Migrating to an OpenVMS AXP System: Planning for Migration

Order Number: AA–PV62A–TE

March 1994

This manual provides an overview of the process of migrating a VAX application to an AXP system and information to help you plan your migration effort.

Revision/Update Information: This manual supersedes *Migrating to an OpenVMS AXP System: Planning for Migration*, Version 1.5.

Software Version: OpenVMS AXP Version 6.1

Digital Equipment Corporation
Maynard, Massachusetts

March 1994

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK5779

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Send Us Your Comments

We welcome your comments on this or any other OpenVMS manual. If you have suggestions for improving a particular section or find any errors, please indicate the title, order number, chapter, section, and page number (if available). We also welcome more general comments. Your input is valuable in improving future releases of our documentation.

You can send comments to us in the following ways:

- Internet electronic mail: OPENVMSDOC@ZKO.MTS.DEC.COM
- Fax: 603-881-0120 Attn: OpenVMS Documentation, ZKO3-4/U08
- A completed Reader's Comments form (postage paid, if mailed in the United States), or a letter, via the postal service. Two Reader's Comments forms are located at the back of each printed OpenVMS manual. Please send letters and forms to:

  Digital Equipment Corporation
  Information Design and Consulting
  OpenVMS Documentation
  110 Spit Brook Road, ZKO3-4/U08
  Nashua, NH 03062-2698
  USA

You may also use an online questionnaire to give us feedback. Print or edit the online file SYS$HELP:OPENVMSDOC_SURVEY.TXT. Send the completed online file by electronic mail to our Internet address, or send the completed hardcopy survey by fax or through the postal service.

Thank you.

Contents

Preface

Migrating to an OpenVMS AXP System: Planning for Migration is one of a series of manuals designed to help you migrate existing application programs from a VAX system to an AXP system. This manual set is intended to help you make reasonable estimates about the type and amount of work involved in the various available migration methods, to enable you to select the method best suited to a given application, and to help you execute the migration.

Intended Audience

This manual is intended for anyone who evaluates an application for migration from OpenVMS VAX to OpenVMS AXP and produces a plan for the migration.

How to Use This Book

This book provides an overview of the process of migrating an application from OpenVMS VAX to OpenVMS AXP and information to help you plan a migration effort. It discusses the decisions you must make in planning your migration and how to get the information you need to make those decisions. For information about the technical details of the migration process, see the books listed in the section in this Preface titled Other Sources of Information About Migration.

Chapter 1 provides a summary of the relationship of OpenVMS and the VAX and Alpha AXP architectures:

- Areas in which OpenVMS AXP is highly compatible with OpenVMS VAX
- The characteristics of an AXP system, comparing the features of the Alpha AXP architecture to those of other RISC architectures and to those of the VAX architecture

Chapter 2 provides an overview of the process of migrating to an AXP system. It includes information on the following:

- Overview of the stages in the migration process
- The two main migration paths—recompiling source code and translating VAX images
- Migration support available from Digital

The remaining chapters describe the steps in the migration process and outline strategies for organizing a migration effort for VAX applications. The chapters describe how to produce an effective migration plan.

Chapter 3 describes how to analyze your application as a whole to determine exactly what you are migrating.

Chapter 4 considers the differences between the two main migration paths and the issues involved in choosing which path to take in migrating your application. It also describes how to analyze the individual parts of your application to identify architectural differences that affect migration and how to assess what is involved in resolving those differences.

Chapter 5 describes how to write a VAX to AXP migration plan.

Chapter 6 describes the steps in the actual migration, from setting up your migration environment to integrating the migrated application into a new environment.

Appendix A contains a checklist that you can use to evaluate your application for migration from OpenVMS VAX to OpenVMS AXP.

Appendix B contains a sample migration plan.

The Glossary contains definitions for terms introduced in this manual.

Other Sources of Information About Migration

Other OpenVMS AXP migration manuals include the following:

- *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*

  This manual describes how to build an AXP version of your VAX application by recompiling and relinking it. It discusses dependencies your application may have on features of the VAX architecture (such as assumptions about page size, synchronization, and condition handling) that you may need to modify to create a native AXP version. In addition, the manual describes how you can create applications in which native AXP components interact with translated VAX components.

- *DECmigrate for OpenVMS AXP Systems Translating Images*

  This manual describes the VAX Environment Software Translator (VEST) utility. This manual is distributed with the optional layered product, DECmigrate for OpenVMS Alpha AXP, which supports the migration of VAX applications to OpenVMS AXP. The manual describes how to use VEST to convert most user-mode VAX images to translated images that can run on AXP systems; how to improve the run-time performance of translated images; how to use VEST to trace AXP incompatibilities in a VAX image back to the original source files; and how to use VEST to support compatibility among native and translated run-time libraries. The manual also includes complete VEST command reference information.

- *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*

  This manual describes how to port VAX MACRO code to an AXP system using the VAX MACRO-32 Compiler for OpenVMS AXP. It describes the features of the compiler, presents a methodology for porting VAX MACRO code, identifies nonportable coding practices, and recommends alternatives to such practices. The manual also provides a reference section with detailed descriptions of the compiler's qualifiers, directives, and built-ins, and the system macros created for porting to OpenVMS AXP.

Other books that may help you in the migration process include the following:

- *OpenVMS Linker Utility Manual*

This manual provides detailed information about using the OpenVMS Linker utility to create AXP images.

- *A Comparison of System Management on OpenVMS AXP and OpenVMS VAX*

  This book contains information about system management aspects of migrating an entire system.

- *OpenVMS Compatibility Between VAX and AXP*

  This manual provides detailed information about the similarities and differences between OpenVMS VAX and OpenVMS AXP.

- *Alpha Architecture Reference Manual*

  This manual is the definition of the Alpha AXP architecture. It provides a complete description of the Alpha AXP central-processor hardware as seen by machine-language programs.

- *VMS for Alpha Platforms: Internals and Data Structures*

  This book describes the kernel of the OpenVMS AXP operating system.

- *OpenVMS Calling Standard*

  This book specifies the conventions that procedures use when calling one another and defines the argument-passing mechanisms and structures that support those conventions.

Conventions

In this manual, every use of OpenVMS AXP means the OpenVMS AXP operating system, every use of OpenVMS VAX means the OpenVMS VAX operating system, and every use of OpenVMS means both the OpenVMS AXP operating system and the OpenVMS VAX operating system.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
PF1 *x*	A sequence such as PF1 *x* indicates that you must first press and release the key labeled PF1 and then press and release another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	Horizontal ellipsis points in examples indicate one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.

( )	In format descriptions, parentheses indicate that if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In format descriptions, brackets indicate optional elements. You can choose one, none, or all of the options. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification, or in the syntax of a substring specification in an assignment statement.)
{ }	In format descriptions, braces surround a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason (user action that triggers a callback). Boldface text is also used to show user input in Bookreader versions of the manual.
italic text	Italic text emphasizes important information and indicates variables and complete titles of manuals. Variables include information that varies in system messages (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
-	A hyphen in code examples indicates that additional arguments to the request are provided on the line that follows.
numbers	All numbers in text are assumed to be decimal, unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1
VAX, Alpha AXP, and OpenVMS

For many applications, migrating from OpenVMS VAX to OpenVMS AXP is straightforward. If your application runs only in user mode and is written in a standard high-level language, you most likely can recompile it with a native AXP compiler and relink it to produce a version that runs successfully on an AXP system. This book is intended to help you evaluate your application and to handle the relatively few cases that are more complicated.

1.1 Compatibility of VAX and AXP Systems

The OpenVMS AXP operating system is designed to preserve as much compatibility with the OpenVMS VAX user, system management, and programming environments as possible. For general users and system managers, OpenVMS AXP has the same interfaces as OpenVMS VAX. For programmers, the goal is to come as close as possible to a "recompile, relink, and run" model for migration.

Many aspects of an application running on an OpenVMS VAX system remain unchanged on an AXP system:

User Interface

- **DIGITAL Command Language (DCL)**

  The DIGITAL Command Language (DCL), the standard user interface to OpenVMS, remains essentially unchanged with OpenVMS AXP. All commands, qualifiers, and lexical functions available on OpenVMS VAX also work on OpenVMS AXP.

- **Command Procedures**

  Command procedures written for earlier versions of OpenVMS VAX continue to work on an AXP system without change. However, certain command procedures, such as build procedures, must be changed to accommodate new compiler qualifiers and linker switches. Linker options files will also require modification, especially for shareable images.

- **DECwindows**

  The window interface, DECwindows Motif, is unchanged.

- **DECforms**

  The DECforms interface is unchanged.

- **Editors**

  The two standard OpenVMS editors, EVE and EDT, are unchanged.

System Management Interface

The system management utilities are mostly unchanged. One major exception is that device configuration functions, which appear in the System Generation utility (SYSGEN) on VAX systems, are provided in the System Management utility (SYSMAN) for OpenVMS AXP. For more information, see *A Comparison of System Management on OpenVMS AXP and OpenVMS VAX*.

Programming Interface

In general, the system service and run-time library (RTL) calling interfaces remain unchanged. You do not need to change the definitions of arguments. The few differences fall into two categories:

- Some system services and RTL routines (such as the memory management system and exception-handling services) operate somewhat differently on VAX and AXP systems. See the *OpenVMS System Services Reference Manual* and *OpenVMS RTL Library (LIB$) Manual* for further information.
- A few RTL routines are so closely tied to the VAX architecture that their presence on an AXP system would not be meaningful:

Routine Name	Restriction
LIB$DECODE_FAULT	Decodes VAX instructions.
LIB$DEC_OVER	Applies to VAX Processor Status Longword (PSL) only.
LIB$ESTABLISH	Similar functionality supported by compilers on AXP systems.
LIB$FIXUP_FLT	Applies to VAX PSL only.
LIB$FLT_UNDER	Applies to VAX PSL only.
LIB$INT_OVER	Applies to VAX PSL only.
LIB$REVERT	Supported by compilers on AXP systems.
LIB$SIM_TRAP	Applies to VAX code.
LIB$TPARSE	Requires action routine interface changes. Replaced by LIB$TABLE_PARSE.

Most VAX images that call these services and routines will work when translated and run under the Translated Image Environment (TIE) on OpenVMS AXP. For more information on TIE, see Section 6.2.2.1 and *DECmigrate for OpenVMS AXP Systems Translating Images*.

Data

The on-disk format for ODS-2 data files is the same on VAX and AXP systems. However, ODS-1 files are not supported on OpenVMS AXP.

Record Management Services (RMS) and file management interfaces are unchanged.

The IEEE little-endian data types S_floating and T_floating have been added.

Most VAX data types are retained in the Alpha AXP architecture; however, support for H_floating and full-precision D_floating has been eliminated from hardware to improve overall system performance.

AXP hardware converts D_floating data to G_floating for processing. On VAX systems, D_floating has 56 fraction bits (D56) and 16 decimal digits of precision. On AXP systems, D_floating has 53 fraction bits (D53) and 15 decimal digits of precision.

The H_floating and D_floating data types can usually be replaced by G_floating or one of the IEEE formats. However, if you require H_floating or the extra precision of D56 (56-bit D_floating), you may have to translate part of your application.

Databases

Standard Digital databases (such as Rdb/VMS) function the same on VAX and AXP systems.

Network Interfaces

VAX and AXP systems both support the following interfaces:

- Interconnects
  - Ethernet
  - X.25
  - FDDI
- Protocols
  - DECnet (Phase IV in Version 6.1; Phase V later)
  - TCP/IP
  - OSI
  - LAD/LAST
  - LAT (Local Area Transport)
- Peripheral connections
  - TURBOchannel
  - SCSI
  - Ethernet
  - CI
  - DSSI
  - XMI
  - Futurebus+
  - VME

1.2 Differences Between the VAX and Alpha AXP Architectures

The VAX architecture is a robust, flexible, complex instruction set computer (CISC) architecture used across the entire family of VAX systems. The use of a single, integrated VAX architecture with the OpenVMS operating system permits an application to be developed on a VAXstation, prototyped on a small VAX system, and put into production on a large VAX processor or run on a fault-tolerant VAXft processor. The advantage of the VAX system approach is that it enables individual solutions to be tailored and fitted easily into a larger, enterprisewide solution. The hardware design of VAX processors is particularly

suitable for high-availability applications, such as dependable applications for mission-critical business operations and server applications for a wide variety of distributed client/server environments.

The Alpha AXP architecture implemented by Digital is a high-performance reduced instruction set computing (RISC) architecture that can provide 64-bit processing on a single chip. It processes 64-bit virtual and physical addresses and 64-bit integers and floating-point numbers. The 64-bit capability is especially useful for applications that require high-performance and very large addressing capacity. For example, AXP processors are especially appropriate for graphics or numeric-intensive software applications such as econometric or weather forecasting that involve imaging, multimedia, visualization, simulation, and modeling.

The Alpha AXP architecture is designed to be scalable and open. It can be implemented on a single chip in a palmtop system or with thousands of chips in a massively parallel supercomputer. The architecture also supports multiple operating systems, including OpenVMS AXP.

Table 1–1 summarizes some major differences between the Alpha AXP and VAX architectures.

Table 1–1 Comparison of Alpha AXP and VAX Architectures

Alpha AXP	VAX
• 64-bit addresses	• 32-bit addresses
• 64-bit processing	• 32-bit processing
• Multiple operating systems: OpenVMS, OSF/1, Windows NT	• One operating system: OpenVMS
• Instructions	• Instructions
– Simple	– Some complex
– All same length (32 bits)	– Variable length
• Load/store memory access	• Permits combining operations and memory access in a single instruction
• Severe penalty for unaligned data	• Moderate penalty for unaligned data
• Many registers	• Relatively few registers
• Out-of-order instruction completion	• Instructions completed in order issued
• Deep pipelines and branch prediction	• Limited use of pipelines
• Large page size (which varies from 8 KB to 64 KB, depending on hardware)	• Smaller page size (512 bytes)

General RISC Characteristics

Some features of the Alpha AXP architecture are typical of newer RISC architectures in general. The following features are especially important:

- A simplified instruction set

The Alpha AXP architecture uses relatively simple instructions, all of which are 32 bits long. Common instructions require only one clock cycle. Uniformly sized simple instructions allow a RISC implementation to achieve high performance goals by adopting techniques such as **multiple instruction issue** and optimized instruction scheduling.

- Multi-instruction issue

  Most AXP platforms issue two instructions per clock cycle. Future machines may issue four instructions per clock cycle.

- A load/store operation model

  The Alpha AXP architecture defines 32 64-bit integer registers and 32 64-bit floating-point registers. Most data manipulation occurs between registers. Before an operation, operands are loaded from memory into registers; after the operation, the results are stored in memory from a result register.

  Restricting operations to register operands allows the use of a simple, uniform instruction set. Moreover, the separation of memory access from arithmetic operations results in a large performance gain in a system that can fully exploit pipelining, instruction scheduling, and parallel operational units.

- Elimination of microcode

  Because the Alpha AXP architecture does not use microcode, AXP processors are saved the time required to fetch microcode instructions from random-access memory (RAM) in order to execute a machine instruction.

- A processor that allows parallel instruction execution and out-of-order completion of instructions

  The Alpha AXP architecture does not require that instructions always complete in the order in which they are issued. As a result, an AXP processor can improve performance by delaying the reporting of an arithmetic or floating-point exception until the execution stream allows the reporting without a performance penalty.

Alpha AXP Specific Characteristics

Besides these generic RISC characteristics, the Alpha AXP architecture offers features that promote running migrated VAX applications on an AXP system. These features include:

- Hardware support for all VAX data types except H_floating and D_floating. (For information on what to do if your application uses H_floating or D_ floating data, see Section 4.2.3.1.2.)
- Certain privileged architecture features, such as four processor modes (user, supervisor, executive, and kernel), 32 interrupt priority levels (IPLs), and asynchronous system traps (ASTs).
- A privileged architecture library (PAL), part of an environment known as PALcode, that supports the atomic execution of certain VAX instructions, such as Change Mode (CHM*x*), Probe (PROBE*x*), queue instructions, and REI.

2

Overview of the Migration Process

The process for converting your VAX programs to run on an AXP system includes the following stages:

1. Evaluate the code to be migrated:
   - Take inventory of the elements of your application and its environment. Identify any dependencies on other programs.
   - Review code in each element to find potential obstacles to migration.
   - Decide on the best method for moving each part of the application to the AXP system.
2. Write a migration plan.
3. Set up the migration environment.
4. Migrate your application.
5. Debug and test the migrated application.
6. Integrate the migrated software into a software system.

There are a number of tools and Digital services available to help you migrate your applications to OpenVMS AXP. These tools are described in the context of the process described in this manual. The migration services are summarized in Section 2.2.

2.1 Migration Paths

There are two ways to convert a program to run on an AXP system:

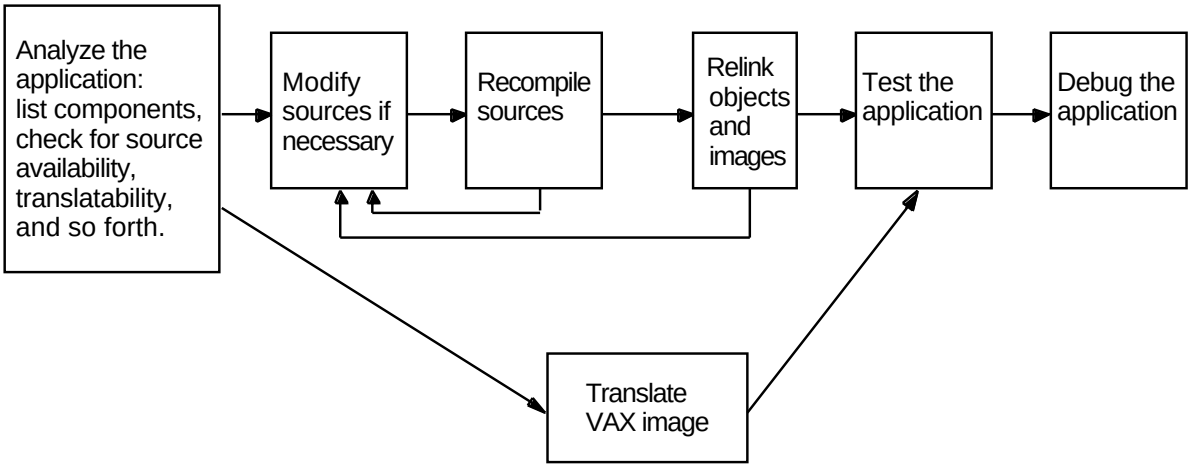
- Recompiling and relinking, which creates native AXP images
- Translating, which creates native AXP images with some routines emulated under TIE

These two methods are illustrated in Figure 2–1. Section 4.4 discusses factors to consider when choosing a migration method.

Recompiling

The most effective way to convert a program from OpenVMS VAX to OpenVMS AXP is to recompile the source code using a native AXP compiler (such as DEC C or DEC Fortran) and then to relink the resulting object files and any required shareable images using the appropriate switches and options files with the OpenVMS Linker. This method produces a native AXP image that takes full advantage of the speed of the AXP system.

Figure 2–1 Methods for Moving VAX Applications to an AXP System



Translating

Despite differences between VAX and AXP systems, you can run most user-mode VAX images without error on an AXP system by using the VAX Environment Software Translator (VEST), which is part of the DECmigrate for OpenVMS Alpha AXP layered product. For a list of exceptions, see Section 4.4. This process provides a higher degree of VAX compatibility than recompiling the sources, but since the translated image does not provide the same high performance as a recompiled image, translation is used primarily as a safety net when recompiling is impossible or impractical. For example, translation is used in the following situations:

- When an appropriate compiler is not yet available for OpenVMS AXP
- When source files are not available

VEST translates the VAX binary image file into a native AXP image that runs under the Translated Image Environment (TIE) on an AXP system. (TIE is a shareable image that is bundled with OpenVMS AXP.) Translation does not involve running a VAX image under emulation or interpretation (with certain limited exceptions). Instead, the new AXP image contains Alpha AXP instructions that perform operations identical to those performed by the instructions in the original VAX image.

A translated image should run as fast on an AXP system as the original image runs on a VAX system. However, since the translated image does not benefit from the optimizing compilers that take full advantage of the Alpha AXP architecture, it will typically run only about 25 to 40 percent as fast as a native AXP image. Major causes of this reduced performance are unaligned data and extensive use of complex VAX instructions.

For more information on image translation and VEST, see Section 6.2.2.1 and *DECmigrate for OpenVMS AXP Systems Translating Images*.

Mixing Native AXP and Translated Images

You can mix migration methods among the individual images that comprise an application. An application can also be partially translated as one stage in a migration: this allows the application to run and to be tested on AXP hardware before being completely recompiled. For more information about interoperability of native AXP and translated VAX images within an application, see Section 4.4.2.

2.2 Support from Digital in Migrating Your Application

Digital offers a variety of services to help you migrate your applications to OpenVMS AXP.

Digital customizes the level of service to meet your needs. The migration services available include the following:

- Orientation Service (Order number: QS-ALPAA-CA)
- Detailed Analysis Service
- Migration Support Service
- Project Planning Service
- Custom Project Service

The Orientation Service is available from DECdirect (1-800-DIGITAL). (The order number listed for this service is in effect in the United States.) The remaining services are tailored to the needs of your organization.

To determine which services are appropriate for you, contact your Digital account representative or authorized reseller, or call 1-800-832-6277 (within the United States) or 1-603-884-8990 (outside the United States.)

2.2.1 Orientation Service

The Orientation Service helps you understand the issues involved in migrating an application from OpenVMS VAX to OpenVMS AXP systems. This package is recommended for all Alpha AXP migration customers.

2.2.2 Detailed Analysis Service

The Detailed Analysis Service provides a customized methodology for planning the migration of applications from OpenVMS VAX to OpenVMS AXP systems. A Digital consultant conducts a detailed investigation of your application, highlights items that may affect the migration effort and schedule, and produces a detailed migration assessment plan. This service is recommended for all customers who would like help with their migration.

2.2.3 Migration Support Service

The Migration Support Service is provided by a Digital migration expert who assists in the migration project. This service includes on-site technical consulting at the level you require.

2.2.4 Project Planning Service

The Project Planning Service is provided by a Digital consultant. The consultant provides the following:

- Information that you need to understand the size and scope of a migration project

- Information that you need to develop a detailed migration strategy based on your requirements
- Detailed migration project plan

2.2.5 Custom Project Service

The Custom Project Service results in a total turnkey migration. It is recommended for customers who want Digital to port their entire application.

2.2.6 Business Partner Development Assistance Centers

Business Partner Development Assistance (BPDA) centers provide migration services worldwide, as shown in Table 2–1. Migration experts staff the centers and assist Digital's business partners with their porting efforts.

For detailed information on migration services, contact your Digital account representative or authorized reseller, or call 1-800-832-6277 or 1-603-884-8990.

Table 2–1 Locations of BPDA Centers

Europe	United States	Asia
Basingstoke	Detroit	Hong Kong
Munich	Marlboro	Tokyo
Paris	Palo Alto	

2.3 Migration Training

Digital Customer Training offers several seminars and courses to provide migration training to third-party application developers and end users. The first course in the following list is designed for technical or MIS managers, and the others are designed for experienced OpenVMS VAX programmers:

- *Alpha AXP Planning Seminar*– 2 days, EY-L570E-SO
- *Migrating HLL Applications to OpenVMS Alpha AXP*– 3 days, EY-L577E-LO
- *Migrating MACRO–32 Applications to OpenVMS AXP*—2 days, EY-L578E-LO

To obtain a schedule and enrollment information in the United States, call 1-800-332-5656. In other locations, contact your Digital account representative or authorized reseller.

3

Evaluating Your Application: Taking Inventory

Evaluating your application identifies the work to be done and allows you to plan the rest of the migration. The evaluation will allow you to produce a migration plan and to answer the following questions:

- How to migrate your application
- How much effort, time, and cost the migration will require
- Whether to use Digital's Multivendor Customer Services

The evaluation process has three main stages:

1. General inventory, including identifying dependencies on other software
2. Source analysis to identify coding practices that affect migration
3. Selection of a migration method: rebuilding from source code or translating

When you have completed these steps, you will have the information necessary to write an effective migration plan.

The first step in evaluating an application for migration is to determine exactly what has to be migrated. This includes not only the application itself, but everything that the application requires in order to run properly. To begin evaluating your application, identify and locate the following items:

- Parts of the application
  - Source modules for the main program
  - Shareable images
  - Object modules
  - Libraries (object module, shareable image, text, or macro)
  - Data files and databases
  - Message files
- Other software on which your application depends, for example:
  - Run-time libraries
  - Digital layered products
  - Third-party products

  To help identify dependencies on other code, use VEST with the qualifier /DEPENDENCY. VEST/DEPENDENCY identifies executable and shareable images on which your application depends, such as run-time libraries, system services, and other applications. For details on using VEST/DEPENDENCY, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

- Required operating environment
  - System characteristics

    What sort of system is required to run and maintain your application; for example, how much memory is required, how much disk space, and so on?
  - Build procedures

    This includes Digital tools such as Code Management System (CMS) and Module Management System (MMS).
  - Testing suite

    You will need your tests to confirm that the migrated application runs correctly and to evaluate its performance.

Many of these items have already been migrated to OpenVMS AXP, for example:

- Digital software bundled with OpenVMS
  - RTLs
  - Other shareable libraries, such as those supplying callable utility routines and application library routines
- Digital layered products
  - Compilers and compiler RTLs
  - Database managers
  - Networking environment
- Third-party products

  Many third-party applications now run on OpenVMS AXP. To determine whether a particular application has been migrated, contact the application vendor.

You will be responsible for migrating your application and your development environment, including build procedures and testing suites.

When you have completed your inventory of your application, continue with a more detailed evaluation of each module and image, as described in Chapter 4.

4
Selecting a Migration Method

When you have completed the inventory of your application, you must then decide how to migrate each part of it: by recompiling and relinking or by translating. The large majority of applications can be migrated just by recompiling and relinking them. If your application runs only in user mode and is written in a standard high-level language, it is probably in this category. For the major exceptions, see Section 4.2.

The remainder of this chapter discusses how to choose a migration method for the relatively few applications that require more work to migrate. To make this decision, you will need to know which methods are possible for each part of the application, and how much work will be required for each method.

Note

The following process assumes that you will recompile your application if possible, and use translation only for parts that cannot be recompiled or as a temporary measure in the course of your migration.

The following sections outline a process for choosing a migration method. This process includes the following steps:

1. Determine which of the two migration methods is possible.

   Under most conditions, you can either recompile and relink your program or translate the VAX image. Section 4.1 describes cases where only one migration method is available.

2. Identify architectural dependencies that affect recompilation.

   Even if your application is generally suitable to be recompiled, it may contain code that depends on features of the VAX architecture that are incompatible with the Alpha AXP architecture.

   Section 4.2 discusses these dependencies and provides information that allows you to identify them and to begin to estimate the type and amount of work required to accommodate any dependencies you find.

   Section 4.3 describes tools and methods you can use to help answer the questions that come up in evaluating your application.

3. Decide whether to recompile or translate.

   After you have evaluated your application, you must decide which migration method to use. Section 4.4 describes how to make the decision by balancing the advantages and disadvantages of each method.

If you cannot recompile and relink your program, or if the VAX image uses features specific to the VAX architecture, you may wish to translate that image. Section 4.4.1 describes ways to increase the compatibility and performance of translated images.

As shown in Figure 4–1, the evaluation process consists of a series of questions and some tasks you can perform to help answer those questions. Digital provides a number of tools which you can use to help answer the questions; these tools are described at the relevant points in the process.

4.1 Which Migration Methods are Possible?

In most cases, you can either recompile and relink, or translate, your application. However, depending on the design of your application, only one of the two migration paths may be available to you:

- Programs that cannot be recompiled

  The following types of images must be translated:

  – Software that is written in a programming language for which no AXP compiler is yet available

    Native compilers for Ada, BASIC, C, C++, COBOL, FORTRAN, Pascal, PL/I, and VAX MACRO are available with OpenVMS AXP Version 6.1; other languages, including LISP, are planned for later releases.

  – Executable and shareable images for which the source code is not available

  – Programs that require H_floating or 56-bit D_floating data

- Images that cannot be translated

  The source code must be recompiled and relinked (and possibly revised) for the following types of images:

  – Images produced prior to OpenVMS VAX Version 4.0

  – Images that use PDP–11 compatibility mode

  – Based images

  – Images that contain coding practices that are currently unsupported by the Alpha AXP architecture. These include code that:

    – Runs in inner access modes or elevated IPL (for example, VAX device drivers)

    – Refers directly to addresses in system space

    – Refers directly to undocumented system services

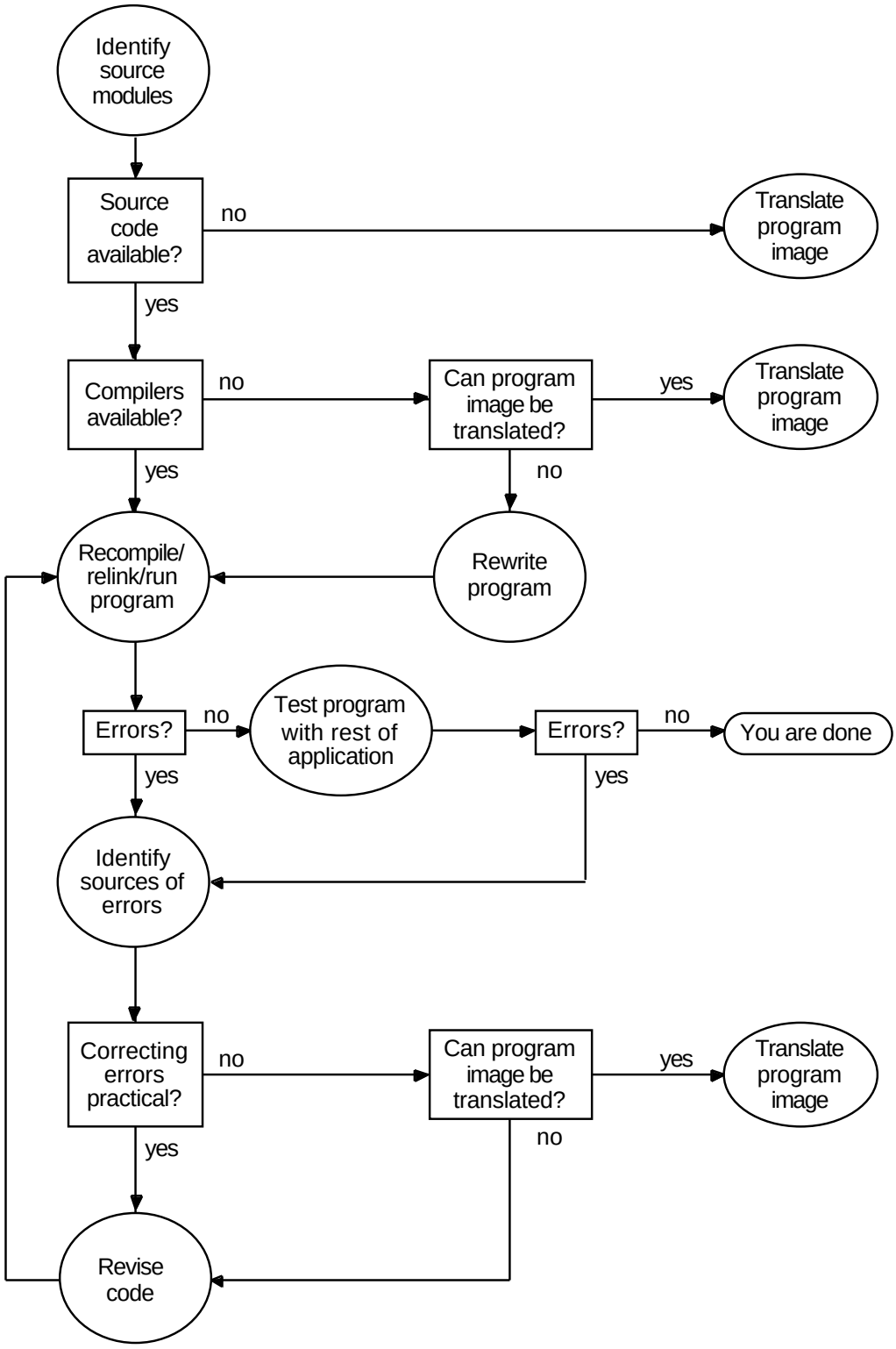
    – Uses threaded code; for example, code that switches stacks

    – Uses VAX vector instructions

    – Uses privileged VAX instructions

    – Inspects or modifies return addresses or makes other decisions based on a program counter (PC)

Figure 4–1 Migrating a Program



ZK–4990A–GE

- Depends on exact access-violation behavior due to 512-byte size memory page dependencies
- Aligns global sections on boundaries other than the native machine page boundary (for example, depends on a 512-byte memory page size)
- Uses most of the VAX P0 or P1 space or is otherwise sensitive to the space taken up by the translated-image run-time support routines

Although the translated image's run-time performance will be degraded because of the amount of VAX code that TIE will be required to interpret, VEST can probably translate the following kinds of images:

- Images that include self-modifying or created-on-the-fly VAX code, except for the code generated at run time by TIE
- Images with code that inspects the instruction stream, except when TIE interprets such code at run time

For more information on which images can be translated, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

4.2 Coding Practices That Affect Recompilation

Many applications, especially those that use only standard coding practices or are written with portability in mind, will migrate from OpenVMS VAX to OpenVMS AXP with little or no trouble. However, recompiling an application that depends on VAX specific features that are incompatible with the Alpha AXP architecture will require modifying your source code. Typical incompatibilities include use of the following:

- VAX MACRO assembly language to obtain high performance on a VAX system or to make use of features specific to the VAX architecture
- Privileged code
- Features specific to the VAX architecture

If none of these incompatibilities is present in your application, the rest of this chapter does not apply to you.

4.2.1 VAX MACRO Assembly Language

On AXP systems, VAX MACRO is not the assembly language, but just another compiled language. However, unlike the high-level language AXP compilers, the VAX MACRO–32 Compiler for OpenVMS AXP does not produce highly optimized code in all cases. Digital strongly recommends that you use the VAX MACRO–32 Compiler for OpenVMS AXP only as a migration aid, not for writing new code.

Many of the reasons for using assembly language on a VAX system are no longer relevant on AXP systems, for example:

- There is no inherent performance advantage in using assembly language on a RISC processor. RISC compilers, such as those in the AXP compiler set, can generate optimized code that takes advantage of architecture- and implementation-specific features more easily and efficiently than a programmer can.
- New system services can perform some functions that previously required assembly language.

For more information on migrating MACRO code, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*.

4.2.2 Privileged Code

VAX code that executes in inner access mode (kernel, executive, or supervisor mode) or that references system space is more likely to use coding practices dependent on the VAX architecture or to refer to VAX data cells that do not exist on OpenVMS AXP. Such code will not migrate to an AXP system without change. These programs will require recoding, recompiling, and relinking.

Code in this category includes:

- User-written system services and other privileged shareable images

  For more information, see the *OpenVMS Programming Concepts Manual* and the *OpenVMS Linker Utility Manual*.

- Device drivers and performance monitors not supplied by Digital
- Code that uses special privileges; for example, code that uses $CMEXEC or $CMKRNL system services, or code that uses the $CRMPSC system service with the PFNMAP option

  For more information, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

- Code that uses internal OpenVMS routines or data, such as:
  - Code that links against the system symbol table, SYS.STB, to access locations in system address space
  - Code that compiles against SYS$LIBRARY:LIB

For assistance in migrating inner-mode code that refers to the OpenVMS executive, contact Multivendor Customer Services.

4.2.3 Features Specific to the VAX Architecture

In order to achieve its high performance, the Alpha AXP architecture differs significantly from the VAX architecture. Software developers who have become accustomed to writing code that relies on certain aspects of the VAX architecture must be aware of architectural dependencies in their code in order to transport it successfully to an AXP system.

Common architectural dependencies, along with ways to identify them and actions you can take to eliminate them, are described briefly in the following sections. For a detailed discussion of ways to identify and to eliminate these dependencies, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.1 Performance Issues

Two differences between the VAX and Alpha AXP architectures do not keep a VAX application from running on OpenVMS AXP, but do have a significant performance impact:

- Data alignment
- Choice of data types

4.2.3.1.1 Data Alignment

Data is **naturally aligned** when its address is an integral multiple of the size of the data in bytes. For example, a longword is naturally aligned at any address that is a multiple of 4, and a quadword is naturally aligned at any address that is a multiple of 8. A structure is naturally aligned when all its members are naturally aligned.

Accessing data that is not naturally aligned in memory incurs a significant performance penalty both on VAX and on AXP systems. On VAX systems, since most languages align data on the next available byte boundary by default, the VAX architecture provides hardware support that minimizes the performance penalty in referencing unaligned data. On AXP systems, however, the default is to align each data item naturally, so Alpha AXP, like other typical RISC architectures, does not provide hardware support to minimize the performance degradation from using unaligned data. As a result, references to naturally aligned data on AXP systems are 10 to 100 times faster than references to unaligned data.

AXP compilers automatically correct most potential alignment problems and flag others.

Finding the Problem

To find instances of unaligned data, you can use the following methods:

- Use a switch provided by most AXP compilers that allows the compiler to report compile-time references to unaligned data. For example, for DEC C and DEC Fortran programs, compile with the qualifier /WARNING=ALIGNMENT.
- To detect unaligned data at run time, use the OpenVMS Debugger or DEC PCA (Performance and Coverage Analyzer).

Eliminating the Problem

To eliminate unaligned data, you will be able to use one or more of the following methods:

- Maximize performance by aligning data items on quadword boundaries, since AXP systems generally provide only quadword **granularity** (see Section 4.2.3.2.2).
- Compile with natural alignment, or, when language semantics do not provide for this, move data to be naturally aligned. Where filler is inserted to ensure that data remains aligned, there is a penalty in increased memory size. A useful technique for ensuring naturally aligned data while conserving memory is to declare longer variables first.
- Use high-level-language instructions that force natural alignment within data structures. For example, in DEC C, natural alignment is the default option. To define data structures that must match the VAX C default alignment—such as on-disk data structures—use the construct #PRAGMA NO_MEMBER_ALIGNMENT. With DEC Fortran, local variables are naturally aligned by default. To control alignment of record structures and common blocks, use the /ALIGN qualifier.
- Use compiler switches that generate VAX compatible unaligned data-structure mappings. Use of these switches will result in AXP programs that are functionally correct but potentially slow.

Note

Software that is converted to natural alignment may be incompatible with other software that is running translated, on a VAX system in the same VMScluster environment, or over a network; for example:

- An existing file format may specify records with unaligned data.
- A translated image may pass unaligned data to, or expect it from, a native image.

In such cases, you will have to adapt all parts of the application to expect the same type of data, either aligned or unaligned.

For more information on data alignment, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.1.2 Data Types

In order to improve their performance, AXP processors implement the packed decimal, H_floating, and full-precision D_floating data types by using software, as follows:

- Packed decimal

  Eighteen-digit packed decimal data is converted to 64-bit binary integers internally, which provides very fast COBOL performance.

- H_floating

  AXP compilers do not support H_floating data; however, the Translated Image Environment (TIE) provides emulated support for H_floating data in translated images.

- D_floating

  D_floating data is implemented on AXP platforms in the following ways:

  – Using G_floating hardware (D53). AXP hardware converts D_floating data (D53) to G_floating for processing. This provides speed and data-type compatibility with existing binary files that contain D_floating data, but loses 3 fraction bits compared to D_floating arithmetic on current VAX systems. D_floating data is thus processed with 15 decimal digits of precision instead of the 16 decimal digits supplied by D56 on a VAX system.

  – Using software emulation (D56) for translated images. This gives exact D56 format VAX results, but is slower than D53 or G_floating.

Eliminating the Problem

To eliminate data type problems, you will be able to use one or more of the following methods:

- Instead of H_floating, use G_floating or IEEE T_floating whenever possible because both:

  – Support data in the range 10^{-308} to 10^{308}

  – Have approximately 15 decimal digits of precision

- Instead of decimal data types, use integer data types whenever possible.

For more information on Alpha AXP data types, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.2 Protection of Shared Data

Several differences between the VAX and Alpha AXP architectures can affect the integrity of shared data.

4.2.3.2.1 Modifying Data in Memory

An **atomic operation** is one in which:

- Intermediate or partial results cannot be seen by other processors or devices.
- The operation cannot be interrupted (that is, once started, the operation continues until completion).

On OpenVMS AXP, any operation that reads, modifies, and stores data in memory will be broken into several instructions, and can be interrupted between any of those instructions. As a result, if your application expects to modify data in shared memory atomically, you must take steps to guarantee the atomicity of the operation.

An application can depend on the atomicity of operations under any of the following conditions:

- An AST routine within the process shares data with the mainline code.
- The process shares data in a **writable global section** with another process that executes on the same CPU (that is, in a uniprocessor system).
- The process shares data in a writable global section with another process that may execute concurrently on another CPU (that is, in a multiprocessor system).

Finding the Problem

To find dependencies on atomicity, reexamine use of shared variables for hidden or explicit assumptions of atomicity.

Eliminating the Problem

To eliminate general problems of instruction atomicity, you will be able to use one or more of the following methods:

- Use language constructs, where available, that guarantee atomicity to protect shared variables: for example, in C, the VOLATILE declaration.
- Use explicit **synchronization** rather than relying on assumptions of atomicity.
- Use OpenVMS locking services (such as $ENQ and $DEQ), Parallel Processing Run-Time Library (PPL$) routines, or LIB$ routines.
- To synchronize with an AST thread, use the $SETAST system service in the mainline code to block the AST and then reenable delivery after the instruction has completed.

For more information on synchronization, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.2.2 Reading or Writing Data Smaller Than a Quadword

Granularity refers to the size of the data that can be read or written to memory as an atomic operation, without interfering with data in adjacent memory locations. Machines such as the VAX that provide granularity at the byte level are said to be **byte granular**. AXP systems are **quadword granular**.[1]

Since OpenVMS AXP is quadword granular, writes to a shared byte, word, or longword may corrupt other data present in the same quadword as the shared data. This occurs when:

- A program attempts to modify a byte, word, or longword.
- An unaligned field of any size crosses an aligned quadword boundary, which creates a byte, word, or longword that must be written independently.

Note

All of the types of data sharing listed in the discussion of atomicity (Section 4.2.3.2.1) can create granularity problems in the rest of the quadword containing the intentionally shared data.

In addition, if a process invokes asynchronous system services or asynchronously completing library functions that write a result back to process space, then the data written back can create granularity problems in the quadword that contains it; for example:

- An asynchronous system service that writes to a status block
- An I/O operation that writes to a process buffer
- An I/O operation in which a direct-memory-access (DMA) controller writes to a process buffer

Finding the Problem

To find uses of byte, word, or longword granularity, you can use the following methods:

- Look for intentionally shared data (between an AST and main thread or between processes). Check whether the shared data occupies the same quadword as other data that might be written.
- Look for data written back by asynchronous system services or library calls that complete asynchronously. Check whether that data occupies the same quadword as other data written by another process.
- Look for I/O buffers that contain data written back asynchronously from a device. Check whether the start and end of the buffers occupy the same quadword as data written by another process.

[1] The Alpha AXP architecture also supports longword granularity, but assuming longword granularity is not recommended. Digital compilers assume by default that source code does not depend on granularity finer than quadword.

Eliminating the Problem

To eliminate use of granularity at a level smaller than the quadword, you will be able to use one or more of the following methods:

- Put shared items in private quadwords.
- Align I/O buffer heads on quadword boundaries and move any data after the buffer into the next quadword.
- If the problem is not caused by data shared with the system, use a higher-level synchronization mechanism to interlock both intentionally shared data and background data in the same quadword.

AXP compilers will not provide byte, word, or longword granularity by default, but to maintain compatibility with your current code, they will allow you to specify byte, word, unaligned longword, and unaligned quadword granularity. For more information, see the documentation for the individual compilers.

For more information on read/write granularity, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.2.3 Page Size Considerations

Page size governs the amount of virtual memory allocated by memory management routines and system services. It is also the basis on which protection is assigned to code and data in memory.

The OpenVMS VAX operating system allocates memory in multiples of 512 bytes. To improve overall system performance, AXP systems have much larger page sizes, ranging from 8 KB to 64 KB, depending on the specific hardware platform.

Page size is a factor in the management of system resources, such as working set quotas. In addition, memory protection on VAX systems can vary for each 512-byte region of memory; on AXP systems, the granularity of memory protection is much larger, depending on the system's page-size implementation.

Note

The change to a larger page size affects only applications that explicitly rely on a 512-byte page size, for example, applications that:

- Use "512" to:
  - Compute memory usage.
  - Calculate page table requirements.
- Change protection on a 512-byte granularity.
- Use the system service Create and Map Section ($CRMPSC) to map a file into a specific location in the process address space (for example, to reuse memory when available memory is limited).

Finding the Problem

To find uses of the VAX page size, identify code that manipulates virtual memory in 512-byte chunks or relies on 512-byte memory protection granularity. Search your code for occurrences of numbers such as the decimal values 511, 512, or 513; the hexadecimal values 1FF, 200, or 201; and so forth.

Eliminating the Problem

To eliminate conflicts between the VAX and AXP page sizes, you will be able to use one or more of the following methods:

- Change hardcoded page size references to symbolic values (assigned at run time using a call to $GETSYI).
- Reevaluate code that assumes that page size and disk (file) block size are equal. On AXP systems, this assumption is not correct.
- Do not depend on being able to use memory-management-related system services (for example, $CRMPSC, $MGBLSC) to map a file into a fixed, page-size-dependent range of addresses (global section). Consider instead using the $EXPREG system service.

For more information on page size, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.2.4 Order of Read/Write Operations on Multiprocessor Systems

The VAX architecture specifies that if a processor in a multiprocessing system writes multiple data items to memory, these items are written to memory in the order specified. This ordering ensures that the writes become visible to other CPUs in the order in which they were specified by the program and I/O devices.

The guarantee that writes become visible to other CPUs in the same order in which they are specified limits the performance optimization that the system can make. It also makes caches more complex and limits the optimization of cache performance.

To benefit overall system performance, AXP systems, as well as other RISC systems, can reorder reads and writes to memory. Therefore, writes to memory can become visible to other CPUs in the system in an order different from that in which they were issued.

Note

This section is relevant only to multiprocessor systems. On a uniprocessor system, all memory accesses are completed in the order in which the program requested them.

Finding the Problem

To find instances of reliance on read/write ordering for applications that may execute on multiprocessor systems, identify algorithms that depend upon the order in which data is written: for example, use of flag-passing protocols for synchronization.

Eliminating the Problem

To eliminate problems with the ordering of read and write operations, you will be able to use one or more of the following methods:

- Instead of flag-passing protocols, use system-supplied routines for synchronization, such as those in the Parallel Processing Run-Time Library (PPL$) or the OpenVMS locking system services ($ENQ, $DEQ).

- The Alpha AXP architecture specifies a memory barrier instruction, which causes the hardware to complete all previous memory reads and writes before performing reads and writes following the barrier. Some AXP languages provide a way of inserting this instruction, but its use will degrade performance.

For more information on synchronization, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.3 Immediacy of Arithmetic Exception Reporting

AXP (and vector VAX) systems treat exceptions differently from scalar VAX systems. Scalar VAX systems use "precise exception reporting"; that is, they guarantee that if an instruction causes an exception, the **program counter (PC)** that is reported is the address of the instruction that caused the exception. Because no subsequent instructions have been issued or have affected the context of the process, a condition handler can remedy the cause of the exception and resume execution of the program at or after the failing instruction.

To achieve the best possible performance in a pipelined environment, vector VAX and AXP systems use "imprecise exception reporting"; that is, the PC reported by the exception handler is not guaranteed to be that of the instruction that caused the arithmetic exception. Furthermore, subsequent instructions may complete before the exception is reported.

In practice, very few, if any, programs rely on knowing the specific instruction that caused an arithmetic exception. Typically, when an arithmetic exception occurs, a program does one of the following:

- Logs the exception and continues
- Prints an error message and aborts the subroutine or program
- Restarts the entire subroutine and uses a different algorithm that scales the data to prevent overflow or underflow

If a VAX program performs one of these actions upon encountering an arithmetic exception, it will not be affected by being migrated to a RISC system that uses imprecise exception handling.

Note

Imprecise exception reporting applies only to arithmetic exceptions. For other types of exceptions, such as faults and traps, OpenVMS AXP uses precise exception reporting, and the specific instruction that caused the exception is reported.

Finding the Problem

To find instances of reliance on precise exception reporting, check for the presence of arithmetic exception handlers. Check whether the handler depends on having the exact program counter (PC) or only needs to know what procedure caused the exception.

Eliminating the Problem

To eliminate the use of precise exception handling, avoid writing arithmetic condition handlers that depend upon knowing the exact instruction that caused an arithmetic exception.

For more information on exception handling, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

For compatibility, most AXP compilers provide a compiler option that allows a programmer to specify at compile time whether or not precise exception reporting is required. Precise exception reporting severely impairs the performance of an application. If only certain operations in an application require precise exception reporting, you should use this option to compile only the portions of the application that contain those operations. For more information, see the documentation for the individual compilers.

4.2.3.4 Explicit Reliance on the VAX Procedure Calling Standard

The OpenVMS calling standard specifies significantly different calling conventions for AXP programs than for VAX programs. Application programs that depend explicitly on certain details of the VAX procedure calling conventions must be modified to run as native code on an AXP system. Such dependencies include:

- Code that locates the placement of arguments relative to the argument pointer (AP)

  In many cases, however, the VAX MACRO–32 Compiler for OpenVMS AXP compensates for this.

- Code that modifies its return address on the stack
- Code that interprets the contents of a call frame

Both VAX and AXP compilers provide techniques for accessing procedure arguments. If your code uses these standard mechanisms, you can simply recompile it to run correctly on an AXP system. If your code does not use these mechanisms, you must rewrite it so that it does. For a description of these standard mechanisms, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

Translated code mimics the behavior of VAX procedure calling. Images that contain the dependencies listed here will execute properly under translation on an AXP system.

4.2.3.5 Explicit Reliance on VAX Exception-Handling Mechanisms

The mechanics of exception handling differ between VAX and AXP systems. *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications* discusses the differences in how arithmetic exceptions are dispatched on VAX and AXP systems. This section focuses on the mechanisms by which code dynamically establishes a condition handler and by which a condition handler accesses the exception state.

4.2.3.5.1 Establishing a Dynamic Condition Handler

VAX systems provide a number of ways in which an application can establish a condition handler dynamically at run time. The OpenVMS calling standard facilitates this operation for VAX programs by providing a space at the top of a call frame in which executing code can place the address of a condition handler that is to service exceptions that occur in the context of that frame. However, the OpenVMS calling standard provides no such writable area in the procedure descriptor for AXP procedures.

For instance, a VAX MACRO program might utilize the following instruction sequence to move the address of a condition handler into a call frame:

```
MOVAB    HANDLER,(FP)
```

The Macro–32 Compiler for OpenVMS Alpha AXP parses this statement and generates appropriate AXP assembly language code that results in the establishment of the condition handler. For more information, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*.

Note

On VAX systems, the run-time library routine LIB$ESTABLISH and its counterpart LIB$REVERT allow an application to establish and disestablish a condition handler for the current frame. These routines do not exist on an AXP system; however, compilers may handle these calls properly (such as with FORTRAN intrinsic functions). For more precise information, see the documentation for the compilers relevant to your application.

Translated code mimics the VAX mechanism for dynamically establishing a condition handler.

Certain AXP compilers (and cross-compilers) provide a language-specific mechanism to establish a dynamic condition handler.

For more information on condition handlers, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.5.2 Accessing Data in the Signal and Mechanism Arrays

During exception processing, both VAX and AXP systems push the exception processor status, an exception PC, a signal array, and a mechanism array onto the stack.

Both the signal array and mechanism array have different contents and field lengths on VAX and AXP systems. To work properly in either system, a condition handler that accesses data in the signal array or the mechanism array must use the appropriate CHF$ symbols rather than hardcoded offsets. For descriptions of the appropriate CHF$ symbols, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

Note

The condition handler cannot successfully locate information in the mechanism array by using hardcoded offsets from AP.

4.2.3.6 Explicit Reliance on the VAX AST Parameter List

OpenVMS VAX passes five longword arguments to an AST service routine. AST service routines written in VAX MACRO access this information by using offsets from the argument pointer (AP). OpenVMS VAX allows an AST service routine to modify any of these arguments, including the saved registers and the return PC. These modifications can then affect the interrupted program once the AST routine returns.

Although the AST parameter list on AXP systems also consists of five parameters, the only argument directly intended for the AST procedure is the AST parameter. Although the other arguments are present, they are not used after the AST procedure exits. Because modifying them has no effect on the thread of operation to be resumed at AST exit, a program that relies on such an effect must be changed to use more conventional argument-passing mechanisms to run on an AXP system. For information on migrating AST service routines, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

4.2.3.7 Explicit Dependency on the Form and Behavior of VAX Instructions

Programs that rely specifically on the execution behavior of VAX MACRO instructions or on binary encoding of VAX instructions must be modified before being recompiled or relinked to run as native code on an AXP system.

For example, the following coding practices will not work on an AXP system:

- In VAX MACRO, embedding a block of VAX instructions in a program data area, and modifying a PC to transfer control to this code block
- Examining condition codes or other information in the processor status word (PSW)

For more information on migrating VAX MACRO code, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code.*

4.2.3.8 Generation of VAX Instructions at Run Time

Creating and executing conventional VAX instructions will not work in native AXP mode.

VAX instructions created at run time will execute by emulation in a translated image.

For more information on code that generates specific VAX instructions at run time, see *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code.*

4.3 Identifying Incompatibilities Between VAX and AXP Systems

To identify architectural incompatibilities in a module of your application, start by doing a test compile of the module using the AXP compiler. For information on diagnostic compiler switches, see your language processor documentation.

Many modules will compile and run with no errors. If errors occur, you may have to revise the module.

Note

Even if a module runs without error in isolation, there may be latent synchronization problems that will turn up only when the module is run together with other parts of the application.

If a module does not run without error after being recompiled and relinked, you can use the following methods to assess what must be revised to make the program run well on an AXP system:

- Examining the source code

A code review at this point can avoid many difficulties in the migration process and save a great deal of time and effort in the later stages of migration. To examine your code, use the checklist in Appendix A, as well as the guidelines in *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*. These migration issues are summarized in Section 4.2.

If a direct code review of your entire application is not practical, an automated search can still be useful: for example, using a combination of DCL SEARCH and an editor to locate and tabulate instances of architectural dependencies.

- Using messages generated by the compiler in your initial test run

  Compilers will give you information on:

  – Differences between VAX and AXP compilers

  – Data alignment

  Specific compilers may also identify other differences between the VAX and Alpha AXP architectures.

- Analyzing the image using VEST

  Even if you intend to recompile and relink a program, you can use VEST as an analysis tool. It can provide a great deal of useful information about changes that will make your program run most efficiently on an AXP system. For example, VEST can help identify the following problems:

  – Static unaligned data (data declarations, including unaligned fields in data structures) and unaligned stack operations

  – Floating-point references (H_ and D_floating)

  – Packed decimal references

  – Privileged code

  – Nonstandard coding practice

    – References to OpenVMS data or code other than by using system services

    – Uninitialized variables

  – Certain synchronization issues, such as multiprocess use of interlocked instructions

  VEST cannot identify some problems, including:

  – Unaligned variables (in data structures created dynamically)

  – Most synchronization issues

- Running the image using the PCA (Performance and Coverage Analyzer)

  The PCA can point out the following issues:

  – Run-time alignment faults

  – Which sections of the application are executed most frequently and hence are critical to performance

Once all the images of the application run without errors on an AXP system, you must combine the rebuilt images to test for problems of synchronization between images. For more information on testing, see Section 6.3.2.

4.4 Deciding Whether to Recompile or Translate

If both methods are possible for a given image, you must balance the projected performance of native and translated versions of the image on an AXP system against the effort required to translate the image or to convert it to a native AXP image.

In general, different images that make up an application can be run in different modes: for example, a native AXP image can call translated shareable images and vice versa. For more information on mixed-architecture applications, see Section 4.4.2.

The two migration paths are compared in Table 4–1.

Table 4–1 Migration Path Comparison

Factor	Recompile/Relink	Translate
Performance	Full AXP capability	Typically 25-40% of native AXP potential; equivalent to performance on VAX
Effort required	Varies: easy to difficult	Easy
Schedule constraints	Based on availability of native compilers	None: available immediately
Programs supported		
–Age	Source for VAX VMS Version 4.0 or earlier accepted	Only OpenVMS VAX Version 4.0 or later supported
–Limitations	Privileged code supported	Only user-mode code supported
VAX compatibility	High: most code will recompile and relink without difficulty	Complete via emulation
Ongoing support and maintenance	Normal source code maintenance	No source maintenance

To determine how to proceed with the migration of your application, answer the following questions:

- Do you build your application entirely from source code, or do you rely on binary images for some functions?

  If you rely on binary images, you will have to translate them.

- Do you have access to the source code for all images that are part of your application?

  If not, you will have to translate those images with missing source code.

- Which images are critical to the performance of your application?

  You will want to recompile those images to take full advantage of the speed of AXP systems.

  – Use the Performance and Coverage Analyzer to identify critical images.

  – Only images that are produced by native AXP compilers use AXP processing capabilities efficiently and achieve optimal performance. A translated VAX image runs at one-third the speed of native AXP code or slower, depending on the translation options used.

- How much work will be required to convert each image under the two methods?

  – Depending on the complexity of the application, software translation usually requires less effort and time than recompiling and relinking.

    You may choose to translate some part of your application in order to get it running on OpenVMS AXP while you complete the migration to an all-native version.

  – Code that depends on details of the VAX architecture and the VAX calling standard cannot be recompiled directly. It must either run under translation, or it must be rewritten, recompiled, and relinked.

You can remove architectural dependencies in several ways:

- Replace an architecture-dependent code sequence with high-level-language lexical elements that support the same operation in a platform-independent manner.
- Use a call to an OpenVMS system service to perform the task in a way that is appropriate for the processor architecture.
- Use a high-level-language compiler switch to help guarantee correct program behavior with minimal changes to the source code.

Table 4–2 summarizes how the architectural dependencies of a given program can affect which method you use to migrate the program to an AXP system. For more detailed information, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

Table 4–2 Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Data alignment[1]	
By default, most compilers align data naturally. For information on qualifiers to retain VAX alignment, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.	Unaligned data supported, but the qualifier /OPTIMIZE=ALIGNMENT can improve overall execution speed by assuming that data is longword aligned.

[1]Unaligned data is primarily a performance issue. Whereas references to unaligned data were only somewhat detrimental to VAX performance, loading unaligned data from memory and storing unaligned data to memory in an AXP system can be up to 100 times slower than the corresponding aligned operations.

(continued on next page)

Table 4–2 (Cont.) Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Data types	
Replace H_floating with G_floating or IEEE T_floating. For D_floating, if the 15 decimal digits of precision provided by the D53 format are sufficient, replace D_floating with G_floating. If the application requires 16-bit decimal precision (D56 format), translate it. COBOL packed decimal is automatically converted to binary format for operations.	To retain 16-bit decimal precision for D_ floating, use the /FLOAT=D56_FLOAT qualifier. Performance using this qualifier will be slower than when using the default, /FLOAT=D53_FLOAT.
Atomicity of read-modify-write operations	
Support depends on options provided by the individual compiler. (For more information, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.)	Use the /PRESERVE=INSTRUCTION_ ATOMICITY qualifier. Execution speed may drop by a factor of 2.
Atomicity and granularity of byte and word write operations	
Supported, using compiler options with appropriate source code changes. (For more information, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.)	Use the /PRESERVE=MEMORY_ATOMICITY qualifier. Execution speed may drop by a factor of 2.
Page size	
The OpenVMS Linker produces large, AXP style pages by default.	Most 512-byte page images are supported. However, because of the permissive protection assigned by VEST, images that rely on restrictive protection to generate access violations will not execute properly on an AXP system when translated.
Read/write ordering	
Supported by adding appropriate synchronization instructions (MB) to source code, but with a performance penalty.	Use the /PRESERVE=READ_WRITE_ ORDERING qualifier.
Immediacy of exception reporting	

(continued on next page)

Table 4–2 (Cont.) Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Immediacy of exception reporting	
Partly supported using compiler options. (For more information, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.)	Use the /PRESERVE=FLOAT_EXCEPTIONS or the /PRESERVE=INTEGER_EXCEPTIONS qualifier. Execution speed may drop by a factor of 2.
Explicit reliance on details of the VAX architecture and calling standard[2]	
Unsupported; dependencies must be removed.	Supported.

[2]Dependencies on details of the VAX architecture and calling standard include explicit reliance on the VAX calling standard, VAX exception handling, the VAX AST parameter list, the format and behavior of VAX instructions, and the generation of VAX instructions at run time.

4.4.1 Translating Your Application

If you are unable to recompile your application, or if it uses features specific to the VAX architecture, you may decide to translate the application. You can choose to translate only some parts of the application, or you can translate parts of it temporarily as a means of staging the overall migration.

Many of the differences that affect recompilation discussed in Section 4.2 can also affect the performance of a translated VAX image. You can use the following methods to increase the compatibility of images that are dependent on the VAX architecture. (For more information, see *DECmigrate for OpenVMS AXP Systems Translating Images*.)

- Data alignment

  Supply the VEST translate-time qualifier /OPTIMIZE=NOALIGNMENT to make VEST generate extra inline AXP code that avoids generating exceptions for references to unaligned data. With this qualifier, VEST produces AXP code that executes about 10 times slower than code that uses only aligned data references. (If you use the default option /OPTIMIZE=ALIGNMENT, unaligned data causes an exception, which takes about 100 times longer to execute than with aligned data.)

- Instruction atomicity

  When you invoke the translator, supply the translate-time qualifier /PRESERVE=INSTRUCTION_ATOMICITY to make VEST generate an Alpha AXP instruction sequence that is AST atomic for a specified set of VAX instructions. Although an AST can be delivered in the middle of an Alpha AXP instruction sequence that performs such an atomic operation, the instruction sequence will be restarted at the beginning when the AST routine completes. Execution speed for a particular code sequence may drop by a factor of 2 if the /PRESERVE=INSTRUCTION_ATOMICITY qualifier is specified. (For a list of VAX instructions for which the translator generates AST-atomic code, as well as additional information about the software translator, see *DECmigrate for OpenVMS AXP Systems Translating Images*.)

- Read/write granularity

  VEST ensures the atomicity of byte or word writes when you use the translate-time qualifier /PRESERVE=MEMORY_ATOMICITY. This qualifier allows a mainline routine and an AST routine to update adjacent bytes within a longword or quadword concurrently without interfering with each other. The /PRESERVE=MEMORY_ATOMICITY qualifier guarantees atomic access of longwords that are not naturally aligned and of data that crosses quadword boundaries. Execution speed may drop by a factor of 2 when these qualifiers are specified.

- Page size and permissive protection

  To enable VAX images to run on an AXP system, VEST, together with the image activator, maps the VAX image sections into large pages. With an AXP processor that supports 8 KB pages, up to 16 VAX pages can fit in a single page. However, because this big page is described by only a single page-table entry, only one protection and a single backing store can be assigned to the page. Consequently, VEST assigns the AXP page the most permissive protection associated with any of the AXP image sections that it maps. Thus, VAX images that rely on restrictive protection to generate access violations will not execute properly on an AXP system when translated.

  One possible alternative is to relink the program on a VAX using the default linker option /BPAGE to align the pages on 64KB boundaries.

- Precise arithmetic exceptions

  VEST allows you to set precise exception reporting for certain exception types at translate time by using the /PRESERVE=FLOAT_EXCEPTIONS qualifier or the /PRESERVE=INTEGER_EXCEPTIONS qualifier. If you specify either of these qualifiers, execution speed for certain code segments may drop by a factor of 2.

- Generating VAX instructions at run time

  VAX instructions created at run time will execute by emulation under translation. However, emulated instructions are significantly slower than translated instructions, which can be important if the code is generated at run time to speed up the performance of critical sections of your application.

Table 4–2 includes a summary of ways you can allow for various architectural dependencies in a translated image.

4.4.2 Combining Native and Translated Images

In general, you can combine native AXP images with translated images on an AXP system. For example, a native AXP image can call translated shareable images and vice versa.

In order to run together, native and translated images must be able to make calls between the VAX and Alpha AXP calling standards. No special action is required if the native and translated images meet the following conditions:

- Routine interface semantics and data alignment conventions for the native AXP image are identical to those on a VAX image.
- All the entry points are CALL*x*; that is, there are no external JSB entry points. This is probably true of any code written in a high-level language.

When a source procedure that uses one calling standard calls a destination procedure that uses a different calling standard, it does so indirectly through a **jacket routine**. The jacket routine interprets the procedure's call frame and argument list and builds the equivalent destination call frame and argument list, then transfers control to the destination procedure. When the destination procedure returns, it does so through the jacket routine. The jacket routine propagates the destination routine's returned register values into the source routine's registers and returns control to the source procedure.

The OpenVMS AXP operating system creates jacket routines automatically for most calls. To make use of automatic jacketing, use the compiler qualifier /TIE and the linker option /NONATIVE_ONLY to create the native AXP parts of your application.

In certain cases, the application program must use a specially written jacket routine. For example, you may have to write jacket routines for nonstandard calls to libraries such as the following:

- A VAX shareable library that includes external JSB entry points
- A library that includes read/write data in the transfer vector
- A library that contains VAX specific functions
- A library that uses resources that would need to be shared between a native and a translated version of the library
- A native AXP library that does not provide or export all the symbols that the VAX image did

  (The term exported means that a routine is included in the Global Symbol Table (GST) for the image.)

For information on how to create a jacket image for one of these situations, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

Translated shareable images (such as run-time libraries for languages without native AXP compilers) that are shipped with the OpenVMS AXP operating system are accompanied by jacket routines that allow them to be called by native AXP images.

5
Writing a Migration Plan

When you have evaluated your application and decided on the migration methods you will use, you will be ready to write a migration plan. A migration plan details the results of the technical analysis of the application, including portability issues and hardware and software dependencies, outlines migration milestones, and specifies who will perform the different parts of the migration. The information in the migration plan will help you make technical and business decisions about the future portability of the application and possibly about optimizing it to improve its performance.

Outline for a Typical Migration Plan

The remainder of this chapter is an outline of a typical migration plan. For a sample migration plan, see Appendix B.

I Executive Summary

An overview of the migration. Describe the goals of the migration effort: address functionality, quality, time to market, and performance goals. List the major platforms on which the application currently runs, and include the languages used in the application. Describe how application dependencies or risks can affect the migration goals and completion. Give overall resource requirements and estimated completion date.

A. Identity of the application

Name, owner

B. Function of the application

What does the application do?

C. General plan of the migration

Include the relationship to your other release plans.

Use of translation:

- To stage the migration of the application
- To prepare for the eventual availability of native compilers

II Technical Analysis

This section defines the scope of the work to migrate the application to OpenVMS AXP, based on the results of the image and source analyses.

A. Application Characteristics

A general description of the application, including the number of images, procedures, and data files it contains; the number of lines of code and number of modules; the languages used and the percentage and particular function of each—if VAX MACRO is used, state why; whether the application is generally portable or dependent on the VAX architecture;

and any special hardware or software that is required to run or modify the application.

B. Results of Image Analysis

- Images tested
- VEST messages/errors found

C. Results of Source Analysis

A general discussion of what source analysis was done and the prominent issues found.

Use the questionnaire in Appendix A.

D. Migration Issues

Describe dependency of your application on each of the following, if relevant:

- Data alignment
- VAX data types
- Read/write granularity
- Page size
- Read/write ordering
- Immediate exception reporting
- VAX procedure calling standard
- VAX exception-handling mechanisms
- VAX AST parameter list
- Form and behavior of VAX instructions
- Generation of VAX instructions at run time

III Milestones and Deliverables

Identify the major intermediate goals of the migration and provide a projected schedule.

IV Technical Approach

Describe the issues to be addressed for each stage of the migration and compare with the milestones in the preceding section: order of addressing the main issues, who will deal with them, and where? Will you use Digital's migration services?

For example:

A. Correct misaligned memory references

B. Correct page size dependencies

C. Correct non-ANSI coding constructs

D. Repair atomicity problems

E. Eliminate other dependencies

Include the following information, if relevant:

A. The need to write jacket routines, in cases where they will not be created automatically

B. Special set-up procedures for compiling, linking, or translating the application; for example, using the compiler qualifier /TIE or the linker option /[NO]NATIVE_ONLY.

How much work will be required for each phase of the migration?

V Dependencies and risks

Other software; hardware; support from other organizations

VI Resource Requirements

A. Hardware

B. On-site training

C. Telephone support

D. Engineering assistance

E. Testing assistance

6
Migrating Your Application

Actually migrating your application to an AXP system involves several steps:

- Setting up the migration environment
- Testing the application on a VAX system to establish baselines for evaluating the migration
- Converting the application to run on an AXP system
- Debugging and testing the migrated application
- Integrating the migrated application into a software system

6.1 Setting Up the Migration Environment

The native AXP environment is a complete development environment equivalent to that on VAX systems. (Some Digital compilers are not yet available on AXP systems, however.)

At present, you will have to complete the debugging and testing of your migrated application on AXP hardware.

An important element of the AXP migration environment is support from Digital, which can provide help in modifying, debugging, and testing your application.

6.1.1 Hardware

There are several issues to consider when planning what hardware you will need for your migration. To begin, consider what resources are required in your normal VAX development environment:

- CPUs
- Disks
- Memory

To estimate the resources needed for an AXP migration environment, consider the following issues:

- Greater image size on AXP systems

  Compare VAX and AXP compiled and translated images.
- Greater page size and physical memory size on AXP systems
- CPU requirements

Using VEST tends to take a lot of CPU time. (It is difficult to predict how much; it depends more on application complexity than size.) VEST also needs a great deal of disk space for log files, for an AXP image if you request one, for flowgraphs, and so on. The new image includes both the original VAX instructions and the new Alpha AXP instructions, so it is always larger than the VAX image.

A suggested configuration consists of:

- 6 VUP multiprocessing system with 256 MB of memory
- 1 GB system disk
- 1 GB disk per application

In a multiprocessing system, each processor should be able to support the image analysis of a separate application.

If computer resources are scarce, Digital suggests that you do one or more of the following:

- Run compilers or VEST as a batch job at off-peak hours.
- Perform your migration at an AXP Migration Center (AMC).
- Lease additional equipment for the migration effort.

6.1.2 Software

To create an efficient migration environment, check the following elements:

- Migration tools

  You need a compatible set of migration tools, including the following:

  - Compilers
  - Translation tools
    - VEST and VEST/DEPENDENCY
    - TIE
  - RTLs
  - System libraries
  - Include files for C programs

- Logical names

  Logical names must be consistently defined to point to VAX and AXP versions of tools and files. For more information, see Section 6.4.

- Compile and link procedures

  These procedures must be adjusted for new tools and the new environment.

- Tools for maintaining sources and building images
  - CMS
  - MMS

Native AXP Development

All of the standard development tools you have on VAX are also available as native tools on AXP systems.

Translation

The software translator VEST runs on both VAX and AXP systems. The Translated Image Environment (TIE), which is required to run a translated image, is part of OpenVMS AXP, so final testing of a translated image must either be done on an AXP system or at an AXP Migration Center.

6.2 Converting Your Application

If you have thoroughly analyzed your code and planned the migration process, this final stage should be fairly straightforward. You may be able to recompile or translate many programs with no change. Programs that do not recompile or translate directly will frequently need only straightforward changes to get them up on an AXP system.

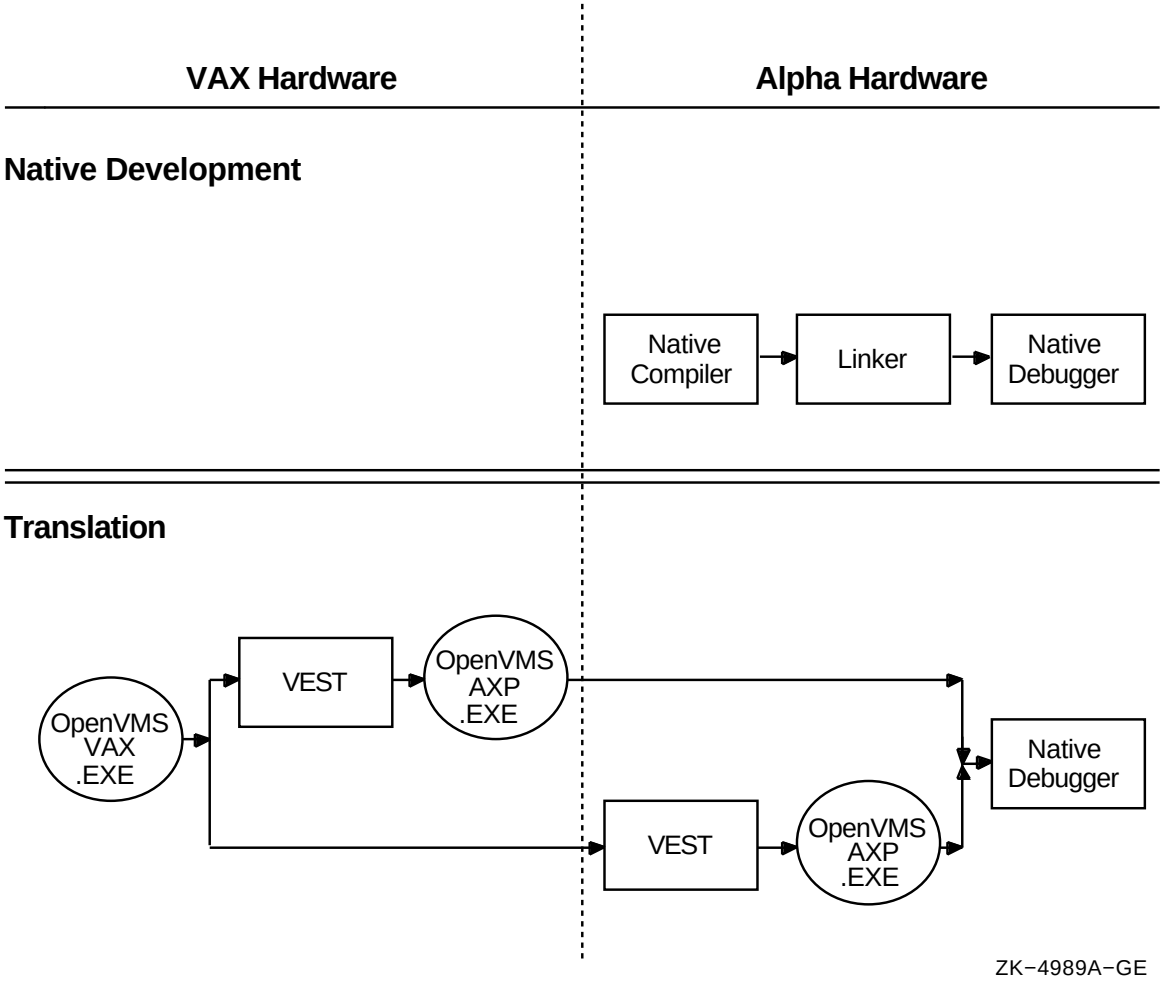
For more detailed information on the actual conversion of your code, see the following OpenVMS AXP migration documentation:

- *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*
- *DECmigrate for OpenVMS AXP Systems Translating Images*
- *Migrating to an OpenVMS AXP System: Porting VAX MACRO Code*

For descriptions of these books, see the Preface of this manual.

The two migration environments and the principal tools used in each are summarized in Figure 6–1.

Figure 6–1 Migration Environments and Tools



6.2.1 Recompiling and Relinking

In general, migrating your application involves repeated cycles of revising, compiling, linking, and debugging your code. During the process, you will resolve all syntax and logic errors noted by the development tools. Syntax errors are usually simple to fix; logic errors typically require significant modifications to your code.

Your compile and link commands will require some changes, such as new compiler and linker switches. For example, to allow portability among different AXP platforms, the linker default page size for AXP systems is 64 KB, which allows any OpenVMS AXP image to run on any AXP processor, regardless of the system page size on that processor. Also, AXP shareable images declare their universal entry points and symbols by means of a symbol vector declaration in a linker options file, not by means of the VAX transfer vector mechanism.

A number of native compilers and other tools are available for software development and migration on an AXP platform.

6.2.1.1 Native AXP Compilers

Recompiling and relinking an existing VAX source produces a native AXP image that executes within the AXP environment with all the performance advantages of a RISC architecture. For AXP code, Digital is using a series of highly optimizing compilers. These compilers have a common optimizing code generator. However, they use a different front end for each language, each of which is compatible with a current VAX compiler.

For OpenVMS AXP Version 6.1, native AXP compilers are available for the following languages:

- Ada
- BASIC
- C
- C++
- COBOL
- FORTRAN
- Pascal
- PL/I
- MACRO-32 (cross compiler)

Later releases of OpenVMS AXP will provide native compilers for other languages, including LISP.

VAX user-mode programs that are written in any other language can be run on an AXP system by translating them with VEST. Compilers for other languages may be available through third-party vendors.

In general, the AXP compilers provide command-line qualifiers and language semantics to allow code with dependencies on the VAX architecture to run on an AXP system with little modification. For a list of such dependencies, see Table 4–2. For more detailed information, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.

Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications describes in greater detail the process of using AXP compilers to migrate VAX programs to an AXP system.

6.2.1.2 Other Development Tools

Several other tools in addition to the compilers are available to create native AXP images:

- OpenVMS Linker

  The OpenVMS Linker can now accept VAX object files or AXP object files to produce either a VAX image or an AXP image. It also functions as a cross linker, since it can produce AXP images while running on VAX hardware.

- OpenVMS Debugger

  The OpenVMS Debugger running on OpenVMS AXP has the same command interface as the current OpenVMS VAX debugger. The graphical interface on OpenVMS VAX systems is also available on OpenVMS AXP systems.

- OpenVMS Librarian utility

  The OpenVMS Librarian utility creates either VAX or AXP libraries.

- OpenVMS Message utility

  The OpenVMS Message utility allows you to supplement the OpenVMS system messages with your own messages.

- MACRO–64 Assembler for OpenVMS AXP

  The MACRO–64 Assembler creates native AXP assembly language.

- ANALYZE/IMAGE

  The Analyze/Image utility can analyze either VAX or AXP images.

- ANALYZE/OBJECT

  The Analyze/Object utility can analyze either VAX or AXP objects.

- DECset

  DECset, a comprehensive set of CASE tools, includes the Language Sensitive Editor (LSE), Source Code Analyzer (SCA), Code Management System (CMS), Module Management System (MMS), and other components.

6.2.2 Translating

The process of translating a VAX image to run on an AXP system is described in detail in *DECmigrate for OpenVMS AXP Systems Translating Images*. In general, the process is straightforward, although you may have to modify your code somewhat to get it to translate without error.

6.2.2.1 VAX Environment Software Translator (VEST) and Translated Image Environment (TIE)

The main tools for migrating VAX user-mode images to OpenVMS AXP are a static translator and a run-time support environment:

- The VAX Environment Software Translator (VEST) is a utility that analyzes a VAX image and creates a functionally equivalent translated image. Using VEST, you will be able to do the following:
  - Determine whether a VAX image is translatable.
  - Translate the VAX image to an AXP image.
  - Identify specific incompatibilities with OpenVMS AXP within the image and, when appropriate, obtain information on how to correct those incompatibilities in the source files.
  - Identify ways to improve the run-time performance of the translated image.
- The Translated Image Environment (TIE) is an AXP shareable image that supports translated images at run time. TIE provides the translated image with an environment similar to OpenVMS VAX and processes all interactions with the native AXP system. Items that TIE provides include:
  - VAX instruction interpreter, which supports:
    - Execution of VAX instructions (including instruction atomicity) that is similar to their execution on a VAX system
    - Complex VAX instructions, as subroutines
  - VAX compatible exception handler
  - Jacket routines that allow communication between native and translated code

- Emulated VAX stack

TIE is invoked automatically for any translated image; you do not need to call it explicitly.

VEST locates and translates as much VAX code as possible into AXP code. TIE interprets any VAX code that cannot be converted into Alpha AXP instructions; for example:

- Instructions that VEST could not statically identify
- H- and D56 (56-bit D_floating) floating-point operations

Since interpreting instructions is a slow process, requiring perhaps 100 Alpha AXP instructions per average VAX instruction, VEST attempts to find and translate as much VAX code as possible in order to minimize the need for interpreting it at run time. A translated image runs at approximately one-third the speed of a comparable native AXP image, depending on how much VAX code TIE needs to interpret. Translated VAX images run at least as fast as they would run on equivalent (same technology) VAX hardware.

Note that you cannot specify dynamic interpretation of a VAX image on an AXP system. You must use VEST to translate the image before it can run on OpenVMS AXP.

Translating a VAX image produces an image that runs as a native image on AXP hardware. The AXP image is not merely an interpreted or emulated version of the VAX image, but contains Alpha AXP instructions that perform operations identical to those performed by the instructions in the original VAX image. The AXP .EXE file also contains the original VAX image in its entirety, which allows TIE to interpret any code that VEST could not translate.

VEST's analysis capability also makes it useful for evaluating programs that you intend to recompile, rather than translate.

See *DECmigrate for OpenVMS AXP Systems Translating Images* for a complete description of VEST and TIE. The manual explains in detail all the output that VEST generates, such as flowgraphs, and how to interpret it. The manual also explains how information provided in image information files (IIFs) created by VEST can help you improve the translated image's run-time performance.

6.3 Debugging and Testing the Migrated Application

Once you have migrated your application to OpenVMS AXP, you may have to debug it.

You will also need to test the application for correct operation.

6.3.1 Debugging

The OpenVMS operating system provides the following debuggers:

- The OpenVMS Debugger supports debugging of both VAX and native AXP programs. This debugger does not support the debugging of translated images.

  The OpenVMS Debugger is a symbolic debugger, that is, the debugger allows you to refer to program locations by the symbols you used for them in your program—the names of variables, routines, labels, and so on. You do not need to specify memory addresses or machine registers when referring to program locations, although you can if you wish.

Although the OpenVMS Debugger does not generally work for translated images, it is helpful in one area. Since the translated image mimics the VAX registers, you can use the commands SHOW CALLS and SHOW STATE to get some VAX context for more detailed debugging.

- The Delta Debugger supports debugging of VAX and AXP programs. This debugger also supports the debugging of translated images.

  The Delta Debugger is an address location debugger, that is, the debugger requires you to refer to program locations by address location. This debugger is primarily used to debug programs that run in privileged processor mode or at an elevated interrupt level.

Debugging must take place on AXP hardware.

6.3.1.1 Debugging with the OpenVMS Debugger

When you debug your migrated application on an AXP system with the OpenVMS Debugger, bear in mind the following considerations:

- You can use the debugger with programs written in any language for which there is an AXP compiler available.
- The debugger does not support debugging of installed resident images. For more information on installed resident images, see the *OpenVMS System Manager's Manual: Tuning, Monitoring, and Complex Systems*.
- The debugger does not support debugging of inlined routines. If you attempt to debug an inlined routine, the debugger issues a message that it cannot access the routine:

  ```
  DBG> %DEBUG-E-ACCESSR, no read access to address 00000000
  ```

- The debugger does not completely support the debugging of Register Frame Procedures or No Frame Procedures. If you issue STEP/OVER or STEP /RETURN commands for these procedures, unexpected results can occur.

For more information on debugging with the OpenVMS Debugger, see the *OpenVMS Debugger Manual*.

6.3.1.2 Debugging with the Delta Debugger

You can use the Delta Debugger to debug applications that are partly or completely translated.

Translated Applications

When attempting to debug a translated image, you should:

- Make sure that the program you are translating works correctly under OpenVMS VAX Version 6.1.
- Make sure that VEST and any IIF files for run-time libraries are of the same release as the version of OpenVMS AXP you are using.
- Use the VEST qualifiers /DEBUG, /LIST, and /SHOW=MACHINE_CODE to capture AXP and VAX instructions. (Note that in the listing, an asterisk identifies a VAX instruction.) Have the VAX map and listing for the VAX image at hand for comparison.

For more information on debugging translated images, contact Multivendor Customer Services.

Mixed Applications

To debug an application that is partly native AXP code and partly translated code, make sure that the native parts of the application were compiled using the /TIE qualifier; in addition, you must link the application with the /NO_NATIVE_ONLY linker option.

For more information on debugging with the Delta Debugger, see the *OpenVMS Delta/XDelta Debugger Manual*.

6.3.2 Testing

You must test your application to compare the performance and functionality of the migrated version with those of the original VAX version.

The first step in testing is to establish baseline values for your application by running your test suite on the VAX application.

Once your application is running on an AXP system, there are two types of tests you will want to apply:

- The standard tests used for the VAX version of the application
- New tests to check specifically for problems due to the change in architecture

6.3.2.1 VAX Tests

Because the changes in your application are combined with use of a new architecture, testing your application after it is migrated to OpenVMS AXP is particularly important. Not only can the changes introduce errors into the application, but the new environment may bring out latent problems in the VAX version.

Testing your migrated application involves the following steps:

1. Get a complete set of standard data for the application prior to the migration.
2. Migrate your test suite along with the application (if the tests are not already available on AXP).
3. Validate the test suite on an AXP system.
4. Run the migrated tests on the migrated application.

Both regression tests and stress tests are useful here. Stress tests are important to test for platform differences in synchronization, particularly for applications that use multiple threads of execution.

6.3.2.2 AXP Tests

While your standard tests should go a long way toward verifying the function of the migrated application, you should add some tests that look at issues specific to the migration. Points to focus on include the following:

- Compiler differences—changes in optimization and data alignment
- Architectural differences—changes in instruction atomicity, memory atomicity, and read/write ordering, for example
- Integration—modules written in different languages, or modules that had to be translated

6.4 Integrating the Migrated Application into a Software System

After you have migrated your application by recompiling or translating it, check for problems that are caused by interactions with other software and that may have been introduced during the migration.

Sources of problems in interoperability can include the following:

- AXP and VAX systems within a VMScluster environment must use separate system disks. You must make sure that your application refers to the appropriate system disk.
- Image names

  In a mixed environment, be sure that your application refers to the correct version.

  – Native VAX and native AXP versions of an image have the same name.

  – The translated version of an image has the string "_TV" added to its name.
- Recompiled images may expect naturally aligned data, while translated images have unaligned data, like the original VAX image.

A

Application Evaluation Checklist

This checklist is based on one used by Digital to evaluate applications for OpenVMS AXP.

Comments in brackets following a question are intended to help clarify the purpose of that question.

Application Evaluation Checklist

Development History and Plans

1. Does the application currently run on other operating systems or hardware architectures? ☐ YES ☐ NO

   If yes, does the application currently run on a RISC system? ☐ YES ☐ NO

   [If so, it will be easier to migrate to OpenVMS AXP.]

2. What are your plans for the application after migration?

   a. No further development ☐ YES ☐ NO

   b. Maintenance releases only ☐ YES ☐ NO

   c. Additional or changed functionality ☐ YES ☐ NO

   d. Maintain separate VAX and AXP sources ☐ YES ☐ NO

   [If you answer "YES" to a, you may wish to consider translating the application. A "YES" response to b or c should give you reason to evaluate the benefits of recompiling and relinking your application, although translation is still possible. If you intend to maintain separate VAX and AXP sources, as indicated by a "YES" to d, you may need to consider interoperability and consistency issues, especially if the different versions of the application can access the same database.]

External Dependencies

3. What is the system configuration (CPUs, memory, disks) required to set up a development environment for the application? _____

   [This will help you plan for the resources needed for migration.]

4. What is the system configuration (CPUs, memory, disks) required to set up a typical user environment for the application, including installation verification procedures, regression tests, benchmarks, or workloads? _____

   [This will help you determine whether your entire environment is available on OpenVMS AXP.]

5. Does the application rely on any special hardware? ☐ YES ☐ NO

   [This will help you determine whether the hardware is available on OpenVMS AXP, and whether the application includes hardware-specific code.]

6. a. What version of OpenVMS does your application currently run on? _____

   b. Does the application run on OpenVMS VAX Version 6.1? ☐ YES ☐ NO

c. Does the application use features that are not available on OpenVMS AXP? ☐ YES ☐ NO

[The migration base for OpenVMS AXP is OpenVMS VAX Version 6.1. If you answer "YES" to c, your application may use features that are not yet supported on OpenVMS AXP, or be linked against an OpenVMS RTL or other shareable image that is incompatible with the current version of OpenVMS AXP.]

7. Does the application require layered products to run?

a. From Digital: (other than compiler RTLs) ☐ YES ☐ NO

b. From third parties: ☐ YES ☐ NO

[If you answer "YES" to a and are uncertain whether the Digital layered products are yet available for OpenVMS AXP, check with your Digital Account Representative. If you answer "YES" to b, check with your third-party product supplier.]

Composition of the Application

8. How large is your application?

How many modules? _____

How many lines or kilobytes of code? _____

How much disk space required? _____

[This will help you "size" the effort and the resources required for migration.]

9. a. Do you have access to all source files that make up your application? ☐ YES ☐ NO

b. If you are considering using Digital Services, will it be possible to give Digital access to these source files and build procedures? ☐ YES ☐ NO

[If you answer "YES" to a, translation may be your only migration option for the files with missing sources. A "YES" answer to b allows you to take advantage of a greater range of Digital migration services.]

10. a. What languages is the application written in? (If multiple languages are used, give the percentages of each.) _____

[If the compilers are not yet available, you must translate or rewrite in a different language.]

b. If you use VAX MACRO, what are your specific reasons? _____

c. Could the function of the VAX MACRO code be performed by a high-level-language compiler or a system service (such as $GETJPI for retrieving process names)? ☐ YES ☐ NO

[Digital does not recommend the use of VAX MACRO or the Macro–64 Assembler for OpenVMS Alpha AXP in AXP applications. You may be able to replace assembly-language code in certain user-mode applications by a call to an OpenVMS system service that did not exist when the application was first written.]

11. a. Do you have regression tests for the application? ☐ YES ☐ NO

    b. If yes, do they require DEC Test Manager? ☐ YES ☐ NO

    [If you answer "YES" to a, you should consider migrating those regression tests. The DEC Test Manager is not available at the initial release of OpenVMS AXP. Contact your Digital Account Representative if your regression tests depend on this product.]

Dependencies on the VAX Architecture

12. a. Does the application use the H-floating data types? ☐ YES ☐ NO

    b. Does the application use the D-floating data types? ☐ YES ☐ NO

    c. If the application uses D-floating, does it require 56 bits of precision (16 decimal digits) or would 53 bits (15 decimal digits) suffice? ☐ 56 bits ☐ 53 bits

    [If you answer "YES" to a, you must either translate your application to obtain H-floating compatibility, or convert the data to G-floating, S-floating, or T-floating format. If you answer "YES" to b, you must either translate the application to obtain full 56-bit VAX precision D-floating compatibility, accept the 53-bit precision D-floating format provided by AXP systems, or convert the data to G-floating, S-floating, or T-floating format.]

13. a. Does the application use large amounts of data or data structures? ☐ YES ☐ NO

    b. Is the data byte, word, or longword aligned? ☐ YES ☐ NO

    [If you answer "YES" to a, but "NO" to b, you should consider aligning your data naturally to achieve optimal AXP performance. You must align data naturally if the data is in a global section shared among a number of processes, or is shared between a main program and an AST.]

14. Does the application make assumptions about how compilers align data (that is, does the application assume that data structures are: packed, aligned naturally, aligned on longwords, and so forth)? ☐ YES ☐ NO

    [If you answer "YES," you should consider portability and interoperability issues resulting from differences in compiler behavior, both on the AXP platform and between the VAX and AXP platforms. Be aware that compiler defaults for data alignment vary, as do compiler switches for forcing alignment. Typically, VAX systems default to a packed style alignment, whereas AXP compilers default to natural alignment where possible.]

15. a. Does the application assume a 512-byte page size? ☐ YES ☐ NO

b. Does the application assume that a memory page is the same size as a disk block? ☐ YES ☐ NO

[If you answer "YES" to a, you should be prepared to adapt the application to accommodate the Alpha AXP page size, which is much larger than 512 bytes and varies from system to system. Avoid hard-coded references to the page size; rather, use memory management system services and RTL routines wherever possible. If you answer "YES" to b, you should examine all calls to the $CRMPSC and $MGBLSC system services that map disk sections to memory and remove these assumptions.]

16. Does the application call OpenVMS system services? ☐ YES ☐ NO

Specifically, services that:

a. Create or map global sections (such as $CRMPSC, $MGBLSC, $UPDSEC) ☐ YES ☐ NO

b. Modify the working set (such as $LCKPAG, $LKWSET) ☐ YES ☐ NO

c. Manipulate virtual addresses (such as $CRETVA, $DELTVA) ☐ YES ☐ NO

[If you answer "YES" to any of these, you may need to examine your code to determine that it specifies the required input parameters correctly.]

17. a. Does the application use multiple, cooperating processes? ☐ YES ☐ NO

If so:

b. How many processes? _____

c. What interprocess communication method is used? _____

☐ $CRMPSC ☐ Mailboxes ☐ SCS ☐ Other
☐ DLM ☐ SHM, IPC ☐ SMG$ ☐ STR$

d. If you use global sections ($CRMPSC) to share data with other processes, how is data access synchronized? _____

[This will help you determine whether you will need to use explicit synchronization, and the level of effort required to guarantee synchronization among the parts of your application. Use of a high-level synchronization method generally allows you to migrate an application most easily.]

18. Does the application currently run in a multiprocessor (SMP) environment? ☐ YES ☐ NO

[If you answer "YES," it is likely that your application already employs adequate interprocess synchronization methods.]

19. Does the application use AST (asynchronous system trap) mechanisms? ☐ YES ☐ NO

[If you answer "YES," you should determine whether the AST and main process share access to data in process space. If so, you may need to explicitly synchronize such accesses.]

20. a. Does the application contain condition handlers? ☐ YES ☐ NO

b. Does the application rely on immediate reporting of arithmetic exceptions? ☐ YES ☐ NO

[The Alpha AXP architecture does not provide immediate reporting of arithmetic exceptions. If your handler attempts to fix the condition and restart the instruction sequence that led to the exception, you will need to alter the handler.]

21. Does the application run in privileged mode or link against SYS.STB? ☐ YES ☐ NO

If so, why? _____

[If your application links against the OpenVMS executive or runs in privileged mode, you must rewrite it for it to work as a native AXP image.]

22. Do you write your own device drivers? ☐ YES ☐ NO

[User-written device drivers are not supported in the initial release of OpenVMS AXP. Contact your Digital Account Representative if you need this feature.]

23. Does the application use connect-to-interrupt mechanisms? ☐ YES ☐ NO

If yes, with what functionality? _____

[Connect-to-interrupt is not supported on OpenVMS AXP systems. Contact your Digital Account Representative if you need this feature.]

24. Does the application create or modify machine instructions? ☐ YES ☐ NO

[Guaranteeing correct execution of instructions written to the instruction stream requires great care on OpenVMS AXP.]

25. What parts of the application are most sensitive to performance? I/O, floating point, memory, realtime (that is, interrupt latency, and so on). _____

[This will help you determine how to prioritize work on the various parts of your application and allow Digital to plan performance enhancements that are most meaningful to customers.]

B

Sample Migration Plan

The following migration plan is for a fictitious application, but is based on actual migration plans written for customer applications.

Migration Plan for Omega-1
Omega Corporation

B.1 Executive Summary

Omega-1 is an enterprise-wide information system for accessing, analyzing, managing, and presenting data.

Omega-1 has more than 4 million lines of source code. Most of the source code, written in the C programming language, is common to a variety of platforms and is considered highly portable.

However, Omega-1 has a set of routines, unique to each platform, that is implemented in VAX C and VAX MACRO for the VAX platform (about 350,000 lines of code). These routines present a number of VAX architectural dependency issues, described in Section B.2.4, that require resolution for successful migration. Resolution of these issues will involve significant work including design changes, but none of these appear to jeopardize shipping Omega-1 to customers in December 1992.

Omega-1 supports connections to a number of Digital and third-party products. Although some of these products will not be available on the AXP platform when OpenVMS AXP is first shipped, Omega Corporation, the Omega-1 vendor, has committed to migrating the base Omega-1 by that date.

Digital Services will provide support services to Omega Corporation throughout the life of the migration project as detailed in this plan. In summary, the support plan for Omega Corporation includes:

- On-site hardware and software tools for AXP development at Omega Corporation
- Engineering assistance for quality assurance testing
- Access to AXP systems at an AXP Migration Center
- Telephone access to a Digital engineer who will provide AXP technical information, support the cross-development tools, and act as a liaison for resolving any problems with Digital software products reported by Omega Corporation
- A three-day technical seminar for Omega Corporation developers at their site

One additional problem is providing the hardware for Omega to carry out an adequate field test of their products prior to the ship date. Normally, the Omega developers conduct field tests for four months before revenue shipment at 30 to 40 of their customer sites. It is unclear whether the number of required AXP units will be available for a field test of this size.

B.2 Technical Analysis

The technical analysis was performed by Omega Corporation in conjunction with Digital Services.

B.2.1 Application Characteristics

Omega-1 runs on most VAX platforms and platforms of other vendors. It consists of three layers that may or may not take advantage of specific aspects of the VAX environment. However, there are no direct dependencies on particular hardware configurations or devices.

Most of the functionality is provided in the applications layer, which contains the user interface, basic data management tools, and the Omega fourth generation language (4GL). The Omega-1 base product does not depend on any Digital or third-party layered products.

Additional products in Omega-1 are layered on top of the base product and provide expanded data management or communications functionality. These options depend on Digital or third-party layered products and will be available when the underlying products are released. The layered product dependencies are listed in Table B–3.

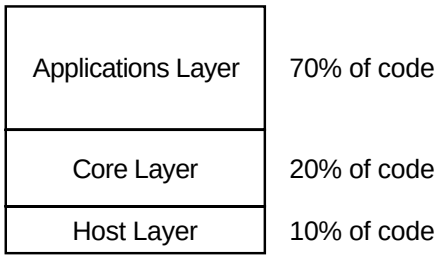
B.2.2 Software Architecture

Omega-1 is built in layers as shown in Figure B–1. This layering creates a high degree of portability for the software, because only about 10% of the system is specific to a particular implementation, and all of this code is contained within one set of modules, the host layer.

The applications and core layers are expected to run on the AXP platform simply by recompiling and relinking all of the source files. The only prerequisite is successful migration of the Omega software development tools, which are also considered quite portable and depend only on the C compiler and run-time libraries for OpenVMS AXP.

The host layer will require a number of changes such as a rewrite of some modules that contain some VAX hardware dependencies.

Figure B–1 Layer Structure of Omega-1



ZK–5174A–GE

- Applications layer

  This layer comprises the bulk of the system and is considered portable because it is implemented similarly on many platforms.

- Core layer

  This layer creates an Omega-designed set of services that conforms to the special needs of Omega-1 on each platform. Essentially, anything that is part of the Omega-1 "virtual operating system" but can be written portably resides in this layer. Components include high-level I/O, high-level memory management, character-based window systems, and the 4GL compiler with its execution environment. This layer is portable.

- Host layer

  This layer provides an interface to specific operating system elements and may be dependent on aspects of the hardware architecture. Components include:

  - I/O operation
  - Image loading and unloading
  - Memory management
  - Lightweight thread management
  - Terminal services
  - Windowing interfaces

  The host layer is different for each platform implementation. The VAX implementation is written in VAX C and VAX MACRO. This layer embodies most of the issues on which the migration project will focus.

B.2.3 Results of Image Analysis

Although Omega-1 will be recompiled and relinked, the VAX Environment Software Translator (VEST) was used to analyze 26 images of the host layer. A large number of these images had instructions that relied on the D_floating data type, which is the default for VAX C. If Omega engineers decide to move from D_floating to another floating-point format, they must be aware of the issues concerning compatibility of data files across mixed VAX and AXP VMScluster environments.

Table B–1 lists the error messages generated by the images during image analysis.

Table B–1 Image Analysis Results

Image Name	% Code Found by VEST	Major Findings
OMEGADEV60	-Fatal errors- no code found	VEST-F-PROTISD VEST-F-ISDALIGN VEST-F-ISDMIXED VEST-W-STACKMATCH Packed decimal instructions
OMECRTL	92%	VEST-W-STACKMATCH VEST-W-STKUNAL Packed decimal instructions
PSCN	64%	VEST-W-STKUNAL VEST-W-STACKMATCH

(continued on next page)

Table B–1 (Cont.) Image Analysis Results

Image Name	% Code Found by VEST	Major Findings
PVSN	65%	VEST-W-STKUNAL VEST-W-STACKMATCH

The following list describes the major findings of the Omega image analysis:

- PROTISD—user-written system service vector that indicates that an image has one or more user-written system services. This problem will be handled automatically when compiling the code using a native AXP compiler.
- ISDALIGN—the image section is not aligned on a 64 KB boundary. Before performing another VEST analysis, it will be necessary to relink the image with 64KB pages. The linker will handle this problem during the migration process.
- ISDMIXED—incompatible VAX image sections were mapped to the same 64KB block. The linker will handle this problem during the migration process.
- STKUNAL—a warning indicates that a block of code changes the stack from longword aligned to unaligned, which causes performance degradation. Omega engineers will review the logs from the VEST analysis.
- STACKMATCH—the stack may be unbalanced at a certain point. Omega engineers will review the logs from the VEST analysis.
- Packed decimal instructions—supported only with software and not with hardware, which may hinder performance of the application. Omega engineers will review the packed decimal code.

B.2.4 Results of Source Analysis

As stated previously, the migration of the applications and core layers is fairly straightforward; however, the host layer of Omega-1 contains many VAX dependencies. Discussions with Omega engineers uncovered the architectural dependencies described in the following list:

- Data alignment

  The Omega-1 software contains unaligned public data structures. To maintain source code portability, Omega engineers will compile this code with the /NOMEMBER_ALIGN qualifier of the DEC C compiler.

- Data types

  Omega-1 extensively uses floating-point calculations and data files. The VAX version uses the D56 format for all operations, which is not implemented in the native Alpha AXP instruction set. For customers who may eventually operate with mixed VAX and AXP clusters, it is best to maintain the D_ floating format. Omega is not concerned about the slight loss of floating-point precision entailed in using the 53-bit (versus 56-bit) version of D_floating available on AXP.

  However, most of the other Omega-1 implementations use the IEEE floating-point formats, which are fully supported by the Alpha AXP instruction set. Reimplementation to the IEEE format involves simply recompiling with a different qualifier, but VAX customers would then need to convert all

floating-point data in their files to the new format as part of their migration process.

- Read/write/modify atomicity

  A few AST routines need to be examined to determine whether any operations on shared variables need to be protected by explicit synchronization methods. No major problems are expected in this area.

- Granularity of byte and word operations

  The Omega-1 software has a latch that protects data that is not aligned on natural quadword boundaries. Digital engineers discussed this problem with Omega engineers and reviewed a code sample for the solution. They determined that the compilers can handle this type of access using shared data declarations and compiler directives.

- VAX page size

  The host layer includes a few routines that handle memory management on behalf of the applications. Although the existing algorithms do not actually require that the page size be 512 bytes, nevertheless, they are hard coded as 512. Correct functionality will be guaranteed by modifying these modules to query for system page size at system startup and then using the system page size for calculations involving memory management operations.

- VAX procedure calling standard

  Omega-1 "chases" the call frame stack to determine call history when a user interrupt occurs. Omega-1 modifies the return address in one of the preceding call frames to redirect flow-of-control, or accept the interrupt, when in a noncritical region of the code. Much of this is similar to what SYS$UNWIND does, except that Omega-1 does it with an AST instead of an exception handler.

  Omega-1 includes a number of functions that depend on the VAX calling standard, including Omega's own implementation of setjmp( ) and longjmp( ). These functions are written in VAX MACRO and are isolated in the operating-system code. Omega will rewrite these functions to remove dependencies on the VAX calling standard.

- Exception handling

  Omega-1 fixes illegal or faulted floating-point operations with a statistical "missing" value. It is a concern whether the current design can correctly decode the actual faulting instruction on the Alpha AXP architecture, where delivery of exception traps may be delayed.

- VAX instruction set and code generation

  The host layer includes a code generator that writes platform-native instructions into memory and executes them as part of its 4GL language. This code generator produces, among other things, "scatter/gather" code to handle database I/O operations. A portable interpreter that is "plug compatible" with the code generator can be used for the early stages of the migration project. However, a new version of the generator that produces Alpha AXP instructions will eventually have to be implemented.

B.3 Milestones and Deliverables

The ship date goal for the Omega-1 base product is December 1992. Table B–2 summarizes the major milestones and deliverables for the base product project. For a discussion of each of the deliverables, see Section B.4.

Table B–2 Milestones and Deliverables

Milestone	Responsible	Digital Role	Completion Date
Omega-1 line-mode prompt	Omega/Digital	Consulting	November 1991
New cross-image bridge	Omega/Digital	Consulting	December 1991
Floating-point decision	Omega	—	December 1991
Omega-1 exception handler	Omega/Digital	Consulting	January 1992
Begin code generator	Omega/Digital	Consulting	January 1992
Build applications layer	Omega/Digital	Consulting	January 1992
Test code generator	Omega/Digital	Run test scripts	March 1992
Test applications	Omega/Digital	Run test scripts	May 1992
Implement and test Motif user interface	Omega/Digital	Run test scripts	July 1992
Begin Omega QA and field test	Omega	On-site support	December 1992
Ship date	Omega	—	December 1992

B.4 Technical Approach

The following sections describe in detail the approach to be taken to reach each milestone of this migration project.

B.4.1 Line Mode Prompt

The first milestone is bringing Omega-1 to a line-mode prompt and entering a meaningful Omega-1 program name or command sequence for execution. Reaching this goal will demonstrate basic functionality of the development tools and run-time libraries. At this point, the host layer will be functional except for the following:

- The Omega-1 interpreter will be used instead of code generation for the 4GL functionality.
- The image bridge will be a temporary implementation.
- Exception handling will be incomplete.

Furthermore, the core layer will be functional (without the windowing user interface), and at least one Omega-1 application will be tried. This work is being done by the Omega VMS Host Group with support from Digital.

B.4.2 The Image Bridge

Omega-1 has a central bridge routine that dispatches all jumps across images. This allows Omega-1 to call routines in unloaded images, which will then be loaded dynamically. The bridge also allows images to be unloaded by Omega-1.

The image activation routines and the format of AXP object files have already established that the bridge for OpenVMS AXP can be implemented similarly to its implementation on OpenVMS VAX.

The work will be done by the Omega VMS Host Group with support from Digital, and the required changes can be completed by December 1991.

B.4.3 Floating-Point Format Decision

Omega-1 and many customers that use it rely on the D56 floating-point data type. Although it is possible to replace D56 with IEEE T_floating for increased speed, this will require that all users convert their data from one format to the other.

This is an Omega business decision, which will be made by the end of calendar year 1991.

B.4.4 Full Omega-1 Exception Handling

The next major issue to be resolved is how to perform Omega-1 exception handling on OpenVMS AXP. General exception handling, such as a run-time access violation upon opening a file, is trapped by the Omega-1 exception handler, which may then proceed with a "stack chase" on the call frames. Omega developers will make the necessary changes to account for the new calling standard on OpenVMS AXP, and will use the "setjmp" and "longjmp" features of DEC C.

Floating-point exception handling will also require design changes and reimplementation. There are a number of options for getting correct functionality, but the best answer for performance considerations is yet to be determined. Still, the Omega developers expect to solve this by the scheduled date.

B.4.5 Begin Code Generator Implementation

The final component required for a full implementation of the host layer is the code generator. To implement the code generator, the Omega developers need a complete definition of the native Alpha AXP instruction set. The code generator effort will be fully handled by the Omega OpenVMS Host Group. Digital will provide support by telephone, as needed.

B.4.6 Build Applications

Once the host layer has achieved full functionality, the applications can be built during January 1992. These applications are expected to recompile and run, since they are all written in C to very strict portability standards. Digital will provide telephone support for tools and compiler issues to the Omega developers.

B.4.7 Test Code Generator

Digital will provide assistance with debugging and functional testing for the Omega-1 code generator on an AXP system during the month of March 1992. An Omega developer will establish a test bed environment on the AXP system in an AXP Migration Center, train a Digital engineer how to run the tests, and supervise the initial set of tests. After that, Omega will send test scripts and data sets by mail for the Digital engineer to run.

B.4.8 Test Complete Application

Omega maintains a number of developmental regression test streams, which must be run on an AXP system to verify successful porting. If Omega does not have an AXP system by the time they are ready for this testing, Digital will run the regression tests against the Omega-1 base system applications on one of Digital's systems.

To do this, an Omega developer will establish a test bed environment on an AXP system in the Digital laboratory, train a Digital engineer how to run the regression tests, and supervise the initial set of tests. After that, Omega will send test scripts and data sets by mail for the Digital engineer to run. Digital will then report results back to Omega. This effort will begin in April 1992 and will be finished by the end of May.

B.4.9 DECwindows Motif User Interface

Omega needs to have the Motif developers tool kit prior to field test, so that the developers can test their Motif user interface.

B.4.10 Omega Quality Assurance and Field Test

Omega maintains an extensive set of test streams used to validate their final product. They routinely run these suites immediately prior to starting their field tests. These tests must be run on a full AXP system implementation.

At this point, some testing and optimization may be required to fix specific performance problems that become apparent only on an AXP system. Digital will provide on-site engineering support for these efforts.

B.5 Dependencies and Risks

The chief risk to the successful delivery of Omega-1 is related to Omega's quality assurance and field test processes. The developers normally conduct their field test with 30 to 40 of their customers, and it takes about four months to complete. Omega and its Digital account team need to determine how Omega can execute a testing program that meets its minimum requirements and can be completed before December 1992.

The following list shows the software dependencies for the Omega-1 base product:

- DEC C for OpenVMS AXP compiler
- VAX MACRO–32 Compiler for OpenVMS AXP
- MACRO–64 Assembler for OpenVMS AXP
- OpenVMS Debugger
- DEC C Run-Time Library
- OpenVMS Run-Time Library (LIB$)
- Screen Management Library (SMG$)
- DECTPU

Omega-1 applications also offer access to Digital or third-party data management or networking options. For example:

- Omega/Graph can use CDA tools to generate graphic images in DDIF format.
- Omega/Access can access Rdb/VMS or ORACLE databases.
- Omega/Share and Omega/Connection can access remote data with TCP/IP using the ULTRIX Connection (UCX).

Table B–3 lists all of the layered product options available to the users of Omega-1 along with the expected delivery dates.

Table B–3 Omega Optional Product Dependences

Item	Field Test	Shipping
Digital Products		
DECwindows Motif user interface (tool kit)	TBS	TBS
CDA	TBS	TBS
ALL–IN–1	TBS	TBS
Rdb/VMS	TBS	TBS
CDD/Repository	TBS	TBS
CDD/Plus	TBS	TBS
DECnet (Phase IV)	TBS	TBS
PATHWORKS	TBS	TBS
SPM	TBS	TBS
Third-Party Products		
ORACLE	TBS	TBS
Ingres	TBS	TBS

B.6 Resource Requirements

Digital resources used to support the plan outlined in Section B.3 are summarized in Table B–4 and are described in the following sections.

Table B–4 Summary of Digital Support

Resource	Time Frame	Activity	Level of Effort
On-site training	Dec 91	Training	1 engineer for 3 days
Telephone support	Dec 91–Aug 92	General support	1 engineer for 8 hours/week
Engineering assistance	Mar 92	Test code generator	1 full-time engineer for 2 weeks, half time for 4 weeks
AXP hardware	Mar 92	Test code generator	5 days/week for two weeks, 2 days/week for 4 weeks
Engineering assistance	Apr–May 92	Application testing	1 engineer, half time for 8 weeks
AXP hardware	Apr–May 92	Application testing	2 days/week for 8 weeks

(continued on next page)

Table B–4 (Cont.) Summary of Digital Support

Resource	Time Frame	Activity	Level of Effort
On-site support	Jan–Aug 92	Omega QA	1 week per month

B.6.1 Hardware

Omega requires that a system be loaned to its site to complete the migration tasks without affecting normal development activities.

The Omega-1 base product spans several RA90 drives on its development system, which is a VAX 6000 Model 550. Disk space may become a problem for building the full application set.

B.6.2 On-Site Training

Digital will assume an active role in support of Omega migration beginning in December 1991 with a three-day AXP technical seminar for the application and core-layer developers at Omega.

B.6.3 Telephone Support

During the first quarter of 1992, the Omega developers will continue their efforts to migrate the host layer at their site using the cross tools on a Digital-supplied platform. During this period, and throughout the entire migration effort, Omega will receive telephone support from a software engineer at Digital. The assigned engineer will spend approximately eight hours per week working with Omega issues, following up on bug reports, and supporting the cross tools.

B.6.4 Testing Assistance

Digital will support several phases of testing the Omega-1 application.

B.6.4.1 Testing the Code Generator

One full-time Digital engineer is required for the first two weeks of March to test the code generator. This period includes training by a Omega developer and running the initial tests. During the following four weeks, the assigned engineer will spend 50 percent of working time performing follow-up tests and reporting the results to Omega.

The code generator testing will require nearly full-time use of an AXP system during the first two weeks, followed by two additional days of AXP hardware time per week during the following four weeks.

B.6.4.2 Testing Applications

Application testing will be done at Digital during the months of April and May 1992, and will require 50 percent of an engineer's time for running regression tests and reporting results to Omega. This effort will require two days on AXP hardware per week during the eight-week test period.

B.6.4.3 Omega Quality Assurance

A Digital engineer will be available for an estimated 15 days during the months of September to November 1992 to provide on-site support at Omega.

Glossary

alignment

See *natural alignment*.

atomic instruction

An instruction that consists of one or more discrete operations that are handled by the hardware as a single operation, without interruption.

atomic operation

An operation that cannot be interrupted by other system events, such as an AST (asynchronous system trap) service routine; an atomic operation appears to other processes to be a single operation. Once an atomic operation starts, it always completes without interruption.

Read-modify-write operations are typically not atomic at an instruction level on a RISC machine.

byte granularity

A property of memory systems in which adjacent bytes can be written concurrently and independently by different processes or processors.

CISC

See *complex instruction set computer*.

compatibility

The ability of programs written for one type of computer system (such as OpenVMS VAX) to execute on another type of system (such as OpenVMS AXP).

complex instruction set computer (CISC)

A computer that has individual instructions that perform complex operations, including complex operations performed directly on locations in memory. Examples of such operations include instructions that do multibyte data moves or substring searches. CISC computers are typically contrasted with *RISC* (*reduced instruction set computer*) computers.

concurrency

Simultaneous operations by multiple agents on a shared object.

cross development

The process of creating software using tools running on one system, but targeted for another type of system; for example, creating code for AXP systems using tools running on a VAX system.

granularity

A characteristic of storage systems that defines the amount of data that can be read or written with a single instruction, or read or written independently. VAX systems have byte or multibyte granularities while disk systems typically have 512-byte or greater granularities.

image information file (IIF)

An ASCII file that contains information about the interface between VAX images. VEST uses IIFs to resolve references to other images and to generate the appropriate linkages.

image section

A group of program sections with the same attributes (such as read-only access, read/write access, absolute, relocatable, and so on) that is the unit of virtual memory allocation for an image.

interlocked instruction

An interlocked instruction performs some action in a way that guarantees the complete result as a single, uninterruptible operation in a multiprocessing environment. Since other potentially conflicting operations can be blocked while the interlocked instruction completes, interlocked instructions can have a negative performance impact.

jacket routine

A procedure that converts procedure calls from one calling standard to another; for example, calls between translated VAX images, which use the VAX calling standard, and native AXP images, which use the AXP calling standard.

load/store architecture

A machine architecture in which data items are first loaded into a processor register, then operated on, and finally stored back to memory. No operations on memory other than load and store are provided by the instruction set.

longword

Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered from right to left, 0 to 31. The address of the longword is the address of the byte containing the low-order bit (bit 0). A longword is *naturally aligned* if its address is evenly divisible by 4.

multiple instruction issue

Issuing more than one instruction during a single clock cycle.

natural alignment

Data storage in memory such that the address of the data is evenly divisible by the size of the data in bytes. For example, a naturally aligned longword has an address that is evenly divisible by 4, and a naturally aligned quadword has an address that is evenly divisible by 8. A structure is naturally aligned when all its members are naturally aligned.

page size

The number of bytes that a system's hardware treats as a unit for address mapping, sharing, protection, and movement to and from secondary storage.

pagelet

A 512-byte unit of memory in an AXP environment. On AXP systems, certain DCL and utility commands, system services, and system routines accept as input or provide as output memory requirements and quotas in terms of pagelets. Although this allows the external interfaces of these components to be compatible with those of VAX systems, OpenVMS AXP internally manages memory only in even multiples of the CPU memory page size.

PALcode

See *privileged architecture library*.

privileged architecture library (PAL)

A library of callable routines for performing instructions unique to a particular operating system. Special instructions call the routines, which must run without interruption.

processor status (PS)

On AXP systems, a privileged processor register consisting of a *quadword* of information including the current access mode, the current interrupt priority level (IPL), the stack alignment, and several reserved fields.

processor status longword (PSL)

On VAX systems, a privileged processor register consisting of a word of privileged processor status and the *processor status word* itself. The privileged processor status information includes the current interrupt priority level (IPL), the previous access mode, the current access mode, the interrupt stack bit, the trace trap pending bit, and the compatibility mode bit.

processor status word (PSW)

On VAX systems, the low-order word of the *processor status longword*. Processor status information includes the condition codes (carry, overflow, 0, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

program counter (PC)

That portion of the CPU that contains the virtual address of the next instruction to be executed. Most current CPUs implement the program counter as a register. This register is visible to the programmer through the instruction set.

quadword

Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered from right to left, 0 to 63. The address of a quadword is the address of the word containing the low-order bit (bit 0). A quadword is *naturally aligned* if its address is evenly divisible by 8.

quadword granularity

A property of memory systems in which adjacent *quadwords* can be written concurrently and independently by different processes or processors.

read-modify-write operation

A hardware operation that involves the reading, modifying, and writing of a piece of data in main memory as a single, uninterruptible operation.

read-write ordering

The order in which memory on one CPU becomes visible to an execution agent (a different CPU or device within a tightly-coupled system).

reduced instruction set computer (RISC)

A computer that has an instruction set reduced in complexity, but not necessarily in the number of instructions. RISC architectures typically require more instructions than *CISC* architectures to perform a given operation, because an individual instruction performs less work than a CISC instruction.

RISC

See *reduced instruction set computer.*

synchronization

A method of controlling access to some shared resource so that predictable, well-defined results are obtained when operating in a multiprocessing environment or in a uniprocessing environment using shared data.

translated code

The native AXP object code in a translated image. Translated code includes:

- AXP code that reproduces the behavior of equivalent VAX code in the original image
- Calls to the *Translated Image Environment (TIE)*

translated image

An AXP executable or shareable image created by *translation* of the object code of a VAX image. The translated image, which is functionally equivalent to the VAX image from which it was translated, includes both *translated code* and the original image. See *VAX Environment Software Translator*.

Translated Image Environment (TIE)

A native AXP shareable image that supports the execution of *translated images*. The TIE processes all interactions with the native AXP system and provides an environment similar to OpenVMS VAX for the translated image by managing VAX state; by emulating VAX features such as exception processing, AST delivery, and complex VAX instructions; and by interpreting untranslated VAX instructions.

translation

The process of converting a VAX binary image to an AXP image that runs with the assistance of the TIE on an AXP system. Translation is a static process that converts as much VAX code as possible to native Alpha AXP instructions. The TIE interprets any untranslated VAX code at run time.

VEST

See *VAX Environment Software Translator.*

VAX Environment Software Translator (VEST)

A software migration tool that performs the *translation* of VAX executable and shareable images into translated images that run on AXP systems. See *translated image.*

word granularity

A property of memory systems in which adjacent words can be written concurrently and independently by different processes or processors.

writable global section

A data structure (for example, FORTRAN global common) or shareable image section potentially available to all processes in the system for use in communicating between processes.

Index

A

B

C

D

E

N

O

P

Q

R

S

T

U

V

W