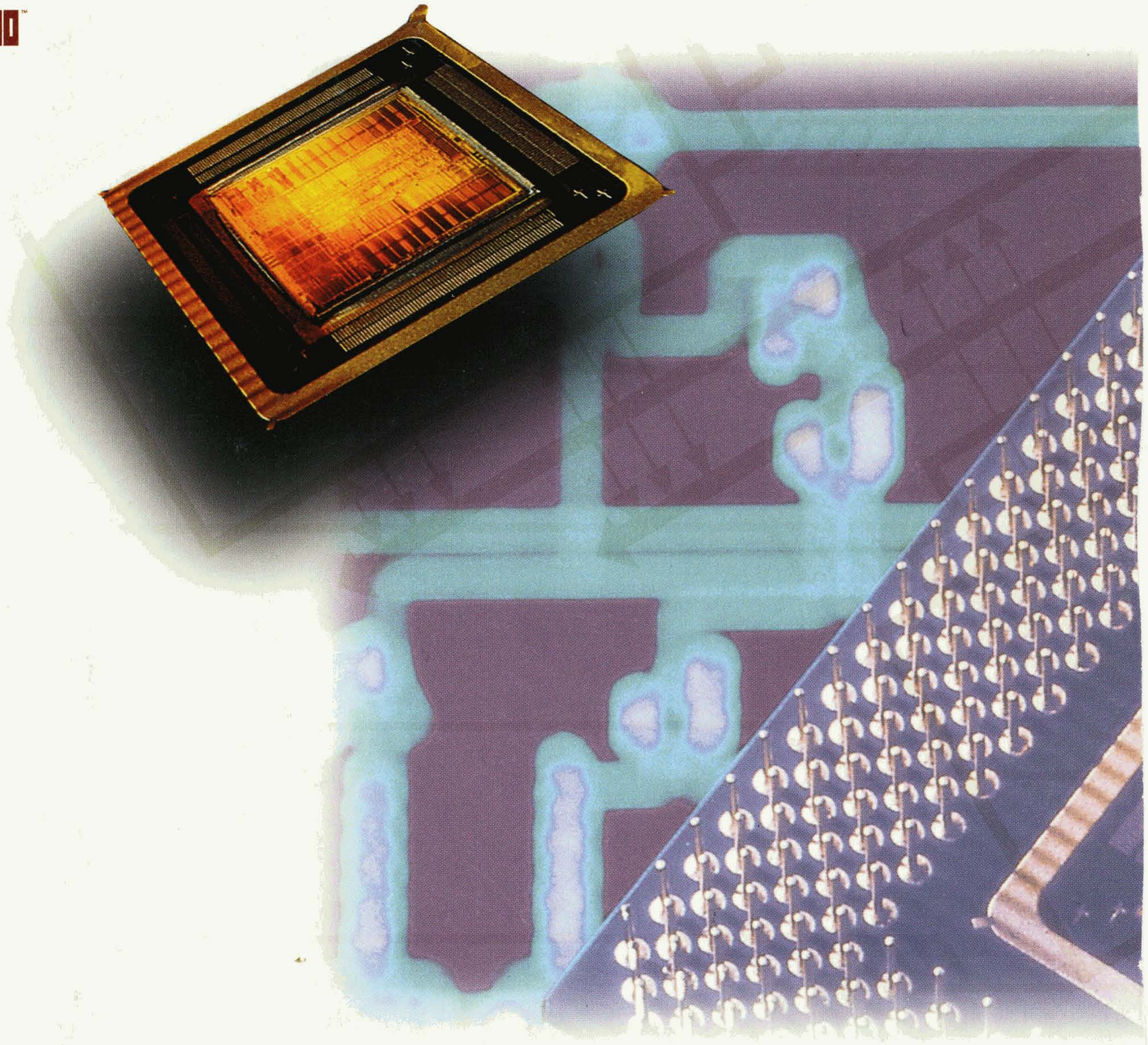
5. A. Ahi, G. Burroughs, A. Gore, S. LaMar, C-Y. Lin, and A. Wiemann, "Design Verification of the HP 9000 Series 700 PA-RISC Workstations," *Hewlett-Packard Journal* (August 1992): 34–42.

6. D. Wood, G. Gibson, and R. Katz, "Verifying a Multiprocessor Cache Controller Using Random Test Generation," *IEEE Design and Test of Computers* (August 1990): 13–25.

Biographies

Michael Kantrowitz
A principal engineer, Mike Kantrowitz is currently leading the verification effort for a new Alpha microprocessor and developing new verification tools and methods. Prior to this project, Mike was co-leader of the 21164 chip verification, responsible for the instruction fetch and execute units. He has also contributed to the verification of the Mariah, NVAX+, 21064 floating-point unit, and FAVOR vector unit. Before joining Digital in 1988, Mike worked at Raytheon Company. He has a B.S.E.E. from Stevens Institute of Technology and an M.S.E.E. from Worcester Polytechnic Institute. Mike is a member of IEEE.

Lisa M. Noack
A principal engineer, Lisa Noack is currently co-leading the chip verification effort for a new Alpha microprocessor. Prior to this work, Lisa was a co-leader of the 21164 chip verification and was responsible for memory, cache, and system interface units. Lisa has also contributed to the verification of the NVAX+ and NEXMI chips and the PVN module and chip set. Before she joined Digital in 1989, Lisa was employed at Data General Corporation as a design engineer responsible for the system design of I/O subsystems and various gate array design projects. She earned her B.S. and M.S. degrees in computer engineering from Syracuse University.



ISSN 0898-901X