OpenVMS MACRO-32 Porting and User's Guide

Order Number: AA–PV64D–TE

April 2001

This manual describes how to use the MACRO-32 Compiler for OpenVMS Alpha to port VAX MACRO code to an OpenVMS Alpha system. It also describes how to use the compiler's 64-bit addressing support.

Revision/Update Information: This manual supersedes *Migrating to an OpenVMS AXP System: Porting VAX Macro Code*, OpenVMS AXP Version 1.5.

Software Version: OpenVMS Alpha Version 7.3

Compaq Computer Corporation
Houston, Texas

ZK5601

The Compaq *OpenVMS* documentation set is available on CD-ROM.

This document was prepared using DECdocument, Version 3.3-1b.

Contents

Part II Reference

E MACRO-32 Macros for 64-Bit Addressing

Preface

Intended Audience

This manual is for software engineers responsible for porting application code written in VAX MACRO. Therefore, it demands that the reader understand the OpenVMS VAX operating system and possess strong programming skills.

Document Structure

This manual is divided into two parts.

Part I: Concepts and Methodology comprises the following four chapters:

- Chapter 1, Preparing to Port VAX MACRO Code. This chapter provides a methodology for porting VAX MACRO code to an OpenVMS Alpha system.
- Chapter 2, How to Use the MACRO-32 Compiler. This chapter describes how and when to use the features of the compiler, including specialized directives and macros.
- Chapter 3, Recommended and Required Source Changes. This chapter describes how to change those coding constructs that cannot be compiled by the MACRO-32 compiler.
- Chapter 4, Improving the Performance of Ported Code. This chapter describes several compiler features that you can use to improve the performance of your ported code.
- Chapter 5, MACRO–32 Programming Support for 64-Bit Addressing. This chapter describes the 64-bit addressing support provided by the MACRO–32 compiler and associated components.

Part II: Reference comprises the following appendixes:

- Appendix A, Compiler Qualifiers
- Appendix B, Compiler Directives
- Appendix C, Compiler Built-Ins
- Appendix D, Macros for Porting to OpenVMS Alpha
- Appendix E, MACRO-32 Macros for 64-Bit Addressing

Related Documents

This manual refers readers to the following manuals for additional information on certain topics:

- *Migrating an Environment from OpenVMS VAX to OpenVMS Alpha*[1] provides an overview of the VAX to Alpha migration process and information to help you plan a migration. It discusses the decisions you must make in planning a migration and the ways to get the information you need to make those decisions. In addition, it describes the migration methods and tools available so that you can estimate the amount of work required for each method and select the method best suited to a given application.
- *Migrating an Application from OpenVMS VAX to OpenVMS Alpha* describes how to build an OpenVMS Alpha version of your OpenVMS 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 OpenVMS Alpha version. In addition, the manual describes how you can create applications in which native OpenVMS Alpha components interact with translated OpenVMS VAX components.
- *OpenVMS Calling Standard* describes the mechanisms used to allow procedure calls on OpenVMS VAX systems and OpenVMS Alpha systems.
- *VAX MACRO and Instruction Set Reference Manual* provides information about VAX instructions and the standard VAX MACRO assembly language directives.
- *OpenVMS System Messages: Companion Guide for Help Message Users* describes how to use the Help Message utility to obtain information about the VAX MACRO assembler messages and MACRO-32 compiler messages.

For additional information about OpenVMS products and services, access the following World Wide Web address:

`http://www.openvms.compaq.com/`

Reader's Comments

Compaq welcomes your comments on this manual. Please send comments to either of the following addresses:

Internet	**openvmsdoc@compaq.com**
Mail	Compaq Computer Corporation OSSG Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How to Order Additional Documentation

Use the following World Wide Web address to order additional documentation:

`http://www.openvms.compaq.com/`

If you need help deciding which documentation best meets your needs, call 800-282-6672.

[1] This manual has been archived but is available on the OpenVMS Documentation CD–ROM.

Conventions

The following conventions are 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.
$\boxed{\text{Return}}$	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.) In the HTML version of this document, this convention appears as brackets, rather than a box.
. . .	A horizontal ellipsis in examples indicates 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.
. . .	A vertical ellipsis indicates 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 command format descriptions, parentheses indicate that you must enclose the choices in parentheses if you specify more than one.
[ ]	In command format descriptions, brackets indicate optional choices. You can choose one or more items or no tiems. Do not type the brackets on the command line. However, you must include the brqackets in the syntax for OpenVMS directory specifications and for a substring specification in an assignment statement.
|	In command format descriptions, vertical bars separate choices within brackets or braces. Within brackets, the choices are optional; within braces, at least one choice is required. Do not type the vertical bar on the command line.
{ }	In command format descriptions, braces indicate required choices; you must choose one of the items listed. Do not type the braces on the command line.
bold text	This typeface represents the introduction of a new term. It also represents the name of an argument, an attribute, or a reason.
italic text	Italic text indicates important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *dd* represents the predefined code for the device type).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.

`Monospace text`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

Part I

Concepts and Methodology

1 Preparing to Port VAX MACRO Code

This chapter describes a process that software engineers can use when planning to port VAX MACRO code to an OpenVMS Alpha system. The chapter contains the following:

- Features of the MACRO-32 compiler (Section 1.1)
- Differences between the compiler and the assembler (Section 1.2)
- Step-by-step porting process (Section 1.3)
- Identifying nonportable coding practices (Section 1.4)
- Establishing useful coding conventions (Section 1.5)
- Maintaining common sources for VAX and Alpha systems (Section 1.6)

Note

The MACRO-32 Compiler for OpenVMS Alpha is provided for porting VAX MACRO code to OpenVMS Alpha. For any new development, Compaq recommends the use of mid- and high-level languages.

1.1 Compiler Features

The MACRO-32 Compiler for OpenVMS Alpha compiles VAX MACRO source code into machine code that runs on Alpha systems. While some code can be compiled without any changes, most code modules will require the addition of entry point directives. Many code modules will require other changes as well.

The compiler may detect only a few problems with a module at its initial compilation and then, after you have corrected them, the compiler may discover additional problems. In such cases, the resolution of one problem can allow the compiler to further examine the code and discover other problems the initial one concealed.

The compiler includes many features that make this process easier, such as:

- Qualifiers that allow you to control the kinds of messages the compiler generates or to enforce VAX behavior in the generated code. For example, the /FLAG qualifier enables you to specify the types of informational messages the compiler reports. Many of these messages identify porting problems, including VAX architectural dependencies. The options to the /FLAG qualifier include reporting unaligned stack and memory references and reporting unsupported directives. (For more information about the /FLAG qualifier, see Appendix A.)

- Directives that indicate routine entry points and describe them to the compiler or enforce VAX behavior for sections of code. For example, .CALL_ENTRY declares the entry point of a called routine to the compiler. Section 2.2, Section 2.4, and Chapter 3 discuss situations when the compiler *requires* special directives. Appendix B describes each directive in detail.
- Built-ins that allow you to access the Alpha instructions that perform 64-bit operations and Alpha PALcode instructions. (PALcode is shorthand for privileged architecture library code.) For example, EVAX_ADDQ, with the appropriate operands, performs the quadword add instruction. (For more information on built-ins, see Appendix C.)

The compiler also provides 64-bit addressing support, which is documented in Chapter 5 and in Appendix E. Support for 64-bit addressing was introduced in OpenVMS Alpha Version 7.0. This support is provided for those rare instances when it is preferable to use VAX MACRO to access 64-bit address space instead of using a high-level language that is supported on OpenVMS Alpha.

1.2 Differences Between the Compiler and the Assembler

It is important to remember that the MACRO-32 compiler is a compiler, not an assembler. It does *not* create output code that exactly matches the input code. In its optimization process, the compiler may move, replicate, or remove code and interleave instructions. Furthermore, the faulting behavior of the ported code may not match that of VAX code. These differences are described in the following sections.

1.2.1 Moving Code

Mispredicted branches are expensive on an Alpha system. The compiler attempts to determine the most likely code path through the module and then generates code that consolidates that code path. Code paths deemed unlikely are moved out of line to the end of the module. Consider the following example:

```
        $ASSIGN_S       DEVNAM=DEVICE,CHAN=CHANNEL
        BLBS    R0,10$
        JSB     PROCESS ERROR
        HALT
10$:
```

In this example, the compiler will treat the HALT as an unlikely code path and detect that the two code streams do not rejoin at 10$. Because of these conditions, it will determine that the branch is likely to be taken. It will then move the intervening instructions out of line to the end of the module, change the BLBS instruction to a BLBC that branches to the moved code, and continue with in-line code generation at the label 10$, as follows:

```
        $ASSIGN_S       DEVNAM=DEVICE,CHAN=CHANNEL
        BLBC    L1$
10$:      .
          .
          .
        (routine exit)
L1$:    JSB     PROCESS ERROR
        HALT
```

You can change the compiler's determination of the likelihood of conditional branches with the compiler directives .BRANCH_LIKELY and .BRANCH_UNLIKELY (see Section 4.2).

1.2.2 Replicating Code

The compiler may replicate small sections of code multiple times to eliminate excessive branching. For example, when compiling branches to the following VAX code, the compiler will replicate the MOVL at each branch to ERROR1 and then branch directly to COMMON_ERROR.

```
ERROR1: MOVL    #ERROR1,R0
        BRW     COMMON_ERROR
```

1.2.3 Removing Code

The compiler's optimizations may determine that some instructions do not contribute to the code flow. In such instances, the instructions may be removed. An example of this is a CMP or TST instruction with no subsequent conditional branch, such as the following:

```
        CMPB    (R2),511(R2)
        JSB     EXE$SENDMSG
```

Removal of this CMPB instruction could cause a problem if its purpose was to touch two memory locations to ensure that the memory pages were faulted in before calling the routine. This would likely have to be changed in porting to OpenVMS Alpha anyway because of the different page sizes of VAX and Alpha systems. In addition to changing the page size, you should replace the instruction with MOV*x* instructions, such as the following:

```
        MOVB    (R2),R1
        MOVB    8191(R2),R0
        JSB     EXE$SENDMSG
```

Note that the two MOVB instructions operated on two different registers. The compiler does not currently remove instructions that load values into a register which is never subsequently read before being overwritten. However, this optimization may be done in the future.

Note

In general, code which requires that a memory read reference actually touch memory should be examined carefully, as current or future optimizations may move or remove the references.

1.2.4 Interleaving Instructions

Instruction scheduling, which is performed by default (see Section 4.3), will interleave the Alpha instructions generated from one VAX instruction with the Alpha instructions generated by surrounding VAX instructions.

1.2.5 Reserved Operand Faults

On VAX systems, some VAX MACRO instructions may generate a reserved operand fault if certain operands are out of a required range. For example, on a bit manipulation instruction such as INSV, if the size operand is greater than 32, a VAX system will generate a run-time reserved operand fault.

On Alpha systems, if the operand that is out of range is a compile-time constant, the compiler will flag this condition with an error message. However, if this operand is variable at run time, the compiler makes no attempt to generate run-time range checks on it. If the operand is out of range, the resulting operation may cause incorrect results yet not create a fault.

1.3 Step-by-Step Porting Process

The following steps have proven to be an efficient means for porting VAX MACRO code to OpenVMS Alpha:

1. Inspect each module you intend to port, from beginning to end, for coding practices that prohibit its successful porting. Such scrutiny is necessary, because it is impossible for the compiler to account for the myriad imaginative uses of VAX MACRO code that take advantage of a comprehensive knowledge of the VAX architecture. Such uses, if not detected and modified, can undermine an effort to port VAX MACRO code to OpenVMS Alpha.
2. At each entry point in the module, add the appropriate entry-point directive (.CALL_ENTRY, .JSB_ENTRY, .JSB32_ENTRY, or .EXCEPTION_ENTRY). You do not need to change the VAX MACRO entry point .ENTRY to .CALL_ENTRY unless you want to use .CALL_ENTRY clauses. Nor do you need to add the register arguments at this time. (Guidelines for the correct placement of these directives appear in Section 2.2; a full syntactical description of each appears in Appendix B.)
3. Invoke the compiler to compile the module. A suggested command procedure for doing this appears in Section 2.11.

   By default, the compiler flags unaligned stack and memory references, routine branches, potentially problematic instructions, and self-modifying code. If you specify /FLAG=HINTS on the command line, the compiler will provide suggestions for the **input** and **output** register arguments of the entry point directives you inserted at Step 2.
4. Take note of the problems the compiler reports.

   For assistance in interpreting compiler messages, you can invoke the Help Message utility for an explanation and user action for each message you received. For more information about the Help Message utility, refer to DCL help or the *OpenVMS System Messages: Companion Guide for Help Message Users*. Also, Section 1.4 and Chapter 3 provide specific details about VAX MACRO coding practices that cannot be directly translated to Alpha code.

   Remember that resolution of the problems detected on this pass may allow the compiler to discover additional problems on a subsequent pass.
5. Edit the VAX MACRO source. Fix the problems indicated by the compiler and look for others the compiler may have missed.

   However, do *not* change code just to avoid compiler *informational* diagnostic messages. Most of the information-level messages are there to point out code which will result in less optimal performance on an Alpha processor but which will compile correctly. If you have examined the offending instructions in the source code and are convinced that all is well, leave the code alone. Remember that you can use the command line qualifiers /FLAG and /WARN to control diagnostic message generation. Also, the .ENABLE and .DISABLE directives can turn off information level messages for segments of code within a module.
6. Add **input**, **output**, **preserve**, and **scratch** arguments—as appropriate—to the entry point directives you provided in Step 2 and supply a list of pertinent registers for each specified argument. Section 2.5 can help you determine which registers to list.

7. Repeat Steps 3 through 6 until the compiler generates informational messages only for VAX MACRO source code that you know—and have verified—produces correct OpenVMS Alpha object code.
8. If your module is common to both VAX and Alpha systems (a coding convention discussed in Section 1.6), your porting effort is not complete until the module is acceptable to the VAX MACRO assembler as well as the compiler.

Once you have some experience in porting VAX MACRO modules, it will be easier to recognize certain problems while inspecting the source and to fix them before your initial invocation of the compiler.

1.4 Identifying Nonportable VAX MACRO Coding Practices

When examining a VAX MACRO module that you intend to compile to OpenVMS Alpha object code, look for any of the following coding constructs. The occurrence of these in a module can make porting take longer than it would otherwise. Although the compiler can identify many of these practices, recognizing them yourself will speed up the porting effort. For more information about nonportable MACRO coding practices, including some that occur less frequently and are less easy to detect, see Chapter 3.

1. Removing the return address from the stack to return to the caller's caller, such as the following (see Section 3.3.4):

   ```
   TSTL    (SP)+               ; remove return address
   RSB                         ; return to caller's caller
   ```

2. Temporarily removing the return address from the stack to allocate space on the stack using mechanisms such as the following (see Section 3.3.4):

   ```
   POPL    R0                  ; remove the return address
   SUBL    #structure_size,SP  ; allocate stack space for caller
   PUSHL   R0                  ; replace the return address
   ```

   or

   ```
   POPL    R1                  ; hold return address
   PUSHL   structure           ; build structure for caller
   ;
   ; code continues
   ;
   JMP     (R1)                ; return to caller
   ```

3. Pushing a label onto the stack, as in the following examples, often as an attempt to construct a return address for an RSB instruction (see Section 3.3.3):

   ```
   MOVAL   routine_label,-(SP)
   RSB
   ```

   or

   ```
   PUSHAL  routine_label
   RSB
   ```

4. Modifying the frame pointer (FP) (see Section 3.1.1). VAX MACRO code typically modifies the frame pointer for one of two reasons:

   - To manually assemble a call frame on the stack.
   - To use the frame pointer to reference local storage allocated in a .JSB_ ENTRY routine.

   ```
   MOVL    SP,FP
   SUBL    #data_area,SP
   ```

5. Constructing an REI target, as in the following examples (see Section 3.3.7):

   ```
   MOVL    #fake_psl,-(SP)
   MOVAL   target_label,-(SP)      ; all three
   REI
   ```

   or

   ```
   MOVPSL  -(SP)
   MOVAL   target_label,-(SP)      ; force AST delivery only
   REI
   ```

   or

   ```
           MOVL    #fake_psl,-(SP)
           BSBW    DOREI
           ;
           ; code continues
           ;
   DOREI:  REI
   ```

6. Branching to a destination that consists of a label plus an offset as in the following example. The appearance of this practice in VAX MACRO code may indicate a branch past some data in the code stream, such as the register save mask at the top of a .CALL_ENTRY routine (see Section 3.2.1). Alternatively, it may be a sign that the code is familiar with and dependent upon the size of VAX instructions (see Section 3.2.3).

   ```
   BRx     label+offset
   ```

7. Moving an opcode to a location, usually the stack or a data area, as shown in the following example. This practice indicates either generated or self-modifying code and will require redesign as indicated in Section 3.2.2.

   ```
   MOVB    #OP$_opcode,(Rx)
   ```

   or

   ```
   MOVZBL  #OP$_opcode,(Rx)
   ```

8. Jumping across modules. Because of architectural requirements, the compiler must handle jumps across modules as JSBs. Therefore, external branch targets as in the following example must be declared with the .JSB_ENTRY directive (see Section 2.2.3).

   ```
   JMP     G^external_routine
   ```

1.5 Establishing Useful Coding Conventions

Section 1.3 describes a recommended process for porting VAX MACRO code to OpenVMS Alpha. Although this process may provide a mechanism for porting code efficiently, it cannot by itself guarantee that the porting effort will be consistent among the engineers who are performing the porting, or will be intelligible to the engineers who will later need to debug and test ported code. To ensure that the porting effort proceeds uniformly and that its effects on source code are well documented, an engineering group should establish coding conventions that are relevant to the goals of the effort.

Naturally, any methodology an engineering group may adopt should be shaped by that group's development environment, including those procedures the group follows for tool management, source code control, common code, and testing. The coding conventions an engineering group should evaluate include:

- Establishing VAX MACRO source modules that are common to VAX and Alpha systems, and conditionalizing architecture specific code (see Section 1.6)
- Providing clear and consistent register declarations in the compiler's entry-point directives (see Section 2.2)

1.6 Maintaining Common Sources for VAX and Alpha Systems

When designing a VAX MACRO code porting effort, consider the benefits of maintaining common sources for VAX and Alpha systems. It is advantageous to an engineering group to have only one copy of its sources to maintain and enhance, and common sources help ensure common user interfaces. However, if you find that you are conditionalizing so much source code that it is no longer intelligible, take steps to restructure the code into architecture-specific and architecture-independent pieces. If the number of these pieces becomes unmanageable, create separate architecture-specific modules.

1.6.1 Including Compiler Directive Definitions

A successful compilation does not preclude VAX MACRO code in a source file from also processing successfully under the VAX MACRO assembler. If you added any compiler directives to your code, they will be resolved by the library SYS$LIBRARY:STARLET.MLB when the code is assembled. The assembler automatically searches this library for any undefined macros. After finding these definitions, the assembler will ignore occurrences of the compiler directives.

However, if you are using OpenVMS VAX Version 6.0 or earlier, you must first extract the directive definitions from ALPHA$LIBRARY:STARLET.MLB on Alpha and insert them into your SYS$LIBRARY:STARLET.MLB on OpenVMS VAX. For example:

```
LIB/EXTRACT=.JSB_ENTRY/OUT=JSB_ENTRY.MAR ALPHA$LIBRARY:STARLET.MLB
.
.
.
LIB/INSERT SYS$LIBRARY:STARLET.MLB JSB_ENTRY.MAR
```

Note that many of the definitions of the compiler directives refer to other macros. Make sure to extract not only the definitions of all the compiler directives used in your code but also all the associated macros and insert them into SYS$LIBRARY:STARLET.MLB on your OpenVMS VAX system.

1.6.2 Removing VAX Dependencies

If you must make changes to source files because they contain certain coding practices that cannot be directly compiled into Alpha code, you may still be able to generate images for both VAX and Alpha systems from common VAX MACRO sources.

Removing such VAX dependencies so that the same code can run on both VAX and Alpha systems can yield great benefits during the porting process. You can debug single modules or groups of modules of the ported code by building and testing the modules in the VAX environment. This can greatly speed your debugging process.

In some cases, you must define and implement procedures for conditionalizing the source files (as described in Section 1.6.3) or for providing separate sources for VAX and Alpha systems.

If the code runs in an inner mode, it is unlikely that an effort to generate VAX and Alpha images from common VAX MACRO sources will be fully successful. Because inner-mode code interoperates with the executive, it is vulnerable to the differences between VAX and Alpha system interfaces and executive data structures. However, user-mode code is generally immune from architectural dependencies and may more easily serve as the basis for common code.

1.6.3 Using Architecture-Specific Symbols

Conditionalizing VAX MACRO code requires the use of two ARCH_DEFS.MAR files with architecture-specific symbols, one to be assembled with the VAX MACRO source code on a VAX processor, the other to be compiled with the VAX MACRO source code on an Alpha processor.

If you choose to make code in a common source module conditional on architecture type, include ARCH_DEFS.MAR in your assembly and compilation and use .IF DF,VAX or .IF DF,ALPHA.

An ARCH_DEFS.MAR file is provided with OpenVMS Alpha. You need to create a corresponding file for OpenVMS VAX. The following is an example of such a file:

```
; This is the VAX version of ARCH_DEFS.MAR, which contains
; architectural definitions for compiling sources for
; VAX systems.
;
VAX = 1
VAXPAGE = 1
ADDRESSBITS = 32
```

The Alpha version exists in SYS$LIBRARY and contains definitions for the following symbols:

- ALPHA—to indicate the source code is Alpha architecture specific
- BIGPAGE—to indicate the source code assumes variable memory page size

Warning

Any other symbols in ARCH_DEFS.MAR on OpenVMS Alpha are specific to OpenVMS Alpha source code and are not guaranteed to be included from release to release.

1.7 Using the MACRO-32 Compiler

Compaq recommends that you use the current version of the MACRO-32 compiler. Make sure that the version of SYS$LIBRARY:STARLET.MLB that ships with the compiler version you are using is installed on your system and that the logical name points to the correct directory.

For compiling code to run on AlphaServer systems with the Alpha processor 21264 (or later), you must use the following version of the MACRO-32 compiler:

- Version 3.1 or later for OpenVMS Alpha Version 7.1-2
- Version 4.1 or later for OpenVMS Alpha Version 7.2
- Shipping version of the compiler for later versions of OpenVMS Alpha

Directions for invoking the compiler are provided in Appendix A.

2 How to Use the MACRO-32 Compiler

The MACRO-32 Compiler for OpenVMS Alpha has been designed to help you port VAX MACRO source code from OpenVMS VAX to OpenVMS Alpha. When operating on VAX MACRO code that complies with the restrictions and guidelines described in this manual, the compiler produces a valid OpenVMS Alpha object module that preserves the semantics of the original VAX MACRO source and adheres to the OpenVMS calling standard.

This chapter describes the following topics:

- Using Alpha registers (Section 2.1)
- Routine calls and declarations (Section 2.2)
- Declaring CALL entry points (Section 2.3)
- Declaring JSB routine entry points (Section 2.4)
- Declaring a routine's register use (Section 2.5)
- Branching between local routines (Section 2.6)
- Declaring exception entry points (Section 2.7)
- Using packed decimal instructions (Section 2.8)
- Using floating-point instructions (Section 2.9)
- Preserving VAX granularity and atomicity behavior (Section 2.10)
- Compiling and linking (Section 2.11)
- Debugging (Section 2.12)

As discussed in Chapter 1, the compiler *cannot* transparently convert all VAX MACRO code. There are many coding practices that cannot be directly compiled into OpenVMS Alpha code. They must be removed or modified to guarantee a successful compilation. The compiler can detect and report many of these coding practices.

2.1 Using Alpha Registers

Alpha computers employ 32 integer registers, R0 through R31. Because code generated by the compiler uses Alpha registers R0 through R12 as if they were VAX registers, VAX MACRO code usage of these registers (for instance, as input to JSB routines) does not have to change to achieve a correct compilation. VAX MACRO instructions (such as MOVL and ADDL) use the lower 32 bits of the Alpha register involved in the operation. The compiler maintains a sign-extended 64-bit form of this value in the register.

Registers R13 and above are also available to VAX MACRO code that will be compiled to OpenVMS Alpha object format. If you decide to use these registers, review the following constraints:

- Generally, existing VAX MACRO code uses only registers R0 through R11, R12 through R14 being specially defined as the argument pointer (AP), frame pointer (FP), and stack pointer (SP), respectively. The compiler will compile legal uses of AP, FP, and SP as references to the Alpha registers that have functions similar to the VAX registers. If a VAX MACRO source is referencing AP as a scratch register, the compiler converts this reference to a reference to R12. If this is not desirable, you should change the code to use a different scratch register.
- If the compiler detects a reference to R12, R13, and R14 in a VAX MACRO source, it will *not* convert it to a reference to the Alpha equivalent of AP, FP, or SP. Rather, it will consider the reference to be to the corresponding Alpha integer register.
- Code that uses R13 and above cannot be assembled by the VAX MACRO assembler. Therefore, it should be conditionalized for OpenVMS Alpha or appear in a module that is specific to OpenVMS Alpha.
- The compiler allows you to access the full 64 bits of Alpha registers by using the EVAX_LDQ and EVAX_STQ built-ins, as described in Appendix C.
- You should take special care when referencing registers with specific architected usage, as defined in the *OpenVMS Calling Standard*.
- Registers 16 and above are scratch registers, as defined in the *OpenVMS Calling Standard*. You cannot expect their values to survive across routine calls.
- The compiler may use registers R13 and above as temporary registers if they are not used in the source code for a routine. R13 through R15 are saved and restored if they are used.

2.2 Routine Calls and Declarations

The *OpenVMS Calling Standard* specifies very different calling conventions for VAX and Alpha systems.

On a VAX system, there are two different call formats, CALL and JSB, with five different instructions: CALLS, CALLG, JSB, BSBW, and BSBB. CALL instructions create a frame on the stack which contains return information, saved registers, and other routine information. Parameters to CALL instructions are passed in consecutive longword memory locations, either on the stack (CALLS) or elsewhere (CALLG). JSB instructions have the return address stored on the stack. For both call formats, the hardware processing of the calling instruction provides all of these functions.

On an Alpha system, there is only one call format and only one subroutine call instruction, JSR. For routines that expect parameters, the first six parameters are passed in R16 through R21, with the parameter count in R25 and subsequent parameters in quadwords on the stack. The hardware execution of the JSR instruction simply passes control to the subroutine and places the return address in a designated register. It neither allocates stack space, nor creates a call frame, nor provides any parameter manipulations. All of these functions must be done by the calling routine or the called routine.

The compiler must generate code that conforms to the *OpenVMS Calling Standard* yet emulates the function of the different VAX MACRO instructions. This requires special code both at the calling instruction to manipulate the input parameters and at the entry point of the routine itself to create a stack frame and other context.

The compiler generates code only for source instructions that are part of a declared routine. For the compiler to generate the proper linkage information and proper routine code, you must insert a compiler directive at the entry points in the VAX MACRO source.

2.2.1 Linkage Section

On Alpha systems, all external (out-of-module) references are made through a *linkage section*. The linkage section is a program section (psect) containing:

- Addresses of external variables
- Constants too large to fit directly in the code stream
- Linkage pairs
- Procedure descriptors

A linkage pair is a data structure used when making a call to an external module. The linkage pair contains the address of the callee's procedure descriptor, and the entry point address. The linkage pair resides in the caller's linkage section; therefore, there may be several linkage pairs for a particular routine, one in each caller's linkage section. Linkage pairs are not used for module-local calls, since the compiler can refer to the callee's frame descriptor directly.

A procedure descriptor is a data structure that provides basic information about a routine, such as register and stack usage. Each routine has one unique procedure descriptor, and it is located in its linkage section. The procedure descriptor is normally not accessed at run time. Its primary use is for interpreting the frame in case of exceptions, by the debugger, for example.

For more information about these and other Alpha data structures, see the *OpenVMS Calling Standard*.

2.2.2 Prologue and Epilogue Code

To mimic the behavior of VAX subroutines, the compiler must generate code at the entry and exit of each routine. This code emulates the functions that are done by the VAX hardware, and is called the prologue and epilogue code.

In the prologue code, the compiler must allocate stack space to save any necessary context. This includes saving any registers that will be preserved by the routine and saving the return address. If this is a CALL routine, it also includes establishing a new stack frame and changing the frame pointer (FP), and may include further manipulation and storage of the input parameters (see Section 2.3).

In the epilogue code, the compiler must restore any necessary registers, stack frame information, and the return address, and trim the stack back to its original position.

2.2.3 When to Declare Entry Points

Any code label that is a possible target of a CALLS, CALLG, JSB, BSBW, or BSBB instruction must be declared as an entry point. In addition, any code label must be declared as an entry point using a .JSB_ENTRY or .JSB32_ENTRY directive if:

- The label can be the target of a global (cross-module) JMP, BRB, or BRW instruction.
- The label can be the target of an indeterminate branch (such as BRB @(R10)), where the address of the label has been stored in R10, even if the reference and the label are within the same module.
- The address of the label is stored in a register or memory location, even if it is never accessed by the current module.

The OpenVMS calling standard for Alpha computers does not provide a way to access indeterminate code addresses directly. All such accesses are accomplished using a procedure descriptor to describe the routine and the code address. When a code label address is stored, the compiler does not know if that address will be referenced only by the current module, or whether it may be accessed by another MACRO module or another module written in another language. Whenever a source instruction stores a code address, the MACRO-32 compiler instead stores the procedure descriptor address for that code address so that other code can access it correctly. For a procedure descriptor to exist, the label must be declared as an entry point.

Likewise, when a stored address is used as a branch destination, the compiler does not know where that address came from, so it always assumes that the stored address is the address of a procedure descriptor and uses that descriptor to pass control to the routine.

2.2.4 Directives for Designating Routine Entry Points

Macros in STARLET.MLB generate directives for designating the entry points to routines. (Appendix B describes the format of each of these macros.) When assembled for VAX systems, the macros are null, except for .CALL_ENTRY; when compiled for Alpha systems, the macros expand to verify the arguments and generate the compiler directives. Therefore, you can use the following macros in common source modules:

- The .CALL_ENTRY directive logically replaces the .ENTRY directive in MACRO-32 code. However, you do not need to replace the .ENTRY directive unless you want to use one of the .CALL_ENTRY clauses. If you replace the .ENTRY directives, note that you cannot do a massive replace with an editor because the arguments do not match.

Note

Since the .ENTRY directive is converted to an Alpha .CALL_ENTRY directive, any registers modified in the routine declared by .ENTRY will be automatically preserved. This is different from VAX MACRO behavior. See Section 2.3.2.

- The .JSB_ENTRY directive indicates a routine that is executed as a result of a JSB, BSBB, or BSBW instruction. The .JSB32_ENTRY directive is used for specialized environments that do not require the preservation of the upper 32 bits of Alpha register values.

  The .JSB_ENTRY directive should also be used to declare entry points which are the targets of JMP or branch instructions from other modules, since such cross-module branches are implemented as JSBs by the compiler.

  Branches to stored code addresses, such as JMP (R0), are also treated as JSB instructions. Hence, potential targets must be declared with .JSB_ENTRY. The compiler warns you of this when you attempt to store a code label or when such a branch is used.
- The .EXCEPTION_ENTRY directive designates exception service routines.

2.2.5 Code Generation for Routine Calls

The Alpha code generated for VAX JSB, BSBW, and BSBB instructions is very simple because no parameter manipulation must be done. However, the code for CALLS and CALLG instructions is more complex because the compiler must translate the parameter handling of the VAX calling standard to the Alpha calling standard. When processing a CALLS instruction with a fixed argument count, the compiler automatically detects any pushes onto the stack that are actually passing parameters to a routine, and generates code that moves the parameters directly to the registers used by the Alpha calling standard.

Consider the following VAX call:

```
PUSHL   R2
PUSHL   #1
CALLS   #2,XYZ
```

This VAX call is compiled as follows:

```
LDQ     R26, 32(R13)          ; Get routine address
LDQ     R27, 40(R13)          ; Get routine linkage pointer
SEXTL   R2, R17               ; R2 is second parameter
BIS     R31, 1, R16           ; #1 is first parameter
BIS     R31, 2, R25           ; 2 parameters
JSR     R26, R26              ; call the routine
```

For a CALLS instruction with a variable argument count or a CALLG instruction, the compiler must call an emulation routine which unpacks the argument list into the Alpha format. This has two side effects. On VAX systems, if part of the argument list pointed to by a CALLG instruction is inaccessible, but the called routine does not access that portion of the argument list, the routine will complete successfully without incurring an access violation. On Alpha systems, since the entire argument list is processed before calling the routine, the CALLG will always incur an access violation if any portion of the argument list is inaccessible. Also, if either the argument count of a CALLG instruction or a CALLS instruction with a variable argument count exceeds 255, the emulated call returns with an error status without ever invoking the target routine.

2.3 Declaring CALL Entry Points

Because of the differences between the VAX and the Alpha calling standards (see Section 2.2), the compiler must translate all VAX parameter references to AP in the following ways:

- It converts parameter references off AP in the called routine into direct register and stack references to the new Alpha parameter locations.
- It detects references to AP that may result in aliased references to the parameter list, or that use variable offsets which cannot be resolved at compile time. The compiler mimics VAX argument lists by packing the quadword register and stack arguments into a longword argument list on the stack. Two arguments to the .CALL_ENTRY directive have been created for this purpose (see Section 2.3.1).

In the latter case, the practice is known as *homing* the argument list. The resulting *homed argument list* is in the fixed position of the stack frame, so all references are FP-based.

2.3.1 Homed Argument Lists

Examples of AP references for which the compiler automatically homes the arguments include:

- Storing AP or an address based on AP in another register.
- Passing AP or an address based on AP as a parameter.
- Adding a variable offset to AP to reference a parameter.
- Using variable indexing off AP to reference a parameter.
- Using a non-longword aligned offset into the arglist, such as 6(AP).

The compiler sets up the homed argument list in the fixed temporary region of the procedure frame on the stack.

Although not mandatory, you can specify homing with two arguments to the .CALL_ENTRY directive, **home_arg=TRUE** and **max_args=*n***.

The argument **max_args=*n*** represents the maximum number of longwords the compiler should allocate in the fixed temporary region. The main reason for using **max_args=*n*** is if your program receives more arguments than the number explicitly used in the source code. A common reason for specifying **home_arg=true** in a routine, which does not in itself require it, is the invocation of a JSB routine that references AP. For such references, the compiler assumes that the argument list was homed in the last .CALL_ENTRY routine, and uses FP-based references. Because this may not always be a valid assumption, the compiler reports an informational message when AP is used inside .JSB_ENTRY routines.

If you want to suppress the informational messages that the compiler reports, use both arguments.

If you omit one or both arguments and the compiler detects a use that requires homing, messages are issued by the compiler, informing you of what the compiler did. You can then make the appropriate changes to your code, such as adding one or both arguments or changing the value of the **max_args** argument.

If you only specify **home_args=TRUE**, the following message is issued:

> AMAC-W-MAXARGEXC, MAX_ARGS exceeded in routine *routine_name*, using *n*

This message is issued because you explicitly asked for homing but did not specify the number of arguments with **max_args**. If the compiler does not find any explicit argument reference, then it allocates six longwords. If the compiler finds explicit argument references, it allocates the same number of longwords as the highest argument reference found. (This message is also issued if you specify a value for **max_args** which is less than the number of arguments found by the compiler to be homed.)

If you specify only **max_args**, and if the compiler detects a reference requiring homing, the compiler issues the following message:

> AMAC-I-ARGLISHOME, argument list must be homed in caller

If you specify neither argument and the compiler detects a reference requiring homing, the compiler issues the following message:

> AMAC-I-ARGLISHOME, argument list must be homed in caller
> AMAC-I-MAXARGUSE, max_args value used for homed arglist is *n*

where *n* represents the highest argument referenced, as detected by the compiler.

2.3.2 Saving Modified Registers

A well-behaved VAX MACRO CALL routine passes all parameters via the argument list and saves all necessary registers via the register mask in the .ENTRY declaration. However, some routines do not adhere to these standards and may pass parameters via registers or save and restore the contents of the necessary registers within the routine with instructions such as PUSHL and POPL.

Using PUSHL and POPL to save and restore registers on an Alpha system is insufficient because these instructions save only the low 32 bits of the register. To avoid register corruption, the compiler automatically saves and restores the full 64-bit values of any register modified within the routine (except R0 and R1), in addition to the registers specified in the register save mask. This means that any registers that were intended to pass an output value out of the called routine will no longer do so. You must either pass the output via the standard argument list or declare the register to be output with the **output** parameter in the .CALL_ENTRY declaration (see Section 2.5).

2.3.3 Modifying the Argument Pointer

If a routine modifies AP, the compiler changes all uses of AP in that routine to R12 and reports all such modifications. Although traversing the argument list in this way is not supported, you can obtain similar results by explicitly moving the address of **0(AP)** to **R12** and specifying **home_args=true** in the entry point. **R12** will point to a VAX format argument list.

2.3.4 Establishing Dynamic Condition Handlers in Called Routines

To establish a dynamic condition handler in a called routine, it is necessary to store a condition handler address in the frame. The compiler generates a static condition handler for .CALL_ENTRY routines that modify 0(FP). The static handler invokes the dynamic handler or returns if no condition handler address has been stored in the frame.

For performance reasons, the compiler does not automatically insert a TRAPB instruction at the end of every routine that establishes a condition handler. Because of the indeterminate nature of traps on an Alpha system, this could allow traps from instructions near the very end of the routine to be processed after the frame pointer has been changed in the routine epilogue code, with the result that the declared handler would not process the trap. If it is essential that any traps generated in the routine be processed by the declared handler, insert an EVAX_TRAPB built-in immediately before the RET instruction that ends the routine.

2.4 Declaring JSB Routine Entry Points

In assembled VAX MACRO code and compiled Alpha object code alike, JSB routine parameters are typically passed in registers.

A JSB routine that writes to registers sometimes saves and restores the contents of those registers, unless the new contents are intended to be returned to a caller as output. However, because it is fairly common for a VAX MACRO module to save and restore register contents by issuing instructions such as PUSHL and POPL, saving only the low 32 bits of the register contents, the compiler must add 64-bit register saves/restores at routine's entry and exit points.

The compiler can compute the set of used registers, but without complete knowledge of the routine's callers (which may be in other modules), it cannot determine which registers are intended to contain output values. For this reason, the user must add, at each JSB routine entry point, a .JSB_ENTRY or a .JSB32_ENTRY directive that declares the routine's register usage.

2.4.1 Differences Between .JSB_ENTRY and .JSB32_ENTRY

The compiler provides two different ways of declaring a JSB entry point. The .JSB_ENTRY directive is the standard declaration, and the .JSB32_ENTRY directive is provided for cases when you know that the upper 32 bits of the 64-bit register values are *never* required to be preserved by the routine's callers.

In routines declared with the .JSB_ENTRY directive, the compiler, by default, saves at routine entry and restores at routine exit the full 64-bit contents of any register (except R0 and R1) that is modified by the routine. If a register is not explicitly modified by the routine, the compiler will not preserve it across the routine call. You can override the compiler's default behavior by means of register arguments that can be specified with the .JSB_ENTRY directive, as described in Section 2.5.

When .JSB32_ENTRY is used, the compiler does not automatically save or restore any registers, thus leaving the current 32-bit operation untouched. The compiler will only save and restore the full 64-bit value of registers you explicitly specified in the PRESERVE argument.

If the routine you are porting is in an environment where you know that no caller *ever* needs to have the upper 32 bits of a register preserved, Compaq recommends that you use the .JSB32_ENTRY directive. It can be much faster for porting code because you do not have to examine the register usage of each routine. You can specify the .JSB32_ENTRY directive with no arguments, and the existing 32-bit register push/pop code will be sufficient to save any necessary register values.

Note

The Alpha compilers for other languages may use 64-bit values. Therefore, any routine which is called from another language — or which is called from another MACRO-32 routine that is in turn called from another language — cannot use .JSB32_ENTRY.

2.4.2 Two General Cases for Using .JSB32_ENTRY

There are two general cases where you can use .JSB32_ENTRY:

- If a user-mode application or self-contained subsystem is written entirely in MACRO-32 you can use .JSB32_ENTRY throughout the application.
- If you have one major MACRO-32 routine that is called from another language or from any source which requires 64-bit register preservation, and that routine calls several other MACRO-32 routines, you can set a barrier of 64-bit preservation. You can do this by using .JSB_ENTRY or .CALL_ENTRY on the major routine and preserving *all* registers that are not explicit outputs. The internal subroutines can then use the .JSB32_ENTRY directive.

  Note that it is not sufficient to just use .JSB_ENTRY or .CALL_ENTRY for the barrier without explicitly preserving all registers, since by default the compiler will only save and restore the registers that are explicitly modified in the major routine. The internal subroutines that use .JSB32_ENTRY may then modify registers that are not being preserved. You must also make sure that none of the internal subroutines can be called in any way that bypasses the barrier.

Warning

The .JSB32_ENTRY directive can be a great time-saver *if* you are sure that you can use it. If you use .JSB32_ENTRY in a situation where the upper 32 bits of a register *are* being used, it may cause very obscure and difficult-to-track bugs by corrupting a 64-bit value that may be several calling levels above the offending routine.

.JSB32_ENTRY should *never* be used in an AST routine, condition handler, or any other code that can be executed asynchronously.

2.4.3 PUSHR and POPR Instructions Within JSB Routines

There will be cases when you add a .JSB_ENTRY directive to a routine which already saves/restores some registers via PUSHR/POPR. Depending on routine usage, some registers may end up being saved/restored twice, once by the compiler and again by the PUSHR/POPR. Do not attempt to optimize this unless the code is extremely performance sensitive. The compiler attempts to detect this and eliminate the redundant save/restores.

2.4.4 Establishing Dynamic Condition Handlers in JSB Routines

The compiler will flag, as illegal, any code in a .JSB_ENTRY routine that attempts to modify 0(FP).

2.5 Declaring a Routine's Register Use

The compiler provides four register declaration arguments: **input**, **output**, **scratch**, and **preserve**, that you can specify in a .CALL_ENTRY, .JSB_ENTRY, or .JSB32_ENTRY entry-point directive. These register arguments are used to describe the usage of registers in the routine and to instruct the compiler how to compile the routine. You can use the register arguments to:

- Override the compiler's default preservation behavior (see Section 2.3.2 and Section 2.4.1).
- Indicate the nonavailability of registers for temporary compiler usage.
- Document routine register usage.

Note

If you specify /OPTIMIZE=VAXREGS to use VAX registers as temporary registers, you must declare all implicit register uses with the **input** and **output** clauses to prevent their use as temporary registers. When this optimization is enabled, the compiler can use as temporary registers any registers that are not explicitly declared.

2.5.1 Input Argument for Entry Point Register Declaration

The **input** argument indicates those registers from which the routine receives input values. In some cases, the routine itself does not use the contents of the register as an input value, but rather calls a routine that does. In the latter case, known as the **pass-through input** technique, the other routine should also declare the register as **input** in its routine entry mask.

The **input** argument has no effect on the compiler's default register preservation behavior. Registers specified only in the **input** argument will still be treated by the compiler exactly as described in Section 2.3.2 and Section 2.4.1. If a register is used as an input and you want to change the default preservation behavior, you should specify the register in the **output**, **preserve**, or **scratch** arguments in addition to the **input** argument.

The **input** argument informs the compiler that the registers specified have meaningful values at routine entry and are unavailable for use as temporary registers even before the first compiler-detected use of the registers. Since the compiler does not normally use any of the VAX registers (R2 through R12) as temporary registers, specifying registers in the **input** argument affects compiler temporary register usage in two cases:

- If you are using the VAXREGS optimization switch. This optimization allows the compiler to use as temporary registers any of the VAX registers which are not explicitly being used by the VAX MACRO code. Note that for .JSB32_ENTRY directives, the compiler always assumes that all the VAX registers are used as input when using the VAXREGS optimization, so it is not necessary to specify VAX registers in the **input** argument to prevent their use as compiler temporary registers.

- If you are explicitly using any of the Alpha registers (R13 and above).

In either of these cases, if you do not specify a register that is being used as input in the **input** argument, the compiler may use the register as a temporary register, corrupting the input value.

If you are not using the VAXREGS optimization switch or any of the Alpha registers, the input mask is used only to document your routine.

2.5.2 Output Argument for Entry Point Register Declaration

The **output** argument indicates those registers to which the routine assigns values that are returned to the routine's caller. In many cases, the routine itself modifies the register; in others, the routine calls another routine that deposits the output value. In the latter case, known as the **pass-through output** technique, the other routine must also declare the register as **output** in its routine entry register set.

The use of this argument prevents the automatic preservation of registers that are modified during the routine. Any register included in this argument will not be preserved by a .CALL_ENTRY or .JSB_ENTRY routine, even if it is modified by the routine.

The **output** argument informs the compiler that the registers specified have meaningful values at routine exit and are unavailable for use as temporary registers even after the last compiler-detected use of the registers. Since the compiler does not normally use any of the VAX registers (R2 through R12) as temporary registers, specifying registers in the **output** argument only affects compiler temporary register usage in two cases:

- If you are using the VAXREGS optimization switch. This optimization allows the compiler to use as temporary registers any of the VAX registers which are not explicitly being used by the VAX MACRO code. Note that for .JSB32_ENTRY directives, the compiler always assumes that all the VAX registers are used as output when using the VAXREGS optimization, so it is not necessary to specify VAX registers in the **output** argument to prevent their use as compiler temporary registers.
- If you are explicitly using any of the Alpha registers (R13 and above).

In either of these cases, if you do not specify a register that is being used as output in the **output** argument, the compiler may use the register as a temporary register, corrupting the output value.

For .JSB32_ENTRY routines, since no registers are preserved by default and all registers are assumed to be outputs by the VAXREGS optimization, the **output** argument is used only to document your code.

2.5.3 Scratch Argument for Entry Point Register Declaration

The **scratch** argument indicates those registers that are used within the routine but should not be saved and restored at routine entry and exit. The caller of the routine does not expect to receive output values in these registers nor does it expect the registers to be preserved.

The use of this argument prevents the automatic preservation of registers that are modified during the routine. Any register included in this argument will not be preserved, even if it is modified by the routine.

The **scratch** argument also pertains to the compiler's temporary register usage. The compiler may use registers R13 and above as temporary registers if they are unused in the routine source code. Since R13 through R15 must be preserved, if modified, according to the OpenVMS Alpha calling standard, the compiler preserves those registers if it uses them.

However, if they appear in the **scratch** register set declaration, the compiler will *not* preserve them if it uses them as temporary registers. As a result, these registers may be scratched at routine exit, even if they were not used in the routine source but are in the **scratch** set. If the VAXREGS optimization is used, this applies to registers R2 through R12, as well.

For .JSB32_ENTRY routines, since R2 through R12 are not preserved by default, their inclusion in the **scratch** declaration is for documentation purposes only.

2.5.4 Preserve Argument for Entry Point Register Declaration

The **preserve** argument indicates those registers that should be preserved over the routine call. This should include only those registers that are modified and whose full 64-bit contents should be saved and restored.

The **preserve** argument causes registers to be preserved whether or not they would have been preserved automatically by the compiler's processing of a .CALL_ENTRY or .JSB_ENTRY directive. This is also the only way in a .JSB32_ ENTRY routine to save and restore the full 64 bits of a register. Note that because R0 and R1 are scratch registers, by calling standard definition, the compiler never saves and restores them in any routine unless you specify them in the **preserve** argument at the routine's entry point.

This argument overrides the **output** and **scratch** arguments. If you specify a register both in the **preserve** argument and in the **output** or **scratch** arguments, the compiler will preserve the register but will report the following warning:

```
%AMAC-W-REGDECCON, register declaration conflict in routine A
```

The **preserve** argument has no effect on the compiler's temporary register usage.

2.5.5 Help for Specifying Register Sets

When you invoke the compiler, specifying /FLAG=HINTS on the command line, the compiler generates messages that can assist you in constructing the register sets for routine entry points. Among the hints the compiler provides are the following:

- Registers that might be used as input to the routine, which may not have been your intention. If a register is read before being written, it will be included in this set.
- Registers that might be used for output values. If a register is written but not subsequently read, the compiler will include it in the list of possible output values. Again, this may not have been your intention.
- Registers the compiler saves and restores that are not listed in the entry point's **preserve** argument.

It is recommended that the .CALL_ENTRY, .JSB_ENTRY, and .JSB32_ENTRY register arguments reflect the routine interface, as described in the routine's text header. Wherever possible, you should declare **input**, **output**, **scratch**, and **preserve** register arguments for all routines. You only need to provide the argument when there are registers to be declared (for instance, **input**=<> is not necessary).

2.6 Branching Between Local Routines

The compiler allows a branch from the body of one routine into the body of another routine in the same module and psect. However, because this may result in additional overhead in both routines, the compiler reports an information-level message.

Note

The compiler does not recognize a call to $EXIT as terminating a routine. Add an extra RET or RSB, whichever is applicable, after $EXIT to terminate the routine.

If a CALL routine branches into a code path that executes an RSB, an error message is reported. Such a CALL routine, if not corrected, will fail at run time.

If a JSB routine branches into a code path that executes a RET instruction, and the JSB routine preserves any registers, an informational message is issued. This construct will work at run time, but the registers saved by the JSB routine will not be restored.

If routines that share a code path have different register declarations, the register restores will be done conditionally. That is, the registers written on the stack at routine entry will be the same for both routines, but whether or not the register is restored will depend upon which entry point was invoked.

For example:

```
rout1: .jsb_entry output=r3
           movl    r1, r3          ! R3 is output, not preserved
           movl    r2, r4          ! R4 should be preserved
           blss    lab1
           rsb

rout2:  .jsb_entry                 ! R3 is not output, and
           movl    #4, r3          ! should be auto-preserved
           movl    r0, r4          ! R4 should be preserved
lab1:      clrl    r0
           rsb
```

For both routines, R3 will be included in the registers saved on the stack at entry. However, at exit, a mask (also in the stack frame) will be tested before restoring R3. The mask will not be tested before R4 is restored, because R4 should be restored for both entry points.

Note that declaring registers that are destroyed in two routines that share code as *scratch* in one but not the other is actually more expensive than letting them be saved and restored. In this case, declare them as *scratch* in both, or if one routine requires that they be preserved, as *preserve* in both.

2.7 Declaring Exception Entry Points

The .EXCEPTION_ENTRY directive, as described in Appendix B, indicates the entry point of an exception service routine. Use the .EXCEPTION_ENTRY directive to declare the entry points for routines serving interrupts such as the following:

- Interval clock
- Interprocessor interrupt
- System/processor correctable error
- Power failure
- System/processor machine abort
- Software interrupt

At routine entry, R3 must contain the address of the procedure descriptor. The routine must exit with an REI instruction.

At exception entry points, the interrupt dispatcher pushes onto the stack registers R2 through R7, the PC, and the PSL. To access the contents of these registers, specify the **stack_base** argument in the .EXCEPTION_ENTRY directive. The compiler generates code that places the value of the SP at routine entry in the register you specify in **stack_base**, allowing the exception service routine to use this register to locate the contents of registers on the stack.

The compiler automatically saves and restores all other registers used in the routine, plus, if the service routine issues a CALL or a JSB instruction, all scratch registers, including R0 and R1.

Note

Error handling routines, established when their addresses are stored in the frame at 0(FP), are not .EXCEPTION_ENTRY routines. Such error handlers should be declared as .CALL_ENTRY routines and end with RET instructions.

2.8 Using Packed Decimal Instructions

The packed decimal directive .PACKED and all packed decimal instructions, except EDITPC, are supported for the MACRO-32 compiler by emulation routines that exist outside the compiled module.

2.8.1 Differences Between the VAX and Alpha Implementations

The differences between the implementations on VAX and Alpha systems are noted in the following list:

- Packed decimal instructions and atomicity

  Since all packed decimal instructions are emulated via subroutine calls, they are not atomic or restartable, and cannot be made atomic by the .PRESERVE ATOMICITY directive or /PRESERVE=ATOMICITY switch. Also, there is no guarantee that Alpha registers from R16 through R28 are preserved across the instruction.

- Use of argument registers (R16 through R21)

Arguments are passed to the emulation routines by means of the argument registers (R16 through R21); attempts to use these registers as arguments in a packed decimal instruction will not work correctly.

The compiler converts references to the VAX argument pointer (AP) into references to the Alpha argument registers (see Section 2.3). For example, the compiler converts a parameter reference off the VAX AP, such as 4(AP), to Alpha R16.

To prevent these problems with AP-based references when packed decimal or floating-point instructions are used, it is simplest to specify /HOME_ ARGS=TRUE on the entry point of the routine that contains these references. This forces all AP-based references to be read from the procedure frame, rather than the argument registers, so that no changes need to be made to the instructions.

If /HOME_ARGS=TRUE is not used, the source code that contains parameter references based on the AP register as operands to packed-decimal instructions (or floating-point instructions) should be changed. Copy any AP-based operand to a temporary register first, and use that temporary register in the packed-decimal or floating-point instruction. For example, change the following code:

```
MOVP     R0,  @8(AP), @4(AP)
```

to:

```
MOVL    8(AP), R1
MOVL    4(AP), R2
MOVP    R0,(R1),(R2)
```

- Overflow traps

  Bits 7 and 5 of the VAX Program Status Word (PSW) are the DV (decimal overflow trap enable) and IV (integer overflow trap enable) bits respectively. These bits are not emulated as part of the emulation of the VAX PSW, but the packed decimal support allows you to enable or disable integer and decimal overflow, although you must do so statically at compile time.

  To enable decimal overflow faults, define the symbol PD_DEC_OVF to be nonzero. If PD_DEC_OVF is not defined or is set to zero, the packed decimal emulation routines will not generate decimal overflow faults. To enable integer overflow faults, define the symbol PD_INT_OVF to be nonzero. If PD_INT_OVF is not defined or is set to zero, the packed decimal emulation routines will not generate integer overflow faults.

  The MACRO qualifier /ENABLE=OVERFLOW and directive .ENABLE OVERFLOW have no effect on overflow trap enabling for the packed decimal emulation routines. You must use PD_DEC_OVF and PD_INT_OVF.

- Trap routines for reserved operand, decimal divide by zero, integer overflow, and decimal overflow

  The emulation routines have their own trap routines for reserved operand, decimal divide by zero, integer overflow, and decimal overflow. Reserved operand and decimal divide by zero traps are always taken when the events occur. The overflow traps are taken only when they have been explicitly enabled. The traps call LIB$SIGNAL with a severity of fatal.

- Messages from the packed decimal emulation routines

All messages from the packed decimal emulation routines of the compiler use the following standard OpenVMS signal values: SS$_ROPRAND,SS$_DECOVF, SS$_INTOVF, and SS$_FLTDIV.

- Restriction on format of arguments

  Because these instructions are implemented by means of macros, there is one restriction on the format of the arguments. In a macro invocation, an initial circumflex (^) is interpreted to mean that the parameter is a string, and the character immediately following the circumflex is the string delimiter. Because of this, you cannot use arguments that begin with an operand type specification, such as ^x20(SP). Note that immediate mode arguments, such as #^XFF, can use an operand type specification because the circumflex is not the initial character.

2.9 Using Floating-Point Instructions

All floating-point instructions and directives, with the exception of POLY*x*, EMOD*x* and all H-Floating instructions, are supported.

These instructions are emulated via subroutine calls. This support is provided to allow hands-off compatibility for most existing VAX MACRO modules and is **not** designed for fast floating-point performance.

Besides the overhead of the emulation routine call, all floating-point operands must be passed through memory because the Alpha architecture does not have instructions to move values directly from the integer registers to the floating-point registers. In addition, on the first floating-point instruction, the FEN (Floating-point enable) bit is set for the process which will cause the entire floating-point register set to be saved and restored on every context switch for the life of the image.

2.9.1 Differences Between the VAX and Alpha Implementations

The differences between the implementations on VAX and Alpha systems are noted in the following list:

- Floating-point instructions and atomicity

  Since all floating-point instructions are emulated via subroutine calls, they are not atomic or restartable, and cannot be made atomic by the .PRESERVE ATOMICITY directive or /PRESERVE=ATOMICITY switch. Also, there is no guarantee that Alpha registers from R16 through R28 are preserved across the instruction.

- Use of argument registers (R16 through R21)

  Arguments are passed to the emulation routines by means of the argument registers (R16 through R21); attempts to use these registers as arguments in a floating-point instruction will not work correctly.

  The compiler converts references to the VAX argument pointer (AP) into references to the Alpha argument registers (see Section 2.3). For example, the compiler converts a parameter reference off the VAX AP, such as 4(AP), to Alpha R16.

  To prevent these problems with AP-based references when packed decimal or floating-point instructions are used, it is simplest to specify /HOME_ARGS=TRUE on the entry point of the routine that contains these references. This forces all AP-based references to be read from the procedure frame, rather than the argument registers, so that no changes need to be made to the instructions.

If /HOME_ARGS=TRUE is not used, the source code that contains parameter references based on the AP register as operands to packed-decimal instructions (or floating-point instructions) should be changed. Copy any AP-based operand to a temporary register first, and use that temporary register in the packed-decimal or floating-point instruction. For example, change the following code:

```
MOVP     R0,  @8(AP), @4(AP)
```

to:

```
MOVL    8(AP), R1
MOVL    4(AP), R2
MOVP    R0,(R1),(R2)
```

- D_floating format

  D_floating format is not fully supported in the Alpha architecture. All arithmetic operations or conversions must be done by converting D_floating format to G_floating format, doing the operation in G_floating format, and converting back to D_floating format. This results in the loss of the extra 3 bits of precision in the D_floating format mantissa, besides taking the time in conversion. This means that there is nothing to be gained by using D_floating format. It is available for compatibility only, with the caveat that some precision will be lost.

- Overflow traps

  It is not possible to enable traps on integer overflow or floating-point underflow. Alpha systems will always trap on floating-point overflow. The /ENABLE qualifier and the .ENABLE directive have no effect on overflow trapping. Overflow traps that lead to *predictable* results on VAX systems will give the same results on Alpha systems.

- Dirty zeros

  Code may need modification to eliminate *dirty zeros*. A true zero in all VAX floating-point formats has all bits set to zero. If the exponent bits are all zero but some of the remaining bits are set, this is a *dirty zero*, and the VAX system will treat this as a zero. An Alpha system will take a reserved operand trap.

- Restriction on format of arguments

  Because these instructions are implemented by means of macros, there is one restriction on the format of the arguments. In a macro invocation, an initial circumflex (^) is interpreted to mean that the parameter is a string, and the character immediately following the circumflex is the string delimiter. Because of this, you cannot use arguments that begin with an operand type specification, such as ^x20(SP). Note that immediate mode arguments, such as #^XFF, can use an operand type specification because the circumflex is not the initial character.

- Floating-point return values

  A MACRO program that calls out to a routine and expects a floating-point return value in R0 may require a “ jacket” between the call and the called routines. The purpose of the jacket is to move the returned value from floating-point register 0 to R0.

2.9.2 Impact on Routines in Other Languages

This support does not make the floating-point register set visible to the compiler. It simply allows floating point-operations to be done on the integer registers. This means that routines in other languages that want to interface with a VAX MACRO routine, either calling it or being called by it, must not expect any floating-point values as inputs or outputs. Compilers for other languages will pass these values in the floating-point registers. Floating-point arguments can be passed into or out of a VAX MACRO routine only by pointer.

Calls to run-time library (RTL) routines of other languages fall into this category. For example, a call to MTH$RANDOM returns a floating value in floating-point register F0. The compiler cannot directly read F0. You need to either create a jacket routine (in another language), which makes the call to MTH$RANDOM and then moves the result to R0, or write a separate routine that only does the move.

2.10 Preserving the Atomicity and Granularity of VAX MACRO

The VAX architecture includes instructions that perform a read-modify-write memory operation so that it appears to be a single, noninterruptible operation in a uniprocessing system. **Atomicity** is the term used to describe the ability to modify memory in one operation. Because the complexity of such instructions severely limits performance, read-modify-write operations on an Alpha system can be performed only by nonatomic, interruptible instruction sequences.

Furthermore, VAX instructions can address single aligned or unaligned byte, word, and longword locations in memory without affecting the surrounding memory locations. (A data item is considered **aligned** if its address is an even multiple of the item's size in bytes.) **Granularity** is the term used to describe the ability to independently write to portions of aligned longwords. Because byte, word, and unaligned longword access also severely limits performance, an Alpha system can only access aligned longword or quadword locations. Therefore, a sequence of instructions to write a single byte, word, or unaligned longword causes some of the surrounding bytes to be read and rewritten.

Both architectural differences can cause data to become corrupted under certain conditions. In an Alpha system, atomicity and granularity preservation are not provided by locking out other threads from modifying memory, but by providing a way to determine if a piece of memory may have been modified during the read-modify-write operation. In this case, the read-modify-write operation is retried.

To ensure data integrity, the compiler provides certain qualifiers and directives to be used for the conditions described in the following sections.

2.10.1 Preserving Atomicity

On VAX and Alpha multiprocessing systems alike, an application in which multiple, concurrent threads can modify shared data in a writable global section must have some way of synchronizing their access to that data. On a VAX single processor system, a memory modification instruction is sufficient to provide synchronized access to shared data. However, it is not sufficient on Alpha systems.

The compiler provides the /PRESERVE=ATOMICITY option to guarantee the integrity of read-modify-write operations for VAX instructions that have a memory modify operand. Alternatively, you can insert the .PRESERVE ATOMICITY and .NOPRESERVE ATOMICITY directives in sections of VAX MACRO source code as required to enable and disable atomicity.

For instance, the instruction INCL (R1) requires a read, modify, and write sequence on the data pointed to by R1. In a VAX system, the microcode performs these three operations. Therefore, an interrupt cannot occur until the sequence is fully completed. In an Alpha system, the following three instructions are required to perform the one VAX instruction:

```
        LDL     R28,(R1)
        ADDL    R28, 1,R28
        STL     R28,(R1)
```

The problem with this Alpha code sequence is that an interrupt can occur between any of the three instructions. If the interrupt causes an AST routine to execute or causes another process to be scheduled between the LDL and the STL, and the AST or other process updates the data pointed to by R1, the STL will store the result (R1) based on stale data.

When an atomic operation is required, and /PRESERVE=ATOMICITY (or .PRESERVE ATOMICITY) is specified, the compiler generates the following Alpha instruction sequence for INCL (R1):

```
Retry:  LDL_L   R28,(R1)
        ADDL    R28,#1,R28
        STL_C   R28,(R1)

        BEQ     R28, fail
          .
          .
          .
fail:   BR      Retry
```

If (R1) is modified by any other code thread on the current or any other processor during this sequence, the Store Longword Conditional instruction (STL_C) will not update (R1), but will indicate an error by writing 0 into R28. In this case, the code branches back and retries the operation until it completes without interference.

The BEQ Fail and BR Retry are done instead of a BEQ Retry because the branch prediction logic of the Alpha architecture assumes that backward conditional branches will be taken. Since this operation will rarely need to be retried, it is more efficient to make a forward conditional branch which is assumed *not* to be taken.

Because of the way atomicity is preserved on Alpha systems, this guarantee of atomicity applies to both uniprocessor and multiprocessor systems. This guarantee applies only to the actual modify instruction and does not extend interlocking to subsequent or previous memory accesses (see Section 2.10.6).

You should take special care in porting an application to an Alpha system if it involves multiple processes that modify shared data in a writable global section, even if the application executes only on a single processor. Additionally, you should examine any application in which a mainline process routine modifies data in process space that can also be modified by an asynchronous system trap (AST) routine or condition handler. See *Migrating to an OpenVMS AXP System:*

Recompiling and Relinking Applications[1] for a more complete discussion of the programming issues involved in read-modify-write operations in an Alpha system.

Warning

When preserving atomicity, the compiler generates aligned memory instructions that cannot be handled by the PALcode unaligned fault handler. They will cause a fatal reserved operand fault on unaligned addresses. Therefore, all memory references for which .PRESERVE ATOMICITY is specified *must* be to aligned addresses (see Section 2.10.5).

2.10.2 Preserving Granularity

To preserve the granularity of a VAX MACRO memory write instruction on a byte, word, or unaligned longword on Alpha means to guarantee that the instruction executes successfully on the specified data and preserves the integrity of the surrounding data.

The VAX architecture includes instructions that perform independent access to byte, word, and unaligned longword locations in memory so two processes can write simultaneously to different bytes of the same aligned longword without interfering with each other.

The Alpha architecture defines instructions that can address only aligned longword and quadword operands. On Alpha, code that writes a data field to memory that is less than a longword in length or is not aligned can do so only by using an interruptible instruction sequence that involves a quadword load, an insertion of the modified data into the quadword, and a quadword store. In this case, two processes that intend to write to different bytes in the same quadword will actually load, perform operations on, and store the whole quadword. Depending on the timing of the load and store operations, one of the byte writes could be lost.

The compiler provides the /PRESERVE=GRANULARITY option to guarantee the integrity of byte, word, and unaligned longword writes. The /PRESERVE=GRANULARITY option causes the compiler to generate Alpha instructions that provide granularity preservation for any VAX instructions that write to bytes, words, or unaligned longwords. Alternatively, you can insert the .PRESERVE GRANULARITY and .NOPRESERVE GRANULARITY directives in sections of VAX MACRO source code as required to enable and disable granularity preservation.

For example, the instruction MOVB R1, (R2) generates the following Alpha code sequence:

```
LDQ_U     R28,(R2)
MSKBL     R28,R2,R28
INSBL     R1,R2,R25
BIS       R25,R28,R25
STQ_U     R25,(R2)
```

If any other code thread modifies part of the data pointed to by (R2) between the LDQ_U and the STQ_U instructions, that data will be overwritten and lost.

[1] This manual has been archived but is available on the OpenVMS Documentation CD–ROM.

If you have specified that granularity be preserved for the same instruction, by either the command qualifier or the directive, the Alpha command sequence becomes the following:

```
          BIC       R2,#^B0111,R24
RETRY:    LDQ_L     R28,(R24)
          MSKBL     R28,R2,R28
          INSBL     R1,R2,R25
          BIS       R25,R28,R25
          STQ_C     R25,(R24)
          BEQ       R25, FAIL
           .
           .
           .
FAIL:     BR        RETRY
```

In this case, if the data pointed to by (R2) is modified by another code thread, the operation will be retried.

For a MOVW R1,(R2) instruction, the code generated to preserve granularity depends on whether the register R2 is currently assumed to be aligned by the compiler's register alignment tracking. If R2 is assumed to be aligned, the compiler generates essentially the same code as in the preceding MOVB example, except that it uses INSWL and MSKWL instructions instead of INSBL and MSKBL, and it uses #^B0110 in the BIC of the R2 address. If R2 is assumed to be unaligned, the compiler generates two separate LDQ_L/STQ_C pairs to ensure that the word is correctly written even if it crosses a quadword boundary.

Warning

The code generated for an aligned word write, with granularity preservation enabled, will cause a fatal reserved operand fault at run time if the address is not aligned. If the address being written to could ever be unaligned, inform the compiler that it should generate code that can write to an unaligned word by using the compiler directive .SET_REGISTERS UNALIGNED=R*n* immediately before the write instruction.

To preserve the granularity of a MOVL R1,(R2) instruction, the compiler always writes whole longwords with a STL instruction, even if the address to which it is writing is assumed to be unaligned. If the address is unaligned, the STL instruction will cause an unaligned memory reference fault. The PALcode unaligned fault handler will then do the loads, masks, and stores necessary to write the unaligned longword. However, since PALcode is noninterruptible, this ensures that the surrounding memory locations are not corrupted.

When porting an application to an Alpha system, you should determine whether the application performs byte, word, or unaligned longword writes to memory that is shared either with processes executing on the local processor, or with processes executing on another processor in the system, or with an AST routine or condition handler. See *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications* for a more complete discussion of the programming issues involved in granularity operations in an Alpha system.

Note

INSV instructions do not generate code that correctly preserves granularity when granularity is turned on.

2.10.3 Precedence of Atomicity Over Granularity

If you enable the preservation of both granularity and atomicity, and the compiler encounters VAX code that requires that both be preserved, atomicity takes precedence over granularity.

For example, the instruction INCW 1(R0), when compiled with .PRESERVE=GRANULARITY, retries the write of the new word value, if it is interrupted. However, when compiled with .PRESERVE=ATOMICITY, it will also refetch the initial value and increment it, if interrupted. If both options are specified, it will do the latter.

In addition, while the compiler can successfully generate code for unaligned words and longwords that preserves granularity, it cannot generate code for unaligned words or longwords that preserves atomicity. If both options are specified, all memory references must be to aligned addresses.

2.10.4 Examples When Atomicity Cannot Be Guaranteed

Because compiler atomicity guarantees only affect memory *modification* operands in VAX instructions, you should take special care in examining VAX MACRO sources for coding problems /PRESERVE=ATOMICITY cannot resolve. For instance, consider the following VAX instruction:

```
ADDL2 (R1),4(R1)
```

For this instruction, the compiler generates an Alpha code sequence such as the following, when /PRESERVE=ATOMICITY (or .PRESERVE ATOMICITY) is specified:

```
        LDL     R28,(R1)
Retry:  LDL_L   R24,4(R1)
        ADDL    R28,R24,R24
        STL_C   R24,4(R1)
        BEQ     fail
        .
        .
        .
fail:   BR      Retry
```

Note that, in this Alpha code sequence, when the STL_C fails, only the modify operand is reread before the add. The data (R1) is *not* reread. This behavior differs slightly from VAX behavior. In a VAX system, the entire instruction would execute without interruption; in an Alpha system, only the modify operand is updated atomically.

As a result, code that requires the read of the data (R1) to be atomic must use another method, such as a lock, to obtain that level of synchronization.

Consider another VAX instruction:

```
MOVL    (R1),4(R1)
```

For this instruction, the compiler generates an Alpha code sequence such as the following whether or not atomicity preservation was turned on:

```
LDL     R28,(R1)
STL     R28,4(R1)
```

The VAX instruction in this example is atomic on a single VAX CPU, but the Alpha instruction sequence is not atomic on a single Alpha CPU. Because the 4(R1) operand is a write operand and not a modify operand, the operation is not made atomic by the use of the LDL_L and STL_C.

Finally, consider a more complex VAX INCL instruction:

```
INCL    @(R1)
```

For this instruction, the compiler generates an Alpha code sequence such as the following, when /PRESERVE=ATOMICITY (or .PRESERVE ATOMICITY) is specified:

```
        LDL     R28,(R1)
Retry:  LDL_L   R24,(R28)
        ADDL    R24,#1,R24
        STL_C   R24,(R28)
        BEQ     fail
        .
        .
        .
fail:   BR      Retry
```

Here, only the update of the modify data is atomic. The fetch required to obtain the address of the modify data is not part of the atomic sequence.

2.10.5 Alignment Considerations for Atomicity

When preserving atomicity, the compiler must assume the modify data is aligned. An update of a field spanning a quadword boundary cannot occur atomically since this would require two read-modify-write sequences. Since software cannot handle an unaligned LD*x*_L or ST*x*_C instruction as it can a normal load or store instruction, a LD*x*_L or ST*x*_C instruction to an unaligned address will generate a fatal reserved operand fault.

When /PRESERVE=ATOMICITY (or .PRESERVE ATOMICITY) is specified, an INCL (R1) instruction generates LDL_L and STL_C instructions so R1 must be longword aligned.

For an INCW (R1) instruction, the compiler generates an Alpha code sequence such as the following:

```
        BIC     R1,#^B0110,R28  ; Compute Aligned Address
Retry:  LDQ_L   R24,(R28)       ; Load the QW with the data
        EXTWL   R24,R1,R23     ; Extract out the Word
        ADDL    R23,#1,R23      ; Increment the Word
        INSWL   R23,R1,R23     ; Correctly position the Word
        MSKWL   R24,R1,R24     ; Zero the spot for the Word
        BIS     R23,R24,R23     ; Combine Original and New word
        STQ_C   R23,(R28)       ; Conditionally store result
        BEQ     fail            ; Branch ahead on failure
        .
        .
        .
fail:   BR      Retry
```

Note that the first BIC instruction uses #^B0110, not #^B0111. This is to ensure that the word does not cross a quadword boundary, which would result in an incomplete memory update. If the address in R1 is not pointing to an aligned word, bit 0 will be set and the bit will not be cleared by the BIC instruction. The Load Quadword Locked instruction (LDQ_L) will then generate a fatal reserved operand fault.

An INCB instruction uses #^B0111 to generate the aligned address since all bytes are aligned.

2.10.6 Interlocked Instructions and Atomicity

The compiler's methods of preserving atomicity have an interesting side effect in compiled VAX MACRO code. On VAX systems, only the interlocked instructions will work correctly to synchronize access to shared data in multiprocessor systems. On Alpha multiprocessing systems, the code resulting from a compilation of modify instructions (with atomicity preserved) and interlocked instructions would both work correctly, because the LD*x*_L and ST*x*_C which the compiler generates for both sets of instructions operate correctly across multiple processors.

Because this compiler side effect is specific to Alpha systems and does *not* port back to VAX systems, you should avoid relying on it when porting VAX MACRO code to Alpha if you intend to run the code on both systems.

However, interlocked instructions must still be used if the memory modification is being used as an interlock for other instructions for which atomicity is not preserved. This is because the Alpha architecture does not guarantee strict write ordering. For example, consider the following VAX MACRO code sequence:

```
.PRESERVE ATOMICITY
INCL (R1)
.NOPRESERVE ATOMICITY
MOVL (R2),R3
```

This code sequence will generate the following Alpha code sequence:

```
Retry:  LDL_L   R28,(R1)
        ADDL    R28,#1,R28
        STL_C   R28,(R1)

        BEQ     R28, fail
        LDL     R3, (R2)
         .
         .
         .
fail:   BR      Retry
```

Because of the data prefetching of the Alpha architecture, the data from (R2) may be read before the store to (R1) is processed. If the INCL (R1) instruction is being used as a lock to prevent the data at (R2) from being accessed before the lock is set, the read of (R2) may occur before the increment of (R1) and thus is not protected.

The VAX interlocked instructions generate Alpha MB (memory barrier) instructions before and after the interlocked sequence. This prevents memory loads from being moved across the interlocked instruction.

For example, consider the following code sequence:

```
ADAWI     #1,(R1)
MOVL      (R2),R3
```

This code sequence will generate the following Alpha code sequence:

```
        MB
Retry:  LDL_L   R28,(R1)
        ADDL    R28,#1,R28
        STL_C   R28,(R1)

        BEQ     R28, Fail
        MB
        LDL     R3, (R2)
         .
         .
         .
Fail:   BR      Retry
```

The MB instructions cause all memory operations before the MB instruction to complete before any memory operations after the MB instruction are allowed to begin.

2.11 Compiling and Linking

The compiler requires the following files, one for compiling, the other for linking:

File	Description
SYS$LIBRARY:STARLET.MLB	Macro library that defines the compiler directives.
SYS$LIBRARY:STARLET.OLB	Object library containing emulation routines and other routines used by the compiler.

When you compile your code, the compiler automatically checks STARLET.MLB for definitions of compiler directives. Similarly, when you link your code, the linker links against STARLET.OLB to resolve undefined symbols.

The following is an example of a command procedure used to compile the MACRO-32 module [SYS]SYSSNDJBC.MAR:

```
$ SET DEFAULT WORK1:[PEAK.A.PORT]

$ MACRO/MIGRATION/LIS=LIS$SYSSNDJBC-ALPHA.LIS -
        ALPHA$LIBRARY:STARLET/LIB+ -
        ALPHA$LIBRARY:LIB/LIB+ -
        ALPHA$LIBRARY:ARCH_DEFS.MAR+ -
        SRC$SYSSNDJBC.MAR
$ MACRO/NOOBJECT/LIS=LIS$:SYSSNDJBC-VAX -
        VAX$LIBRARY:STARLET/LIB+ -
        VAX$LIBRARY:LIB/LIB+ -
        VAX$LIBRARY:ARCH_DEFS.MAR+ -
        SRC$SYSSNDJBC.MAR
$ EXIT
```

Not all modules need both libraries and many modules need component-specific libraries, but this example shows the basic approach to using the compiler.

Note

Compaq recommends that you use the latest version of the compiler. Make sure to use the version of SYS$LIBRARY:STARLET.MLB that ships with the compiler and make sure that the logical name points to the correct directory. Note that SYS$LIBRARY:STARLET.MLB is equivalent to ALPHA$LIBRARY:STARLET.MLB.

2.11.1 Line Numbering in Listing File

The macro expansion line numbering scheme in the listing file is X*nn*/*mmm*, where X*nn* shows the nesting depth and *mmm* is the line number relative to the outermost macro, as shown in the following example.

```
.MAIN.                  Source Listing    9-SEP-1996 11:36:03    AMAC V3.0-20-311D
                                         20-JUL-1992 11:05:38    X6AJ_RESD$:[SYSLIB]ARCH_DEFS.MAR;1

              00000000    1 ;
              00000000    2 ; This is the ALPHA (previously called "EVAX") version of ARCH_DEFS.MAR,
              00000000    3 ; which contains architectural definitions for compiling VMS sources
              00000000    4 ; for VAX and ALPHA systems.
              00000000    5 ;
00000001      00000000    6 EVAX = 1
00000001      00000000    7 ALPHA = 1
00000001      00000000    8 BIGPAGE = 1
00000020      00000000    9 ADDRESSBITS = 32
              00000000   10         .macro  test1
              00000000   11         clrl    r1
              00000000   12         clrl    r2
              00000000   13         tstl    48(sp)    ; generate uplevel stack error
              00000000   14         clrl    r3
              00000000   15         .endm
              00000000   16         .macro  test2
              00000000   17         clrl    r4
              00000000   18         clrl    r5
              00000000   19         test1
              00000000   20         clrl    r6
              00000000   21         .endm
              00000000   22
              00000000   23  foo:    .jsb_entry
                 .
                 .
                 .
              00000000   44         clrl    r0
              00000011   45         test2
                          1.......
%AMAC-E-UPLEVSTK, (1) up-level stack reference in routine FOO

    X01/001   00000002            clrl    r4
    X01/002   00000004            clrl    r5
    X01/003   00000006            test1
    X02/004   00000006            clrl    r1
    X02/005   00000008            clrl    r2
    X02/006   0000000A            tstl    48(sp)     ; generate uplevel stack error
    X02/007   0000000D            clrl    r3
    X02/008   0000000F
    X01/009   0000000F            clrl    r6
    X01/010   00000011
              00000011   46         rsb
              00000012   47         .end
```

2.11.2 Linking an Object Module

To link the object files produced by the compiler, use the following commands as a basis:

```
$ @ALPHA$TOOLS:LINK       ! Set up DCL and Logical to EXE
$ LINK/ALPHA image_name,object1,object2,...
```

For certain VAX instructions (such as the divide instructions and others described in this manual), the compiler produces object code that issues a call to the OpenVMS General-Purpose Run-Time Library (OTS$ RTL). By default, the linker links against the library that that contains these routines.

2.12 Debugging

The compiler provides full debugger support. The debug session for compiled VAX MACRO code is similar to that for assembled VAX MACRO code. However, there are some important differences that are described in this section. For a complete description of debugging, see the *OpenVMS Debugger Manual*.

2.12.1 Code Relocation

One major difference is that the code is compiled rather than assembled. On a VAX system, each VAX MACRO instruction is a single machine instruction. On an Alpha system, each VAX MACRO instruction may be compiled into many Alpha machine instructions. A major side effect of this difference is the relocation and rescheduling of code if you do not specify /NOOPTIMIZE in your compile command.

By default, several optimizations are performed that cause the movement of generated code across source boundaries (see Section 1.2, Section 4.3, and Appendix A). For most code modules, debugging is simplified if you compile with /NOOPTIMIZE, which prevents this relocation from happening. After you have debugged your code, you can recompile without /NOOPTIMIZE to improve performance.

2.12.2 Symbolic Variables for Routine Arguments

Another major difference between debugging compiled code and debugging assembled code is a new concept to VAX MACRO, the definition of symbolic variables for examining routine arguments. On VAX systems, when you are debugging a routine and want to examine the arguments, you typically do something like the following:

```
DBG> EXAMINE @AP        ; to see the argument count
DBG> EXAMINE @AP+4      ; to examine the first arg
```

or

```
DBG> EXAMINE @AP        ; to see arg count
DBG> EXAMINE .+4:.+20   ; to see first 5 args
```

On Alpha systems, the arguments do not reside in a vector in memory as they do on VAX systems. Furthermore, there is no AP register on Alpha systems. If you type EXAMINE @AP when debugging VAX MACRO compiled code, the debugger reports that AP is an undefined symbol.

In the compiled code, the arguments can reside in some combination of:

- Registers
- On the stack above the routine's stack frame
- In the stack frame, if the argument list was homed (see Section 2.3) or if there are calls out of the routine that would require the register arguments to be saved

The compiler does not require that you figure out where the arguments are by reading the generated code. Instead, it provides $ARG*n* symbols that point to the correct argument locations. The $ARG0 symbol is the same as @AP+0 is on VAX systems, that is, the argument count. The $ARG1 symbol is the first argument, $ARG2 is the second argument, and so forth. These symbols are defined in CALL_ENTRY and JSB_ENTRY directives, but not in EXCEPTION_ENTRY directives.

2.12.3 Locating Arguments Without $ARG*n* Symbols

There may be additional arguments in your code for which the compiler did not generate a $ARG*n* symbol. The number of $ARG*n* symbols defined for a .CALL_ENTRY routine is the maximum number detected by the compiler (either by automatic detection or as specified by MAX_ARGS) or 16, whichever is less. For a .JSB_ENTRY routine, since the arguments are homed in the caller's stack frame and the compiler cannot detect the actual number, it always creates eight $ARG*n* symbols.

In most cases, you can easily find any additional arguments, but in some cases you cannot.

2.12.3.1 Additional Arguments That Are Easy to Locate

You can easily find additional arguments if:

- The argument list is not homed, and $ARG*n* symbols are defined to $ARG7 or higher. If the argument list is not homed, the $ARG*n* symbols $ARG7 and above always point into the list of parameters passed as quadwords on the stack. Subsequent arguments will be in quadwords following the last defined $ARG*n* symbol.
- The argument list is homed, and you want to examine an argument that is less than or equal to the maximum number detected by the compiler (either by automatic detection or as specified by MAX_ARGS). If the argument list is homed, $ARG*n* symbols always point into the homed argument list. Subsequent arguments will be in longwords following the last defined $ARG*n* symbol.

For example, you can examine arguments beyond the eighth argument in a JSB routine (where the argument list must be homed in the caller), as follows:

```
DBG> EX $ARG8  ; highest defined $ARGn
.
.
.
DBG> EX .+4  ; next arg is in next longword
.
.
.
DBG> EX .+4  ; and so on
```

This example assumes that the caller detected at least 10 arguments when homing the argument list.

To find arguments beyond the last $ARG*n* symbol in a routine that did not home the arguments, proceed exactly as in the previous example except substitute EX .+8 for EX .+4.

2.12.3.2 Additional Arguments That Are Not Easy to Locate

You cannot easily find additional arguments if:

- The argument list is not homed, and $ARG*n* symbols are defined only as high as $ARG6. In this case, the existing $ARG*n* symbols will either point to registers or to quadword locations in the stack frame. In both cases, subsequent arguments cannot be examined by looking at quadword locations beyond the defined $ARG*n* symbols.

- The argument list is homed, and you want to examine arguments beyond the number detected by the compiler. The $ARG*n* symbols point to the longwords that are stored in the homed argument list. The compiler only moves as many arguments as it can detect into this list. Examining longwords beyond the last argument that was homed will result in examining various other stack context.

The only way to find the additional arguments in these cases is to examine the compiled machine code to determine where the arguments reside. Both of these problems are eliminated if MAX_ARGS is specified correctly for the maximum argument that you want to examine.

2.12.4 Debugging Code with Packed Decimal Data

The following list provides important information about debugging compiled VAX MACRO code with packed decimal data on an Alpha system:

1. When using the EXAMINE command to examine a location that was declared with a .PACKED directive, the debugger automatically displays the value as a packed decimal data type.
2. You can deposit packed decimal data. The syntax is the same as it is on VAX.

2.12.5 Debugging Code with Floating-Point Data

The following list provides important information about debugging compiled VAX MACRO code with floating-point data on an Alpha system:

1. You can use the EXAMINE/FLOAT command to examine an Alpha integer register for a floating-point value.

   Even though there is a set of registers for floating-point operations on Alpha systems, those registers are not used by compiled VAX MACRO code that contains floating-point operations. Only the Alpha integer registers are used.

   Floating-point operations in compiled VAX MACRO code are performed by emulation routines that operate outside the compiler. Therefore, performing VAX MACRO floating-point operations on, say, R7, has no effect on Alpha floating-point Register 7.
2. When using the EXAMINE command to examine a location that was declared with a .FLOAT directive or other floating-point storage directives, the debugger automatically displays the value as floating-point data.
3. When using the EXAMINE command to examine the G_FLOAT data type, the debugger does not use the contents of two registers to build the value for VAX data.

   Consider the following example:

   ```
   EXAMINE/G_FLOAT   R4
   ```

   In this example, the lower longwords of R4 and R5 are not used to build the value as is the case on VAX. Instead, the quadword contents of R4 are used.

   The code the compiler generates for D_FLOAT and G_FLOAT operations preserves the VAX format of the data in the low longwords of two consecutive registers. Therefore, using EXAMINE/G_FLOAT on either of these two registers will not give the true floating-point value, and issuing DEPOSIT/G_FLOAT to one of these registers will not give the desired results. You can manually combine the two halves of such a value, however. For example, assume you executed the following instruction:

```
MOVG    DATA, R6
```

You could then read the G_FLOAT value which now resides in R6 and R7 with a sequence like the following:

```
DBG> EX R6
.MAIN.\%LINE 100\%R6:   0FFFFFFFF D8E640D1
DBG> EX R7
.MAIN.\%LINE 100\%R7:   00000000 2F1B24DD
DBG> DEP R0 = 2F1B24DDD8E640D1
DBG> EX/G_FLOAT R0
.MAIN.\%LINE 100\%R0:   4568.89900000000
```

4. You can deposit floating-point data in an Alpha integer register with the DEPOSIT command. The syntax is the same as it is on a VAX system.
5. H_FLOAT is unsupported.

3
Recommended and Required Source Changes

This chapter describes the coding constructs you should examine when porting VAX MACRO code to OpenVMS Alpha. The occurrence of any of these in a module can make porting take longer than it would otherwise. Although the compiler can identify many of these practices and flag them with diagnostic messages, recognizing them yourself will speed up the porting effort.

In most cases, it will be necessary to change your source code. The exceptions are noted.

The coding constructs described in this chapter are:

- Stack usage (Section 3.1)
- Instruction stream (Section 3.2)
- Flow control mechanisms (Section 3.3)
- Dynamic image relocation (Section 3.4)
- Overwriting static data (Section 3.5)
- Static initialization using external symbols (Section 3.6)
- Transfer vectors (Section 3.7)
- Arithmetic exceptions (Section 3.8)
- Page size (Section 3.9)
- Locking pages into a working set (Section 3.10)
- Synchronization (Section 3.11)

3.1 Stack Usage

The OpenVMS calling standard defines a stack frame format for Alpha systems substantially different from that defined for VAX systems. If your code relies on the format of the VAX stack frame you will need to change it when porting it to an Alpha system.

3.1.1 References to the Procedure Stack Frame

The compiler disallows references to positive stack offsets from FP, and flags them as errors. A single exception to this rule is the practice whereby VAX MACRO code establishes a dynamic condition handler by moving a routine address to the stack location pointed to by FP. The compiler detects this and generates the appropriate Alpha code for establishing such a handler. However, if the write to 0(FP) occurs inside a JSB routine, the compiler will flag it as an error. The compiler allows negative FP offsets, as used for referring to stack storage allocated at procedure entry.

Recommended Change

If possible, remove stack frame references entirely, rather than converting to the Alpha format. For example, if the offending code was attempting to change saved register values, store the new value in a stack temporary and set the register value on routine exit (removing the register from the entry register mask).

3.1.2 References Outside the Current Stack Frame

By monitoring stack depth throughout a VAX MACRO module, the compiler detects references in a routine to data pushed on the stack by its caller and flags them as errors.

Recommended Change

You must eliminate references in a routine to data pushed on the stack by its caller. Instead, pass the required data as parameters or pass a pointer to the stack base from which the data can be read.

3.1.3 Nonaligned Stack References

At routine calls, the compiler octaword-aligns the stack, if the stack is not already octaword-aligned. Some code, when building structures on the stack, makes unaligned stack references or causes the stack pointer to become unaligned. The compiler flags both of these with information-level messages.

Recommended Change

Provide sufficient padding in data elements or structures pushed onto the stack, or change data structure sizes. Because unaligned stack references also have an impact on VAX performance, you should apply these fixes to code designed for both the VAX and Alpha architectures.

3.1.4 Building Data Structures on the Stack

A common coding practice is to produce a structure on the stack by pushing the elements and relying on auto decrement to move the stack pointer to allocate space. The problems with this technique follow:

- If the SDL source for the structure being built is modified to pad the structure in order to align fields, the code will be broken. If the code does not contain assume statements this will not be detected at compile time.
- If the size of a field is extended, the current instruction will not access all the data and this will also go undetected.

Recommended Change

To correct the first problem and detect the second, use the coding technique illustrated in this example. Consider the following code example:

```
; Build a descriptor on the stack.
;
MOVW    length, -(SP)
MOVB    type,   -(SP)
MOVB    class,  -(SP)
MOVAL   buffer, -(SP)
```

Replace this code with the following:

```
SUBL2   DSC$S_DSCDEF, SP  ; pre-allocate space on stack
MOVW    length, DSC$W_LENGTH(SP)
MOVB    type,   DSC$B_DTYPE(SP)
MOVB    size,   DSC$B_CLASS(SP)
MOVAL   buffer, DSC$A_POINTER(SP)
```

3.1.5 Quadword Moves Into the VAX SP and PC

Due to architectural differences between VAX and Alpha computers, it is not possible to completely emulate quadword moves into the VAX stack pointer (SP) and the program counter (PC) without programmer intervention. The VAX architecture defines R14 as the SP and R15 as the PC. A MOVQ instruction with SP as the target would simultaneously load the VAX SP and PC, as shown in the following example:

```
MOVQ    R0,SP   ; Contents of R0 to SP, R1 to PC
```

```
MOVQ    REGDATA, SP   ; REGDATA to SP
                      ; REGDATA+4 to PC
```

If the compiler encounters a MOVQ instruction with SP as the destination, it generates a sign-extended longword load from the supplied source into R30 (the Alpha stack pointer) and issues the following informational message:

```
%AMAC-I-CODGENINF, (1) Longword update of Alpha SP, PC untouched
```

Recommended Change

If the intended use of the MOVQ instruction was to achieve the VAX behavior, a MOVL instruction should be used, followed by a branch to the intended address, as shown next:

```
MOVL    REGDATA, SP     ; Load the SP
MOVL    REGDATA+4, R0   ; Get the new PC
JMP     (R0)            ; And Branch
```

If the intended use of the MOVQ instruction was to load the stack pointer with an 8-byte value, the EVAX_LDQ built-in should be used instead, as shown next:

```
EVAX_LDQ        SP, REGDATA
```

3.2 Instruction Stream

The following VAX MACRO coding practices and VAX instructions either do not work on OpenVMS Alpha or they can produce unexpected results.

3.2.1 Data Embedded in the Instruction Stream

The compiler detects data embedded in the instruction stream, and reports it as an error.

Data in the instruction stream often takes the form of a JSB instruction followed by a .LONG. This construct allows VAX MACRO code to implicitly pass a parameter to a JSB routine which locates the data by using the return address on the stack. Another occasional use of data in the code stream is to make the data contiguous with the code in memory, so that it can be relocated as a unit.

Recommended Change

For implicit JSB parameters, pass the parameter value in a register. For values larger than a longword, put the data in another program section (psect) and explicitly pass its address.

Because static data must reside in a separate data psect, any code that tries to relocate code and data together must be rewritten.

3.2.2 Run-Time Code Generation

The compiler detects branches to stack locations and to static data areas and flags them as errors.

Recommended Change

You must either remove or modify code that builds instructions for later execution, branches to stack locations, or branches to static data areas. If the code is absolutely necessary, you should conditionalize it for VAX, and generate corresponding, suitable Alpha code.

3.2.3 Dependencies on Instruction Size

Code that computes branch offsets based on instruction lengths, for example, must be changed.

Recommended Change

Use a label and standard branch or a CASE instruction for computed GOTOs.

3.2.4 Incomplete Instructions

Some CASE instructions in OpenVMS VAX code are not followed by offset tables, but instead depend on psect placement by the linker to complete the instruction. The compiler will flag the incomplete instruction as an error.

Recommended Change

Complete the instruction in the module or build a table of addresses in a data psect, and replace the CASE instruction with code to select a destination address from the table and branch.

3.2.5 Untranslatable VAX Instructions

Because the compiler cannot translate the following VAX instructions, it flags them as errors:

- LDPCTX and SVPCTX
- XFC
- ESCD, ESCE, and ESCF
- BUG*x*

Recommended Change

These instructions usually appear in code that is highly dependent on the VAX architecture. You will need to rewrite such code to port it to Alpha systems.

3.2.6 References to Internal Processor Registers

Pay special attention to the following instructions:

- MFPR
- MTPR

Recommended Change

Verify that they reference valid Alpha internal processor registers (IPRs). If they do not, they will be flagged. For more information about the Alpha internal processor registers, refer to the *Alpha Architecture Reference Manual*.

3.2.7 Use of Z and N Condition Codes with the BICPSW Instruction

The BICPSW instruction is supported, but the Z and N condition codes cannot be set at the same time. Setting the Z condition code will clear the N condition code and vice versa.

Recommended Change

If you find that your code sets both condition codes at the same time, modify the code.

3.2.8 Interlocked Memory Instructions

The *Alpha Architecture Reference Manual, Third Edition* describes strict rules for using interlocked memory instructions. In particular, branches within or into LD*x*L/ST*x*C sequences are not allowed. Branches out of interlocked sequences are valid and need not change. The Alpha 21264 (EV6) processor and all subsequent Alpha processors are more stringent than their predecessors in their requirement that these rules be followed. The Alpha 21264 processor was first supported by OpenVMS Alpha Version 7.1-2.

The MACRO-32 compiler observes these rules in the code it generates from MACRO-32 source code. However, the compiler provides EVAX_LQ*x*L and EVAX_ST*x*C built-ins which enable programmers to write these instructions directly in source code.

To help ensure that these instructions are used in conformance with the rules for using interlocked memory instructions, additional checking was added to the MACRO-32 compiler, starting in Version 3.1 of the compiler for OpenVMS Alpha Version 7.1-2 and in Version 4.1 for OpenVMS Alpha Version 7.2.

The compiler reports the following warning messages under the circumstances described:

BRNDIRLOC, branch directive ignored in locked memory sequence

Explanation: The compiler found a .BRANCH_LIKELY directive within an LDx_L/STx_C sequence.

User Action: None. The compiler will ignore the .BRANCH_LIKELY directive and, unless other coding guidelines are violated, the code will work as written.

BRNTRGLOC, branch target within locked memory sequence in routine 'routine_name'

Explanation: A branch instruction has a target that is within an LDx_L/STx_C sequence.

User Action: To avoid this warning, rewrite the source code to avoid branches within or into LDx_L/STx_C sequences. Branches out of interlocked sequences are valid and are not flagged.

MEMACCLOC, memory access within locked memory sequence in routine 'routine_name'

Explanation: A memory read or write occurs within an LDx_L/STx_C sequence. This can be either an explicit reference in the source code, such as "MOVL data, R0", or an implicit reference to memory. For example, fetching the address of a data label (e.g., "MOVAB label, R0") is accomplished by a read from the linkage section, the data area that is used to resolve external references.

User Action: To avoid this warning, move all memory accesses outside the LDx_L/STx_C sequence.

RETFOLLOC, RET/RSB follows LDx_L instruction

Explanation: The compiler found a RET or RSB instruction after an LDx_ L instruction and before finding an STx_C instruction. This indicates an ill-formed lock sequence.

User Action: Change the code so that the RET or RSB instruction does not fall between the LDx_L instruction and the STx_C instruction.

RTNCALLOC, routine call within locked memory sequence in routine 'routine_ name'

Explanation: A routine call occurs within an LDx_L/STx_C sequence. This can be either an explicit CALL/JSB in the source code, such as "JSB subroutine", or an implicit call that occurs as a result of another instruction. For example, some instructions such as MOVC and EDIV generate calls to run-time libraries.

User Action: To avoid this warning, move the routine call or the instruction that generates it, as indicated by the compiler, outside the LDx_L/STx_C sequence.

STCMUSFOL, STx_C instruction must follow LDx_L instruction

Explanation: The compiler found an STx_C instruction before finding an LDx_L instruction. This indicates an ill-formed lock sequence.

User Action: Change the code so that the STx_C instruction follows the LDx_L instruction.

Recommended Change

If the compiler detects any nonconformant use of interlocked memory instructions, follow the recommended user actions described with these warning messages.

3.2.9 Use of the MOVPSL Instruction

The MOVPSL instruction fetches the Alpha PS, whose contents differ from the VAX PSL (processor status longword). For example, the Alpha PS does not contain the condition code bits. For more information about the Alpha PS, refer to the *Alpha Architecture Reference Manual*.

Recommended Change

The MOVSPL instruction is rarely used. If your code contains the MOVPSL instruction, you will need to rewrite the code to port it to Alpha systems.

3.3 Flow Control Mechanisms

Certain flow control mechanisms used with VAX MACRO do not produce the desired results on Alpha systems. Therefore, some changes to your code are either recommended or required.

Included in this category are several frequently used variations of modifying the return address on the stack, from within a JSB routine, to change the flow of control. All must be recoded.

3.3.1 Communication by Condition Codes

The compiler detects a JSB instruction followed immediately by a conditional branch, or a conditional branch as the first instruction in a routine, and generates an error message.

Recommended Change

Return a status value or a flag parameter to take the place of implicit communication by means of condition codes.

For example:

```
BSBW    GET_CHAR
BNEQ    ERROR           ; Or BEQL, or BLSS or BGTR, etc
```

can be replaced with:

```
BSBW    GET_CHAR
BLBC    R0, ERROR       ; Or BLBS
```

If you are already using R0, you must push it onto the stack and restore it later when you have handled the error.

3.3.2 Branches from JSB Routines into CALL Routines

The compiler will flag, with an information-level message, a branch from a JSB routine into a CALL routine, if the .JSB_ENTRY routine saves registers. The reason such a branch is flagged is because the procedure's epilogue code to restore the saved registers will not be executed. If the registers do not have to be restored, no change is necessary.

Recommended Change

The .JSB_ENTRY routine is probably trying to execute a RET on behalf of its caller. If the registers that were saved by the JSB_ENTRY must be restored before executing the RET, change the common code in the .CALL_ENTRY to a .JSB_ENTRY which may be invoked from both routines.

For example, consider the following code example:

```
Rout1:  .CALL_ENTRY
        .
        .
X:      .
        .
        .
        RET
Rout2:  .JSB_ENTRY INPUT=<R1,R2>, OUTPUT=<R4>, PRESERVE=<R3>
        .
        .
        BRW X
        .
        .
        RSB
```

To port such code to an Alpha system, break the .CALL_ENTRY routine into two routines, as follows:

```
Rout1:  .CALL_ENTRY
        .
        .
        JSB X
        RET
X:      .JSB_ENTRY INPUT=<R1,R2>, OUTPUT=<R4>, PRESERVE=<R3>
        .
        .
        RSB
Rout2:  .JSB_ENTRY INPUT=<r1,r2>, OUTPUT=<R4>, PRESERVE=<R3>
        .
        .
        JSB X
        RET
        .
        .
        RSB
```

3.3.3 Pushing an Address onto the Stack

The compiler detects any attempt to push an address onto the stack (for instance, PUSHAB label) to cause a subsequent RSB to resume execution at that location and flags this practice as an error. (On VAX systems, the next RSB would return to the routine's caller.)

Recommended Change

Remove the PUSH of the address, and add an explicit JSB to the target label just before the current routine's RSB. This will result in the same control flow. Declare the target label as a .JSB_ENTRY point.

For example, the compiler would flag the following code as requiring a source change.

```
Rout:   .JSB_ENTRY
        .
        .
        PUSHAB  continue_label
        .
        .
        RSB
```

By adding an explicit JSB instruction, you could change the code as follows. Note that you would place the JSB just before the RSB. In the previous version of the code, it is the RSB instruction that transfers control to *continue_label*, regardless of where the PUSHAB occurs. The PUSHAB is removed in the new version.

```
Rout:   .JSB_ENTRY
        .
        .
        .
        JSB     continue_label
        RSB
```

3.3.4 Removing the Return Address from the Stack

The compiler detects the removal of the return address from the stack (for instance, TSTL (SP)+) and flags this practice as an error. The removal of a return address in VAX code allows a routine to return to its caller's caller.

Recommended Change

Rewrite the routine so it returns a status value to its caller that indicates that the caller should return to its caller. Alternatively, the initial caller could pass the address of a continuation routine, to which the lowest-level routine can JSB. When the continuation routine RSBs back to the lowest-level routine, the lowest-level routine RSBs.

For example, the compiler would flag the following code as requiring a source change:

```
Rout1:  .JSB_ENTRY
        .
        .
        JSB   Rout2
        .
        RSB
Rout2:  .JSB_ENTRY
        .
        .
        JSB   Rout3       ; May return directly to Rout1
        .
        RSB
Rout3:  .JSB_ENTRY
        .
        .
        TSTL  (SP)+       ; Discard return address
        RSB               ; Return to caller's caller
```

You could rewrite the code to return a status value, as follows:

```
Rout2:  .JSB_ENTRY
        .
        .
        JSB          Rout3
        BLBS R0, No_ret   ; Check Rout3 status return
        RSB               ; Return immediately if 0
No_ret: .
        .
        RSB
Rout3:  .JSB_ENTRY
        .
        .
        CLRL  R0          ; Specify immediate return from caller
        RSB               ; Return to caller's caller
```

3.3.5 Modifying the Return Address

The compiler detects any attempt to modify the return address on the stack and flags it as an error.

Recommended Change

Rewrite the code that modifies the return address on the stack to return a status value to its caller instead. The status value causes the caller to either branch to a given location or contains the address of a special .JSB_ENTRY routine the caller should invoke. In the latter case, the caller should RSB immediately after issuing the JSB to a special .JSB_ENTRY routine.

For example, the compiler would flag the following code as requiring a source change.

```
Rout1:  .JSB_ENTRY
        .
        .
        JSB  Rout2                    ; Might not return
        .
        RSB
Rout2:  .JSB_ENTRY
        .
        .
        MOVAB continue_label, (SP)   ; Change return address
        .
        RSB
```

You could rewrite the code to incorporate a return value as follows:

```
Rout1:  .JSB_ENTRY
        .
        .
        JSB   Rout2
        TSTL  R0                      ; Check for alternate return
        BEQL  No_ret                  ; Continue normally if 0
        JSB   (R0)                    ; Call specified routine
        RSB                           ; and return
No_ret: .
        .
        RSB
Rout2:  .JSB_ENTRY
        CLRL  R0
        .
        .
        MOVAB  continue_label, R0    ; Specify alternate return
        RSB
```

3.3.6 Coroutine Calls

Coroutine calls between two routines are generally implemented as a set of JSB instructions within each routine. Each JSB transfers control to a return address on the stack, removing the return address in the process (for instance, (JSB @(SP)+). The compiler detects coroutine calls and flags them as errors.

Recommended Change

You must rewrite the routine that initiates the coroutine linkage to pass an explicit callback routine address to the other routine. The coroutine initiator should then invoke the other routine with a JSB instruction.

For example, consider the following coroutine linkage:

```
Rout1:  .JSB_ENTRY
        .
        JSB Rout2   ; Rout2 will call back as a coroutine
        .
        JSB @(SP)+  ; Coroutine back to Rout2
        .
        RSB
Rout2:  .JSB_ENTRY
        .
        JSB @(SP)+  ; coroutine back to Rout1
        .
        RSB
```

You could change the routines participating in such a coroutine linkage to exchange explicit callback routine addresses (here, in R6 and R7) as follows:

```
Rout1:  .JSB_ENTRY
        .
        .
        MOVAB Rout1_callback, R6
        JSB Rout2
        RSB
Rout1_callback: .JSB_ENTRY
        .
        .
        JSB  (R7)    ; Callback to Rout2
        .
        RSB
Rout2:  .JSB_ENTRY
        .
        .
        MOVAB Rout2_callback, R7
        JSB (R6)    ; Callback to Rout1
        RSB
Rout2_callback: .JSB_ENTRY
        .
        RSB
```

To avoid consuming registers, the callback routine addresses could be pushed onto the stack at routine entry. Then, JSB @(SP)+ instructions could still be used to perform “direct” JSBs to the callback routines. In the following example, the callback routine addresses are passed in R0, but pushed immediately at routine entry:

```
Rout1:  .JSB_ENTRY
        .
        .
        MOVAB Rout1_callback, R0
        JSB rout2
        RSB
Rout1_callback: .JSB_ENTRY
        PUSHL   R0      ; Push callback address received in R0
        .
        .
        JSB     @(SP)+  ; Callback to rout2
        .
        .
        RSB
Rout2:  .JSB_ENTRY
        PUSHL   R0      ; Push callback address received in R0
        .
        .
        MOVAB Rout2_callback, R0
        JSB @(SP)+      ; Callback to Rout1
        RSB
Rout2_callback: .JSB_ENTRY
        .
        .
        RSB
```

3.3.7 Using REI to Change Modes

A common VAX use of the REI instruction is to change modes by pushing an explicit target PC and PSL on the stack. This code cannot be compiled without some changes to the source code for the following reasons:

- The destination code requires that a linkage section pointer be established. REI does not provide a way to do this.
- An REI frame on an Alpha system is more complex than on a VAX system and includes saved registers. In addition, all subroutines have epilogue code to restore saved, nonscratch registers. A new syntax would be necessary to accommodate register passing and restoration.
- The mode change means that the process will be executing on a different stack at the target. This places new requirements on cleaning up the previous stack.

Recommended Change

A system routine, EXE$REI_INIT_STACK, has been created to perform the corresponding function for Alpha systems. This routine accepts the new mode and a callback routine address as parameters. This routine has the advantage of being usable from higher level languages as well.

You must restructure existing code so that this routine call can be used. The code label where execution is to continue must be declared with an entry point directive, since it will be called by the system routine.

The following examples show two ways that the REI instruction is used in VAX MACRO code for changing modes and one way to accomplish the same thing using the EXE$REI_INIT_STACK routine for Alpha.

Before (1)

```
PUSH  new_PSL
PUSHl new_PC
REI
```

Before (2)

```
PUSHl     new_PSL
JSB       10$
 .
 .
 .
CONTINUE                  ;With new PSL
 .
 .
 .
10$:  REI
```

After

```
PUSHL     Continuation_routine
PUSH      new_mode        ;Not a PSL
CALLS #2, exe$rei_init_stack
 .
 .
 .
Continuation routine:  .JSB_ENTRY
```

When your program reaches the continuation routine it will be executing in a new mode at a new location, the FP will be cleared, and the old mode stack will be reinitialized.

3.3.8 Loop Nesting Limit

The compiler must identify loops in program code in order to generate code correctly. A loop is a “nested” loop if it is inside another loop or if it partially overlaps another loop. If loops are nested more than 32 levels deep, the compiler will report a fatal error, and compilation will terminate. The VAX MACRO–32 assembler did not need to identify loops, and so did not enforce such a restriction.

Recommended Change

If the compiler reports this error, restructure the code so that the program does not exceed the limit.

3.4 Dynamic Image Relocation

On VAX systems, you can copy position independent code (PIC) from one address range to another, and it assembles and runs correctly. If you compile this code for Alpha, which includes a range of code that you copied using source code labels, it does not work for two reasons. First, the compiled code may not be in the same order as the source code. Second, the code requires the linkage section to make external references. Not only is the linkage section in another psect, but it also contains absolute addresses fixed up by the linker and image activator.

Recommended Change

Replicate the code or otherwise eliminate the copy of code from one address range to another.

3.5 Overwriting Static Data

The VAX MACRO assembler allows complete freedom in overwriting previously initialized static storage. This is usually done by changing the current location counter with a “.=” construct.

By contrast, the MACRO-32 compiler restricts overwriting so that partial overwriting of an existing data item is not allowed, except for .ASCII data. You may overwrite:

- Any scalar item with another of the same size
- Any storage left blank (declared with .BLK*x* or “.=.+*n*”)
- Sections of .ASCII data with other .ASCII or .BYTE directives

Recommended Change

Where possible, change your code to one of the following forms that is allowed:

- Code that overwrites data declared by any of the scalar storage directives (.BYTE, .WORD, or .LONG, and so forth) using a directive of the same type or one that occupies the same number of bytes. The items must be the same size at the same location; they cannot partially overlap. For instance:

  ```
  LAB1:   .WORD 1
          .WORD 2
          .=LAB1
          .WORD 128
  ```

- Partial overwriting of .ASCII data

You can overwrite portions of previously written .ASCII data using .ASCII or .BYTE. (Since .ASCIZ and .ASCIZ are implemented as a pair of .ASCII/ .BYTE directives, they can also overwrite .ASCII). For example:

```
DATA:  .ascii /abcdefg/
.=data
    .ascii /z/          ; change "a" to "z"
.=data
    .byte   0           ; change "z" to 0
.=data+4
    .ascii /xyz/        ; change "efg" to "xyz"
```

The new data must be completely within the bounds of the previous .ASCII string. The following is illegal:

```
DATA:  .ascii /abcdefg/
.=data+4
       .ascii /lmnop/      ; exceeds end of previous .ASCII
```

Partial overwriting with other directive types (.LONG, and so forth), is not allowed.

3.6 Static Initialization Using External Symbols

Some forms of static initialization using external symbols are not allowed.

Recommended Change

Where possible, change your code to one of the forms that is allowed. This can often be done by defining additional symbols using $*xxx*DEF macros. The compiler includes support for expressions of the form:

`<symbol1 +/- offset1>` *`OPR`* `<symbol2 +/- offset2>`

where *OPR* can be:

+ : Add
- : Subtract
* : Multiply
/ : Divide
@ : Arithmetic shift
& : Logical inclusive AND
! : Logical inclusive OR
\ : Logical exclusive OR

where **symbol1** and **symbol2** (both optional) are external values or labels in another psect, and where **offset1** and **offset2** may be expressions but must resolve to compile-time constants.

If a static initialization expression cannot be reduced to this form, the compiler will report an illegal static initialization. Use brackets to ensure correct operator precedence.

3.7 Transfer Vectors

The compiler flags any occurrence of the .TRANSFER directive in VAX MACRO code as an error unless you have specified **/noflag=directives**. In that case, no message is reported, but the .TRANSFER directive is ignored.

On VAX systems, you can establish universal entry points for relocatable code in shareable images by explicitly defining transfer vectors in VAX MACRO source code. On Alpha systems, you establish them by declaring the symbol values of universal entry points in a linker options file. The linker constructs a symbol vector table within the shareable image that allows external images to locate relocatable universal procedure entry points and storage addresses.

Recommended Change

You must remove the transfer vector from the VAX MACRO source. When linking the object file produced by the compiler, you must specify a linker options file that includes the SYMBOL_VECTOR statement. Within the SYMBOL_VECTOR statement, you must list each universally-visible, relocatable symbol (procedure entry point or data address), indicating whether each is DATA or a PROCEDURE.

Note that the linker builds the symbol vector in the order in which the symbols appear in the linker options file. You must retain this symbol order over the course of later builds of the shareable images. That is, you can add entries to the end of the symbol list or remove entries, but ongoing entries must keep the same ordinal position in the list. For more information about transfer vectors, refer to the *OpenVMS Linker Utility Manual*.

3.8 Arithmetic Exceptions

On Alpha systems, the treatment of arithmetic exceptions is different from and incompatible with that of VAX systems. Exception handlers designed for a VAX system that field arithmetic exceptions will be unable to match the expected signal names with those actually signaled on Alpha systems.

On VAX systems, the architecture ensures that arithmetic exceptions are reported synchronously, whereas on Alpha systems, arithmetic exceptions are reported asynchronously. On a VAX system, a VAX arithmetic instruction that causes an exception (such as an overflow) enters any exception handlers immediately and no subsequent instructions are executed. The program counter (PC) reported to the exception handler is that of the failing arithmetic instruction. This allows application programs, for example, to resume the main sequence, with the failing operation being emulated or replaced by some equivalent or alternate set of operations.

On Alpha systems, a number of instructions (including branches and jumps) can execute beyond that which caused the exception. These instructions may overwrite the original operands used by the failing instruction, therefore causing information integral to interpreting or rectifying the exception to be lost. The PC reported to the exception handler is not that of the failing instruction, but rather is that of some subsequent instruction. When the exception is reported to an application's exception handler, it may be impossible for the handler to fix up the input data and restart the instruction.

Because of this fundamental difference in arithmetic exception reporting, the OpenVMS Alpha operating system defines a new, single condition code, SS$_HPARITH, to indicate all of the arithmetic exceptions. For information about the SS$_HPARITH exception signal array, see *Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications*.[1]

[1] This manual has been archived but is available on the OpenVMS Documentation CD–ROM.

Recommended Change

If the condition handling routine in your application only counts the number of arithmetic exceptions that occurred, or aborts when an arithmetic exception occurs, it does not matter that the exception is delivered asynchronously on Alpha. These condition handling routines require only the addition of a test for the SS$_HPARITH condition code. The VAX arithmetic exceptions will never be returned on Alpha (except from translated VAX images). If your application attempts to restart the operation that caused the exception, you must either rewrite your code or use a compiler qualifier or directive that ensures the exact reporting of arithmetic exceptions. Note, however, that the compiler must drain the instruction pipeline after using each arithmetic instruction to provide this function, which severely affects performance.

The EVAX_TRAPB built-in can be used to force any preceding traps to be signaled. Because of the performance impact of this built-in, it should be used only after arithmetic instructions you need to isolate.

3.9 Page Size

A set of macros has been developed which can be used to standardize the way page size dependencies are coded and to make future changes easier. These are discussed in Appendix D.

3.10 Locking Pages into a Working Set

Pages are commonly locked down in the following ways:

- At image initialization, locking a section of code for the life of the image. This method uses the $LKWSET system service call to lock the pages in memory. The $LKWSET call is often in a different module from the code to be locked.
- On the fly, locking down a small section of code for a specific operation. This method often uses the *poor programmer's lockdown* method of raising IPL and specifying the IPL as a data location at the end of the code to be locked.

On an Alpha system, not only must the code pages be locked in memory, but the code's linkage section must also be locked. Because of this and other constraints, the LOCK_SYSTEM_PAGES, UNLOCK_SYSTEM_PAGES, PMLREQ, and PMLEND macros do not exist on Alpha systems.

Recommended Change

To minimize code changes, both cases are accommodated by means of several new macros.

For lockdown done at image initialization time, three macros are provided: two as *begin* and *end* markers to delimit code to be locked, and an additional initialization macro to create the descriptors and issue the $LKWSET calls.

To lock and unlock code where poor programmer's lockdown or other on-the-fly lockdown was done, two other macros are provided. For architectural reasons discussed in this section, these macros also use the $LKWSET and the $ULWSET system service calls to lock and unlock pages.

These macros reside in LIB.MLB, since anyone needing to lock pages into the working set should already be executing privileged code.

Note

These two methods (lockdown done at image initialization time and lockdown done on the fly) *cannot* be combined in one image. The $ULWSET service issued by the on-the-fly lockdown will also unlock the section that was locked at initialization time. You must choose one method or the other for the entire image. Be particularly careful in images that contain modules compiled from different source languages.

How the Alpha Architecture Affects Locking Pages

Locking pages into a working set is much more complicated on an Alpha system than it is on a VAX system, for the following reasons:

1. It is not sufficient to lock the code in memory. Since the code will make references to its linkage section for various values and routine addresses, the linkage section must also be locked.
2. Due to the optimization that is done by the compiler, it is not sufficient to put labels before and after the code to be locked—the compiler may move some of the code between the labels to an area outside of their scope.
3. The common Poor Programmer's method of raising IPL and specifying the IPL as a data location at the end of the code to be locked does not work for these reasons as well as for the following reasons:
   - The sequence requires the execution of multiple instructions, making it impossible to reference the IPL and raise IPL atomically
   - Alpha compiler technology prevents code and data, for example, the longword containing the IPL, from being in the same program section

The only way to lock pages into the working set on Alpha is to call the system service $LKWSET.

Using Psects to Delineate Code

The macros provided here use psects to enclose the section of code to be locked. The macros create three psects, name them sequentially, and use them in the following way:

- The beginning and ending labels of the code are defined in the first and last psects, respectively
- The code itself is put in the middle psect

Provided that the attributes of all psects are the same, the linker will place them sequentially in the image. The same thing must be done for the linkage section, which requires a second $LKWSET call.

Since all code to be locked anywhere in the image goes into the same psect, this method of using psects has the side effect of locking all lockable code when any requester locks any section. This is probably advantageous; Alpha pages are at least 8KB and most requesters are locking a pagelet or less, so most of the time all of the code to be locked will fit on a single page.

Note

If the code is being locked because the IPL will be raised above 2, where page faults cannot occur, make sure that the delimited code does not call run-time library routines or other procedures. The VAX MACRO compiler generates calls to routines to emulate certain VAX instructions. An image that uses these macros must link against the system base image so that references to these routines are resolved by code in a nonpageable executive image.

Image Initalization-Time Lockdown

For image initialization-time lockdown, three macros are used:

- $LOCKED_PAGE_START
- $LOCKED_PAGE_END
- $LOCK_PAGE_INIT

The macros $LOCKED_PAGE_START and $LOCKED_PAGE_END mark the beginning and end of a code segment which may be locked. The code delineated by these macros must contain complete routines—execution cannot fall through either macro, nor can the locked code be branched into or out of. Any attempt to branch into or out of the locked code section, or to fall through the macros will be flagged by the compiler with the following error message:

```
%AMAC-E-MULTLKSEC, Routines which share code must use the same linkage psect.
```

$LOCKED_PAGE_END has an optional parameter, LINK_SECT, which is used to specify the linkage psect to return to after the routine is executed. It is only used if the linkage psect in effect when the $LOCKED_PAGE_START macro was executed was not the default linkage psect, $LINKAGE.

The macro $LOCK_PAGE_INIT must be executed in the initialization routines of an image which is using $LOCKED_PAGE_START and $LOCKED_PAGE_END to delineate areas to be locked. It creates the necessary psects and issues the $LKWSET calls to lock the code and linkage sections into the working set. R0 and R1 are destroyed by this macro.

$LOCK_PAGE_INIT has an optional parameter, ERROR, which is an error address to which to branch if one of the $LKWSET calls fail. If this address is reached, R0 reflects the status of the failed call, and R1 contains 0 if the call to lock the code failed, or 1 if that call succeeded but the call to lock the linkage section failed.

Note that since psects are used to identify code to be locked, the $LOCK_PAGE_INIT macro need not be in the same module as the code delineated by the $LOCKED_PAGE_START and $LOCKED_PAGE_END macros. The invocation of $LOCK_PAGE_INIT locks all delineated code in the entire image.

Table 3–1 shows the code changes necessary for using these macros. The delineating labels are replaced by the $LOCKED_PAGE_START and $LOCKED_PAGE_END macros. The descriptor is eliminated, and the $LKWSET call in the initialization code is replaced by $LOCK_PAGE_INIT.

Table 3–1 Image Initialization-Time Lockdown

Code Section	On VAX Systems	On Alpha Systems

Data declaration	`LOCK_DESCRIPTOR:`
	`     .ADDRESS   LOCK_START`
	`     .ADDRESS   LOCK_END`
		Nothing. Eliminate the descriptor altogether.

Initialization	`     $LKWSET_S   LOCK_DESCRIPTOR`
	`     BLBC        R0,ERROR`
		`     $LOCK_PAGE_INIT   ERROR`

Main code	`LOCK_START:`
	`Routine_A:`
	`       .`
	`       .`
	`       .`
	`     RSB`
	`LOCK_END:`
		`     $LOCKED_PAGE_START`
		`Routine_A:`
		`       .`
		`       .`
		`       .`
		`     RSB`
		`     $LOCKED_PAGE_END`

Locking Code Written in Other Languages

Code written in other programming languages can also be locked down by using the $LOCK_PAGE_INIT macro in a VAX MACRO module. Any code in any module written in any language will be locked by this macro if the psect $LOCK_PAGE_2 is used for the generated code and the psect $LOCK_LINKAGE_2 is used for the generated linkage section.

On-the-Fly Lockdown

For on-the-fly lockdown, $LOCK_PAGE and $UNLOCK_PAGE, respectively, mark the beginning and end of a section of code to be locked. The marked code becomes a separate routine in the locked psect, where all code locked anywhere in the image is placed.

$LOCK_PAGE locks the pages and linkage section of the locked routine into the working set and JSRs to it. This macro is placed inline in executable code. All code between this macro and the matching $UNLOCK_PAGE macro is included in the locked routine and is locked down.

$UNLOCK_PAGE returns from the locked routine and then unlocks the pages and linkage section from the working set. The macro is placed inline in executable code at some point after a $LOCK_PAGE macro.

$LOCK_PAGE and $UNLOCK_PAGE both have an optional parameter, ERROR, which is an error address to which to branch if the $LKWSET or $ULWSET calls fail. $UNLOCK_PAGE has a second optional parameter, LINK_SECT. LINK_SECT is a linkage psect to which to return if the linkage psect in effect when the $LOCK_PAGE macro was executed was not the default linkage psect, $LINKAGE.

All registers are preserved by both macros unless the error address parameter is present and one of the calls fail, in which case R0 reflects the status of the failed call. R1 then contains 0 if the call to lock or unlock the code failed, and 1 if that call succeeded but the call to lock or unlock the linkage section failed.

Control must enter the code through the $LOCK_PAGE macro, and must leave through the $UNLOCK_PAGE macro. The local symbol block that is in effect when the $LOCK_PAGE macro is executed is restored when the $UNLOCK_PAGE macro is executed, but since the locked code becomes a separate routine, the locked code itself is a separate local symbol block. Even if named symbols are used, branches into or out of the locked code section are not allowed, and will be flagged by the compiler with the following error:

```
%AMAC-E-MULTLKSEC, Routines which share code must use the same linkage psect.
```

Note that since the locked code is made into a separate routine, any references to local stack storage within the routine will have to be changed, as the stack context is no longer the same.

Note

Because on-the-fly lockdown requires the overhead of four system service calls plus an extra subroutine call every time it is executed, it is recommended that this be changed to initialization-time lockdown if the lockdown is done for any performance-critical code. If other routines in the image use initialization-time lockdown, then you must change the on-the-fly lockdown to initialization-time lockdown.

Table 3–2 shows the code changes required to use these macros for on-the-fly lockdown. Note that the $UNLOCK_PAGE macro precedes the RSB, so that it is executed. Any status being passed by the routine in R0 and R1 remains intact because $UNLOCK_PAGE preserves these registers.

Table 3–2 On-the-Fly Lockdown

Code Section	On VAX Systems	On Alpha Systems
Main code		

	`Routine_A:`
	`      .`
	`      .`
	`      .`
	`  SETIPL    100$`
	`      .`
	`      .`
	`      .`
	`  RSB`
	`100$: .LONG   IPL$SYNCH`
		`Routine_A:`
		`     .JSB_ENTRY`
		`       .`
		`       .`
		`       .`
		`     $LOCK_PAGE`
		`       .`
		`       .`
		`       .`
		`     $UNLOCK_PAGE`
		`     RSB`

Table 3–3 shows the same original code and the changes necessary for initialization-time lockdown.

Table 3–3 Image Initialization-Time Lockdown with the Same Code

Code Section	On VAX Systems	On Alpha Systems
Initialization		
	Nothing.	`$LOCK_PAGE_INIT`
Main code		
	`Routine_A:`	`$LOCKED_PAGE_START`
	`.`	`Routine_A:`
	`.`	`.JSB_ENTRY`
	`.`	`.`
	`SETIPL     100$`	`.`
	`.`	`.`
	`.`	`RSB`
	`.`	`$LOCKED_PAGE_END`
	`RSB`	
	`100$: .LONG    IPL$SYNCH`	

3.11 Synchronization

The following statements and recommendations regarding synchronization are relevant to the porting of code from VAX systems to Alpha systems:

1. Code that issues longword operations to aligned longwords in memory continues to work on Alpha systems without additional synchronization required. This is architecturally guaranteed.

   The Alpha architecture extends this guarantee to include quadword operations to aligned quadwords in memory. However, this is not backwards-compatible to VAX systems. Only Alpha code can depend upon this feature.

2. Interlocked instructions (BBSSI, BBCCI, and ADAWI) still work. However, keep the following in mind when you use them:

   a. When compiling these instructions, the MACRO-32 compiler provides memory barrier functionality implicitly.

   b. These instructions assume a byte granularity environment. If the data segment on which these instructions operate can be concurrently written by different threads, you may need to impose additional synchronization of the data segment using the MACRO-32 compiler's PRESERVE feature.

   c. Another way to address the byte granularity problem and achieve greater performance at the same time is to restructure the data segments to be unpacked. That is, the bit that is changed by BBSSI or BBCCI, or the word that is modified by ADAWI, should reside in a longword where the other portions of the longword are not modified by an independent and concurrent instruction thread.

      Further separation of the data in question, such that independent and concurrent access to any location in the aligned 128-byte lock range that contains the data is not occurring, will result in additional performance gains on many Alpha implementations of the Load-locked/Store-conditional instructions.

3. The VAX interlocked queue instructions work unchanged on Alpha systems and result in the PALcode equivalents being called which incorporate the necessary interlocks and memory barriers.

Note that the noninterlocked queue instructions are also compiled to their PALcode equivalents and that they are still atomic on a single processor.

4. The VAX synchronization tools work unchanged on Alpha. All of the following mechanisms use interlocked instructions directly or indirectly for synchronization. The interlocked instructions that are used provide memory barriers transparently.
   - Event Flags—all of the system services that manipulate them
   - Spin locks—all of the acquisition and release operators (LOCK and UNLOCK, FORKLOCK and FORKUNLOCK, DEVICELOCK and DEVICEUNLOCK)
   - Mutexes—protected by spin locks

   Note

   This synchronization guarantee is only true for multiprocessing systems. The uniprocessing version of spin locks does not use interlocked instructions. As a result, memory barriers are not provided in uniprocessor spin lock, mutex, and lock manager synchronization.

   - Lock Manager—protected by spin locks
5. Regarding ASTs, concurrent threads and atomicity, one must either redesign the code or force atomicity using features provided by the compilers. The MACRO-32 compiler provides the PRESERVE feature.
6. Code that modifies exception handlers may require changes if it is possible for an outstanding arithmetic trap or a machine abort or both to occur asynchronously. The TRAPB and DRAINA instructions provide the synchronization mechanisms that are required. If you want to force synchronization when changing handlers, you must manually add these to your program as shown in the following example:

```
ADDL3 R1, R2, 4(R3)      ; Save total
EVAX_TRAPB                 ; Insure any arithmetic traps handled by
                           ; existing handler
MOVAB   HANDLER2, 0(FP)    ; Enable new condition handler
```

7. When writing OpenVMS Alpha assembly language code, make sure that you understand the read/write ordering of the Alpha architecture. Encode MB instructions where necessary.

4 Improving the Performance of Ported Code

This chapter describes how you can improve the performance of your ported code. The topics described in this chapter follow:

- Aligning data (Section 4.1)
- Code flow and branch prediction (Section 4.2)
- Code optimization (Section 4.3)
- Common-based referencing (Section 4.4)

4.1 Aligning Data

An unaligned data reference will work but will be slow on OpenVMS Alpha, because the system must take an unaligned address fault to complete the unaligned reference. If it is known that a data reference is unaligned, the compiler can generate unaligned quadword loads and masks to manually extract the data. This is slower than an aligned load but much faster than taking an alignment fault. Global data labels that are not longword or quadword aligned are flagged with information-level messages.

In addition, unaligned memory modification references *cannot* be made atomic with /PRESERVE=ATOMICITY or .PRESERVE ATOMICITY. If this is attempted, it will cause a fatal reserved operand fault.

4.1.1 Alignment Assumptions

By default, the compiler assumes the following:

- Addresses in registers used as base pointers are longword aligned at routine entry
- External references are longword aligned
- Addresses that resulted from certain types of instructions, such as DIVL, are assumed unaligned

Every time a register is changed, the compiler determines whether the base address in the register is still aligned. If the register and specified offset result in an aligned address, the compiler uses an aligned load or store for a memory reference. The compiler attempts to track register usage in terms of whether the base address remains aligned. When a stored memory address is loaded, for instance, MOVL 4(R1),R0, or used indirectly for instance, MOVL@4(R1),R0, the compiler assumes the resulting address is aligned.

For quadword memory references such as MOVQ instructions, the compiler assumes the base address is quadword aligned, *unless* it has determined by means of its register tracking code that the address may not be *longword* aligned. In other words, quadword register alignment is *not* tracked—only longword alignment.

Quadword references in Alpha built-ins, such as those in the following example, will be in new code, where alignment should be correct. Therefore all memory references in the following example will use aligned quadword load/stores:

```
EVAX_LDQ  R1, (R2)
EVAX_ADDQ  (R1), #1, (R3)
```

If an Alpha built-in (other than EVAX_LDQU or EVAX_STQU) is used on an address that is not quadword aligned, an alignment fault will occur at run time.

4.1.2 Directives and Qualifier for Changing Alignment Assumptions

The compiler provides two directives and one qualifier for changing the compiler's alignment assumptions. Both directives enable the compiler to produce more efficient code. The .SET_REGISTERS directive allows you to specify whether a register is aligned or unaligned. This directive should be used when the result of an operation is the reverse of what the compiler expects. It also allows you to declare registers that the compiler would not otherwise detect as input or output registers.

The .SYMBOL_ALIGNMENT directive allows you to specify the alignment of any memory reference that uses a symbolic offset. This directive should be used when you know the data will be aligned for every use of the symbolic offset.

These directives are described in detail in Appendix B. The examples in each description show how to use them.

The /UNALIGN qualifier to the MACRO/MIGRATION command tells the compiler to assume unaligned all the time for all register-based memory references rather than try to track the alignment. This does not affect stack-based or static references where the compiler knows the alignment.

This qualifier is described in detail in Appendix A.

4.1.3 Precedence of Alignment Controls

The order of precedence of the compiler's alignment controls, from strongest (.SYMBOL_ALIGNMENT) to weakest (built-in assumptions and tracking mechanisms), follows:

1. .SYMBOL_ALIGNMENT directive
2. .SET_REGISTER directive
3. /UNALIGN qualifier
4. Built-in assumptions and tracking mechanisms

4.1.4 Recommendations for Aligning Data

The following recommendations are provided for aligning data:

- If references to the data must be made atomic with /PRESERVE=ATOMICITY or .PRESERVE ATOMICITY, the data *must* be aligned.
- Do not fix alignment problems in public interfaces; this could break existing programs.

- For data in internal or privileged interfaces, do not automatically make changes to improve data alignment. You should consider the frequency with which the data structure is accessed, the amount of work involved in realigning the structure, and the risk that things might go wrong. In judging the amount of work involved, make sure you *know* all accesses to the data, do not just guess. If you own all accesses in the code for which you are responsible and if you are making changes in the module (or modules) anyway, then it is safe to fix the alignment problem.
- Do not routinely unpack byte and word data into longwords or quadwords. The time to do this is when you are fixing an alignment problem (word not on word boundary), subject to the aforementioned cautions and constraints, or if you know the data granularity is a problem.
- If you do not own all the accesses to the data, there still may be circumstances under which fixing alignment is appropriate. If the data is frequently accessed, if performance is a real issue, and if you must unavoidably scramble the data structure anyway, it makes sense to align the structure at the same time.

  It is important that you notify other programmers whose code may be affected. Do not assume in such cases that all related modules will recompile or that program documentation will help others detect errant data cell separation assumptions. Always assume that changes like this will reveal irregular programming practices and not go smoothly.

4.2 Code Flow and Branch Prediction

The Alpha architecture is pipelined, which means that before completing the current instruction, it starts to execute several instructions beyond it. By tailoring the code to keep the pipeline filled, you can make the code run significantly faster.

On each conditional branch, the Alpha architecture attempts to predict whether or not the branch is taken so that it can correctly fill the instruction pipeline with the next instruction to be executed. The architecture predicts that forward conditional branches will not be taken and backward conditional branches will be taken. A mispredicted branch costs extra time because the pipeline must be flushed, and, in addition, the instruction at the branch destination may not be in the instruction cache.

The compiler tries to follow the flow of the VAX MACRO code to generate Alpha code that has the most common code path in a contiguous block, to allow the pipelined Alpha architecture to process the code with the greatest efficiency. However, in some situations, the compiler's default rules do not generate the most efficient code. In performance sensitive code sections, you can often improve the efficiency of the generated code by giving the compiler information about which code paths will most likely be taken.

4.2.1 Default Code Flow and Branch Prediction

Generally, the compiler generates Alpha code that follows unconditional VAX MACRO branches and falls through conditional VAX MACRO branches unless it is directed otherwise. For example, consider the following VAX MACRO code sequence:

```
        (Code block A)
        BLBS    R0,10$
        (Code block B)
10$:
        (Code block C)
        BRB     30$
20$:
        .
        (Code block D)
        .
30$:
        (Code block E)
```

The Alpha code generated for this sequence looks like the following:

```
        (Code block A)
        BLBS    R0,10$
        (Code block B)
10$:
        (Code block C)
30$:
        (Code block E)
```

Note that the compiler fell through the BLBS instruction, continuing with the instructions immediately following the BLBS. At the BRB instruction, it did not generate a branch instruction at all but followed the Alpha code generated from Code block C with the Alpha code generated from Code block E, at the branch destination. Code from Code block D at label 20$ will be generated at a later point in the routine. If there is no branch to the label 20$, the compiler will report the following informational message and will not generate Alpha code for Code block D:

```
UNRCHCODE, unreachable code
```

In most cases, this algorithm produces Alpha code that matches the assumptions of the architecture:

- If a conditional branch is backward in the VAX MACRO code, then the destination likely has been generated already in the Alpha code, and so the generated branch will also be backward.
- If the conditional branch is forward in the VAX MACRO code, then the destination will likely not have been generated yet in the Alpha code, and so the generated branch will also be forward.

However, because the compiler follows unconditional branches, the destination of a backward VAX MACRO branch may not have been generated yet. In this case, a conditional branch that was backward in the VAX MACRO source code may become a forward branch in the generated Alpha code. See Section 4.2.5 for a further discussion and resolution of this problem.

There are some cases where the compiler may assume that a forward branch *is* taken. For example, consider the following common VAX MACRO coding practice:

```
       JSB   XYZ            ;Call a routine
       BLBS  R0,10$         ;Branch to continue on success
       BRW   ERROR          ;Destination too far for byte offset
10$:
```

In this case, and any case where the inline code following the branch is only a few lines and does not rejoin the code flow at the branch destination, the forward branch is considered taken. This eliminates the delay that occurs on Alpha systems for a mispredicted branch. The compiler will automatically change the sense of the branch, and will move the code between the branch and the label out of line to a point beyond the normal exit of the routine. For this example it would generate the following code:

```
        JSR     XYZ
        BLBC    $L1
10$:
        .
        .
        .
        (routine exit)

$L1:    BRW     ERROR
```

4.2.2 Changing the Compiler's Branch Prediction

The compiler provides two directives, .BRANCH_LIKELY and .BRANCH_ UNLIKELY, to change its assumptions about branch prediction. The directive .BRANCH_LIKELY is for use with forward conditional branches when the probability of the branch is large, say 75 percent or more. The directive .BRANCH_UNLIKELY is for use with backward conditional branches when the probability of the branch is less than 25 percent.

These directives should only be used in performance-sensitive code. Furthermore, you should be more cautious when adding .BRANCH_UNLIKELY, because it introduces an additional branch indirection for the case when the branch is actually taken. That is, the branch is changed to a forward branch to a branch instruction, which in turn branches to the original branch target.

There is no directive to tell the compiler not to follow an unconditional branch. However, if you want the compiler to generate code that does not follow the branch, you can change the unconditional branch to be a conditional branch that you know will always be taken. For example, if you know that in the current code section R3 always contains the address of a data structure, you could change a BRB instruction to a TSTL R3 followed by a BNEQ instruction. This branch will always be taken, but the compiler will fall through and continue code generation with the next instruction. This will always cause a mispredicted branch when executed, but may be useful in some situations.

4.2.3 How to Use .BRANCH_LIKELY

If your code has forward conditional branches that you know will most likely be taken, you can instruct the compiler to generate code using that assumption by inserting the directive .BRANCH_LIKELY immediately before the branch instruction. For example:

```
MOVL    (R0),R1          ; Get structure
.BRANCH_LIKELY
BNEQ    10$              ; Structure exists
  .
 (Code to deal with missing structure, which is too large for
  the compiler to automatically change the branch prediction)
  .
10$:
```

The compiler will follow the branch and will modify the code flow as described in the previous example, moving all the code that deals with the missing structure out of line to the end of the module.

4.2.4 How to Use .BRANCH_UNLIKELY

If your code has backward conditional branches which you know will most likely *not* be taken, you can instruct the compiler to generate code using that assumption by inserting the directive .BRANCH_UNLIKELY immediately before the branch instruction. For example:

```
        MOVL    #QUEUE,R0               ;Get queue header
10$:    MOVL    (R0),R0                 ;Get entry from queue
        BEQL    20$                     ;Forward branch assumed unlikely
        .                               ;by default
        .                               ;Process queue entry
        .
        TSTL    (R0)                    ;More than one entry (known to be
        .BRANCH_UNLIKELY                ;unlikely)
        BNEQ    10$                     ;This branch made into forward
20$:                                    ;conditional branch
```

The .BRANCH_UNLIKELY directive is used here because the Alpha hardware would predict a backward branch to 10$ as likely to be taken. The programmer knows it is a rare case, so the directive is used to change the branch to a forward branch, which is predicted not taken.

There is an unconditional branch instruction at the forward branch destination which branches back to the original destination. Again, this code fragment is moved to a point beyond the normal routine exit point. The code that would be generated by the previous VAX MACRO code follows:

```
        LDQ     R0, 48(R27)             ;Get address of QUEUE from linkage sect.
10$:    LDL     R0, (R0)                ;Get entry from QUEUE
        BEQ     R0, 20$
        .
        .                               ;Process queue entry
        .
        LDL     R22, (R0)               ;Load temporary register with (R0)
        BNE     R22,$L1                 ;Conditional forward branch predicted
20$:                                    ;not taken by Alpha hardware
        .
        .
        .
        (routine exit)

$L1:    BR      10$                     ;Branch to original destination
```

4.2.5 Forward Jumps into Loops

Because of the way that the compiler follows the code flow, a particular case that may not compile well is a forward unconditional branch into a loop. The code generated for this case usually splits the loop into two widely separated pieces. For example, consider the following macro coding construct:

```
        (Allocate a data block and set up initial pointer)
        BRB     20$
10$:    (Move block pointer to next section to be moved)

20$:    (Move block of data)
        (Test - is there more to move?)
        (Yes, branch to 10$)

        (Remainder of routine)
```

The macro compiler will follow the BRB instruction when generating the code flow and will then fall through the subsequent conditional branch to 10$. However, because the code at 10$ was skipped over by the BRB instruction, it will not be generated until after the end of the routine. This will convert the conditional branch into a forward branch instead of a backward branch. The generated code layout will look like the following:

```
        (Allocate a data block and set up initial pointer)
20$:    (Move block of data)
        (Test - is there more to move?)
        (Yes, branch to 10$)
        .
        .
        (Remainder of routine)
        (Routine exit)
        .
        .
10$:    (Move block pointer to next section to be moved)
        BRB     20$
```

This results in the loop being very slow because the branch to 10$ is always predicted not taken, and the code flow has to keep going back and forth between the two locations. This situation can be fixed by inserting a .BRANCH_LIKELY directive before the conditional branch back to 10$. This will result in the following code flow:

```
        (Allocate a data block and set up initial pointer)
20$:    (Move block of data)
        (Test - is there more to move?)
 (No, branch to $L1)
10$:    (Move block pointer to next section to be moved)
        BRB     20$
$L1:
        (Remainder of routine)
```

4.3 Code Optimization

The MACRO-32 compiler performs several optimizations on the generated code. It performs all of them by default except VAXREGS. You can change these default values with the /OPTIMIZE switch on the command line. The valid options are:

- ADDRESSES

  The compiler recognizes that the same address is referenced multiple times, and only loads the address once for use by multiple references.

- REFERENCES

  The compiler recognizes that the same data value is referenced multiple times, and only loads the data once for use by multiple references, subject to restrictions to ensure that the data being used is not stale.

- PEEPHOLE

  The compiler identifies instruction sequences that can be identically performed by smaller instruction sequences, and replaces the longer sequences with the shorter ones.

- SCHEDULING

  The compiler uses its knowledge of the nature of the multiple instruction issue ability of the Alpha architecture to reschedule the code for optimum performance.

- VAXREGS

  By default, the registers from R13 through R28 may be used as temporary scratch registers by the compiler if they are not used in the source code. When VAXREGS is specified, the compiler may also use any of the VAX register set (R0 through R12) that are not explicitly used by the MACRO source code. VAX registers used in this way will be restored to their original values at routine exit unless declared SCRATCH.

Note

Debugging is simplified if you specify /NOOPTIMIZE, because the optimizations include relocating and rescheduling code. For more information, see Section 2.12.1.

4.3.1 Using the VAXREGS Optimization

To use the VAXREGS optimization, you must ensure that all routines correctly declare their register usage in their .CALL_ENTRY, .JSB_ENTRY, or .JSB32_ENTRY routine declarations. In addition, you must identify any VAX registers that are required or modified by any routines that are called. By default, the compiler assumes that no VAX registers are required as input to any called routine, and that all VAX registers except R0 and R1 are preserved across the call. To declare this usage, use the READ and WRITTEN qualifiers to the compiler directive .SET_REGISTERS. For example:

```
.SET_REGISTERS  READ=<R3,R4>, WRITTEN=R5
JSB     DO_SOMETHING_USEFUL
```

In this example, the compiler will assume that R3 and R4 are required inputs to the routine DO_SOMETHING_USEFUL, and that R5 is overwritten by the routine. The register usage can be determined by using the **input** mask of DO_SOMETHING_USEFUL as the READ qualifier, and the combined **output** and **scratch** masks as the WRITE qualifier.

Note

Using the VAXREGS qualifier without correct register declaration for both routine entry points and routine calls will produce incorrect code.

4.4 Common-Based Referencing

On an Alpha system, references to data cells generally require two memory references—one reference to load the data cell address from the linkage section and another reference to the data cell itself. If several data cells are located in proximity to one other, and the ADDRESSES optimization is used, the compiler can load a register with a common base address and then reference the individual data cells as offsets from that base address. This eliminates the load of each individual data cell address and is known as *common-based referencing*.

The compiler performs this optimization automatically for local data psects when the ADDRESSES optimization is turned on. The compiler generates symbols of the form $PSECT_BASE*n* to use as the base of a local psect.

To use common-based referencing for external data psects, you must create a prefix file which defines symbols as offsets from a common base. The prefix file cannot be used when assembling the module for OpenVMS VAX because the VAX MACRO assembler does not allow symbols to be defined as offsets from external symbols.

4.4.1 Creating a Prefix File for Common-Based Referencing

The following example illustrates the benefits of creating a prefix file to use common-based referencing. It shows:

- Code generated without the use of a prefix file
- How to create a prefix file
- Code generated with the use of a prefix file

Consider the following simple code section (CODE.MAR), which refers to data cells in another module (DATA.MAR):

```
Module DATA.MAR:

        .PSECT  DATA    NOEXE
BASE::
A::     .LONG   1
B::     .LONG   2
C::     .LONG   3
D::     .LONG   4
        .END
```

```
Module CODE.MAR:

        .PSECT CODE     NOWRT

E::     .CALL_ENTRY
        MOVL    A,R1
        MOVL    B,R2
        MOVL    C,R3
        MOVL    D,R4
        RET
        .END
```

When compiling CODE.MAR without using common-based referencing, the following code is generated:

In the linkage section:

```
        .ADDRESS        A
        .ADDRESS        B
        .ADDRESS        C
        .ADDRESS        D
```

In the code section (not including the prologue/epilogue code):

```
        LDQ     R28, 40(R27)            ;Load address of A from linkage section
        LDQ     R26, 48(R27)            ;Load address of B from linkage section
        LDQ     R25, 56(R27)            ;Load address of C from linkage section
        LDQ     R24, 64(R27)            ;Load address of D from linkage section
        LDL     R1, (R28)               ;Load value of A
        LDL     R2, (R26)               ;Load value of B
        LDL     R3, (R25)               ;Load value of C
        LDL     R4, (R24)               ;Load value of D
```

By creating a prefix file that defines external data cells as offsets from a common base address, you can cause the compiler to use common-based referencing for external references. A prefix file for this example, which defines A, B, C, and D in terms of BASE, follows:

```
A = BASE+0
B = BASE+4
C = BASE+8
D = BASE+12
```

When compiling CODE.MAR using this prefix file and the ADDRESSES optimization, the following code is generated:

In the linkage section:

```
.ADDRESS        BASE            ;Base of data psect
```

In the code section (not including the prologue/epilogue code):

```
LDQ     R16, 40(R27)            ;Load address of BASE from linkage section
LDL     R1, (R16)               ;Load value of A
LDL     R2, 4(R16)              ;Load value of B
LDL     R3, 8(R16)              ;Load value of C
LDL     R4, 12(R16)             ;Load value of D
```

In this example, common-based referencing shrinks the size of both the code and the linkage sections and eliminates three memory references. This method of creating a prefix file to enable common-based referencing of external data cells can be useful if you have one large, separate module that defines a data area used by many modules.

5 MACRO-32 Programming Support for 64-Bit Addressing

This chapter describes the 64-bit addressing support provided by the MACRO-32 compiler and associated components. The changes are primarily for argument passing and receiving and for address computations.

5.1 Guidelines for 64-Bit Addressing

The following guidelines pertain to using 64-bit addressing in VAX MACRO code that is compiled for OpenVMS Alpha:

- Limit its use to code that you have ported to OpenVMS Alpha.

  For any new development on OpenVMS Alpha, Compaq recommends the use of higher-level languages.

- Make 64-bit addressing explicit in your code.

  The 64-bit addressing qualifiers, macros, directives, and built-ins produce code that is more reliable and easier to maintain.

5.2 New and Changed Components for 64-Bit Addressing

The new and changed components that provide MACRO-32 programming support for 64-bit addressing are shown in Table 5–1.

Table 5–1 New and Changed Components for 64-Bit Addressing

Component	Description
$SETUP_CALL64	New macro that initializes the call sequence.
$PUSH_ARG64	New macro that does the equivalent of argument pushes.
$CALL64	New macro that invokes the target routine.
$IS_32BITS	New macro for checking the sign extension of the low 32 bits of a 64-bit value.
$IS_DESC64	New macro for determining if descriptor is a 64-bit format descriptor.
QUAD=NO/YES	New parameter for page macros to support 64-bit virtual addresses.
/ENABLE=QUADWORD	The QUADWORD parameter was extended to include 64-bit address computations.

(continued on next page)

Table 5–1 (Cont.) New and Changed Components for 64-Bit Addressing

Component	Description
.CALL_ENTRY QUAD_ ARGS=TRUE | FALSE	QUAD_ARGS=TRUE | FALSE is a new parameter that indicates the presence (or absence) of quadword references to the argument list.
.ENABLE QUADWORD/.DISABLE QUADWORD	The QUADWORD parameter was extended to include 64-bit address computations.
EVAX_SEXTL	New built-in for sign extending the low 32 bits of a 64-bit value into a destination.
EVAX_CALLG_64	New built-in to support 64-bit calls with variable-size argument lists.
$RAB64 and $RAB64_STORE	New RMS macros for using buffers in 64-bit address space.

5.3 Passing 64-Bit Values

The method that you use for passing 64-bit values depends on whether the size of the argument list is fixed or variable. These methods are described in the following sections.

5.3.1 Calls with a Fixed-Size Argument List

For calls with a fixed-size argument list, use the new macros shown in Table 5–2.

Table 5–2 Passing 64-Bit Values with a Fixed-Size Argument List

Step	Use...
1. Initialize the call sequence	$SETUP_CALL64
2. “Push” the call arguments	$PUSH_ARG64
3. Invoke the target routine	$CALL64

An example of using these macros follows. Note that the arguments are pushed in reverse order, which is the same way a 32-bit PUSHL instruction is used.

```
MOVL           8(AP), R5        ; fetch a longword to be passed
$SETUP_CALL64  3                ; Specify three arguments in call
$PUSH_ARG64    8(R0)            ; Push argument #3
$PUSH_ARG64    R5               ; Push argument #2
$PUSH_ARG64    #8               ; Push argument #1
$CALL64        some_routine     ; Call the routine
```

The $SETUP_CALL64 macro initializes the state for a 64-bit call. It is required before $PUSH_ARG64 or $CALL64 can be used. If the number of arguments is greater than six, this macro creates a local JSB routine, which is invoked to perform the call. Otherwise, the argument loads and call are inline and very efficient. Note that the argument count specified in the $SETUP_CALL64 does *not* include a pound sign (#). (The standard call sequence requires octaword alignment of the stack with its arguments at the top. The JSB routine facilitates this alignment.)

The inline option can be used to force a call with greater than six arguments to be done without a local JSB routine. However, there are restrictions on its use (see Appendix E).

The $PUSH_ARG64 macro moves the argument directly to the correct argument register or stack location. It is not actually a stack push, but it is the analog of the PUSHL instructions used in a 32-bit call.

The $CALL64 macro sets up the argument count register and invokes the target routine. If a JSB routine was created, it ends the routine. It reports an error if the number of arguments pushed does not match the count specified in $SETUP_CALL64. Both $CALL64 and $PUSH_ARG64 check that $SETUP_CALL64 has been invoked prior to their use.

5.3.1.1 Usage Notes for $SETUP_CALL64, $PUSH_ARG64, and $CALL64

Keep these points in mind when using $SETUP_CALL64, $PUSH_ARG64, and $CALL64:

- The arguments are read as aligned quadwords. To pass a longword from memory, move it to a register first, and then use that register in $PUSH_ARG64, as shown in the example in Section 5.3.1. Similarly, if you know the quadword you want to pass is unaligned, move the value to a register first. Also, keep in mind that indexed operands, such as (R4)[R0], will be evaluated using quadword indexing when used in $PUSH_ARG64.
- If the number of arguments is greater than six, so that a local JSB routine is created, no SP or AP references are allowed between the $SETUP_CALL64 and $CALL64. The $PUSH_ARG64 and $CALL64 macros do report uses of these registers in operands, but they are not allowed in other instructions in this range and cannot be flagged. To pass an AP- or SP-based argument in this case, move it to a register before the $SETUP_CALL64 invocation.
- If the number of arguments is greater than six, do not rely on values in registers above R15 surviving the $SETUP_CALL64 invocation. Use a nonscratch register as a temporary register instead. For example, suppose you want to pass a value from a stack location, and the call has more than six arguments. In this case, you need to move the value to a register. Rather than using a scratch register such as R28, use a VAX register, such as R0. If all the VAX registers are in use, use R13, R14, or R15.
- It is safe to use the scratch registers above R16 within the range between the $SETUP_CALL64 and the $CALL64. However, you must be careful not to use an argument register that has already been loaded. The argument registers are loaded in downward order, from R21 through R16. So, suppose a call passes six arguments. It is not safe to use R21 after the first $PUSH_ARG64, because that has loaded R21. The $PUSH_ARG64 macro checks for operands that refer to argument registers that have already been loaded. If any are found, the compiler reports a warning. The safest approach is to use registers R22 through R28 when a temporary register is required.

Note

The $SETUP_CALL64, $PUSH_ARG64, and $CALL64 macros are intended to be used in an inline sequence. That is, you cannot branch into the middle of a $SETUP_CALL64/$PUSH_ARG64/$CALL64 sequence, nor can you branch around $PUSH_ARG64 macros or branch out of the sequence to avoid the $CALL64.

For more information about $SETUP_CALL64, $PUSH_ARG64, and $CALL64, see Appendix E.

5.3.2 Calls with a Variable-Size Argument List

For calls with a variable-size argument list, use the new EVAX_CALLG_64 built-in, as shown in the following steps:

1. Create an in-memory argument list.
2. Call a routine, passing the in-memory argument list. For example:

   ```
   EVAX_CALLG_64 (Rn), routine
   ```

The argument list in the EVAX_CALLG_64 built-in is read as a series of quadwords, beginning with a quadword argument count.

5.4 Declaring 64-Bit Arguments

You can use QUAD_ARGS=TRUE, a new .CALL_ENTRY parameter, to declare the use of quadword arguments in a routine's argument list. With the presence of the QUAD_ARGS parameter, the compiler behaves differently when a quadword reference to the argument list occurs. First, it does not force argument-list homing, which such a reference normally requires. (An argument list containing a quadword value cannot be homed because homing, by definition, packs the arguments into longword slots.) Second, **unaligned memory reference** will not be reported on these quadword references to the argument list.

Note that the actual code generated for the argument-list reference itself is not changed by the presence of the QUAD_ARGS clause, except when the reference is in a VAX quadword instruction, such as MOVQ. For the most part, QUAD_ARGS only prevents argument-list homing due to a quadword reference and suppresses needless alignment messages. This suppression applies to both EVAX_ built-ins and VAX quadword instructions such as MOVQ.

For VAX quadword instructions, the QUAD_ARGS clause causes the compiler to read the quadword argument as it does for EVAX_ built-ins—as an actual quadword. Consider the following example:

```
MOVQ    4(AP), 8(R2)
```

If the QUAD_ARGS clause is specified, MOVQ stores the entire 64 bits of argument 1 into the quadword at 8(R2). If the QUAD_ARGS clause is not specified, MOVQ stores the low longwords of arguments 1 and 2 into the quadword at 8(R2).

QUAD_ARGS also affects the code generated for deferred mode operands that are AP-based. If the effective address must be loaded from an argument in memory, it will be read as a quadword, rather than a longword, if QUAD_ARGS is in effect.

5.4.1 Usage Notes for QUAD_ARGS

Keep these points in mind when using QUAD_ARGS:

- AP-based quadword argument-list references look strange because they appear to overlap. You can improve this situation by defining symbolic names for the argument-list offsets, for example, FIRST_ARG, SECOND_ARG, and so forth. Users are encouraged to define meaningful symbolic names that describe the uses of the arguments to make the source code more readable. Alternatively, you can still use direct argument register references to refer to the first six arguments. Either way, it is useful to declare QUAD_ARGS to ensure that the argument list is not homed.

- Routines that share code must have the same setting for QUAD_ARGS. If they do not, the compiler will report a warning message.
- JSB routines cannot refer to their caller's argument list if the caller has QUAD_ARGS. References to AP within JSB routines require that the last CALL_ENTRY have its argument list homed. HOME_ARGS and QUAD_ARGS are mutually exclusive.
- QUAD_ARGS causes the $ARGn symbols, which the compiler places in the debug symbol table, to be defined as quadwords rather than longwords. These symbols allow easy access to received argument values and can be used in place of register numbers or stack offsets when debugging with the symbolic debugger.

5.5 Specifying 64-Bit Address Arithmetic

There are no explicit pointer-type declarations in MACRO-32. You can create a 64-bit pointer value in a register in a variety of ways. The most common are the EVAX_LDQ built-in for loading an address stored in memory and the MOVA*x* for getting the address of the specified operand.

After a 64-bit pointer value is in a register, an ordinary instruction will access the 64-bit address. The amount of data read from that address depends on the instruction used. Consider the following example:

```
MOVL    4(R1), R0
```

The MOVL instruction reads the longword at offset 4 from R1, regardless of whether R1 contains a 32- or 64-bit pointer.

However, certain addressing modes require the generation of arithmetic instructions to compute the effective address. For VAX compatibility, the compiler computes these as longword operations. For example, `4 + <1@33>` yields the value 4 because the shifted value exceeds 32 bits. If quadword mode is enabled, the upper bit will not be lost.

In compilers shipping with previous versions of OpenVMS Alpha, the /ENABLE=QUADWORD qualifier (and the corresponding .ENABLE QUADWORD and .DISABLE QUADWORD directives) only affected the mode in which constant expression evaluations were performed. For OpenVMS Alpha Version 7.0, these have been extended to affect address computations. They will result in addresses being computed with quadword instructions, such as S*x*ADDQ and ADDQ.

To have quadword operations used throughout a module, specify /ENABLE=QUADWORD on the command line. If you want quadword operations applied only to certain sections, use the .ENABLE QUADWORD and .DISABLE QUADWORD directives to enclose those sections.

There is no performance penalty when using /ENABLE=QUADWORD.

5.5.1 Dependence on Wrapping Behavior of Longword Operations

The compiler cannot use quadword arithmetic for all addressing computations because existing code may rely on the wrapping behavior of the 32-bit operations. That is, code may perform addressing operations that actually overflow 32 bits, knowing that the upper bits are discarded. Doing the calculation in quadword mode causes an incompatibility.

Before using /ENABLE to set quadword evaluation for an entire module, check the existing code for dependence on longword wrapping. There is no simple way to do this, but as a programming technique, it should be rare and may be called out in the code.

The following example shows the wrapping problem:

```
MOVAL   (R1)[R0], R2
```

Suppose R1 contains the value 7FFFFFFF and R0 contains 1. The MOVAL instruction generates an S4ADDL instruction. The shift and add result exceeds 32 bits, but the stored result is the low 32 bits, sign-extended.

If quadword arithmetic were used (S4ADDQ), the true quadword value would result, as shown in the following example:

```
S4ADDL  R0, R1, R2   =>  FFFFFFFF 80000003
S4ADDQ  R0, R1, R2   =>  00000000 80000003
```

The wrapping problem is not limited to indexed-mode addressing. Consider the following example:

```
MOVAB   offset(R1), R0
```

If the symbol offset is not a compile-time constant, this instruction causes a value to be read from the linkage section and added (using an ADDL instruction) to the value in R1. Changing this to ADDQ may change the result if the value exceeds 32 bits.

5.6 Sign Extending and Checking

A new built-in, EVAX_SEXTL (sign-extend longword), is available for sign extending the low 32 bits of a 64-bit value into a destination. This built-in makes explicit the sign extension of the low longword of the source into the destination.

EVAX_SEXTL takes the low 32 bits of the 64-bit value, fills the upper 32 bits with the sign extension (whatever is in bit 31 of the value) and writes the 64-bit result to the destination.

The following examples are all legal uses:

```
evax_sextl r1,r2
evax_sextl r1,(r2)
evax_sextl (r2), (r3)[r4]
```

As shown by these examples, the operands are not required to be registers.

A new macro, $IS_32BITS, is available for checking the sign extension of the low 32 bits of a 64-bit value. It is described in Appendix E.

5.7 Alpha Instruction Built-ins

The compiler supports many Alpha instructions as built-ins. Many of these built-ins (available since the compiler first shipped) can be used to operate on 64-bit quantities. The function of each built-in and its valid operands are documented in Appendix C.

5.8 Calculating Page-Size Dependent Values

A parameter, QUAD=NO/YES, for supporting 64-bit virtual addresses is available for each of the page macros, as shown in the following list:

- $BYTES_TO_PAGES
- $NEXT_PAGE
- $PAGES_TO_BYTES
- $PREVIOUS_PAGE
- $START_OF_PAGE

These macros provide a standard, architecture-independent means for calculating page-size dependent values. For more information about these macros, see Appendix D.

5.9 Creating and Using Buffers in 64-Bit Address Space

The $RAB and $RAB_STORE control block macros have been extended for creating and using data buffers in 64-bit address space. The 64-bit versions are named $RAB64 and $RAB64_STORE, respectively. The rest of the RMS interface is restricted to 32 bits at this time. For more information about $RAB64 and $RAB64_STORE, see the *OpenVMS Programming Concepts Manual, Volume I*.

5.10 Coding for Moves Longer Than 64 KB

The MACRO-32 instructions MOVC3 and MOVC5 properly handle 64-bit addresses but the moves are limited to a 64 KB length. This limitation is because MOVC3 and MOVC5 accept word-sized lengths.

For moves longer than 64 KB, use OTS$MOVE3 and OTS$MOVE5. OTS$MOVE3 and OTS$MOVE5 accept longword-sized lengths. (LIB$MOVC3 and LIB$MOVC5 have the same 64 KB length restriction as MOVC3 and MOVC5.) An example of replacing MOVC3 with OTS$MOVE3 follows.

Code using MOVC3:

```
MOVC3      BUF$W_LENGTH(R5), (R6), OUTPUT(R7)   ; Old code, word length
```

The equivalent 64-bit code with longword length:

```
$SETUP_CALL64  3              ; Specify three arguments in call
EVAX_ADDQ      R7, #OUTPUT, R7
$PUSH_ARG64    R7             ; Push destination, arg #3
$PUSH_ARG64    R6             ; Push source, arg #2
MOVL           BUF$L_LENGTH(R5), R16
$PUSH_ARG64    R16            ; Push length, arg #1
$CALL64        OTS$MOVE3

MOVL           BUF$L_LENGTH(R5), R16
EVAX_ADDQ      R6, R16, R1    ; MOVC3 returns address past source
EVAX_ADDQ      R7, R16, R3    ; MOVC3 returns address past destination
```

Because MOVC3 clears R0, R2, R4, and R5, make sure that these side effects are no longer needed.

OTS$MOVE3 and OTS$MOVE5 are documented with other LIBOTS routines in the *OpenVMS RTL General Purpose (OTS$) Manual*.

Part II

Reference

A
Compiler Qualifiers

This appendix describes the invocation format of the MACRO-32 Compiler for OpenVMS Alpha and each of its qualifiers.

MACRO/MIGRATION

Invokes the MACRO-32 Compiler for OpenVMS Alpha to compile one or more VAX MACRO assembly language source files into native OpenVMS Alpha object code.

Format

MACRO/MIGRATION   filespec[+...]

Parameters

filespec[+...]
Specifies a VAX MACRO assembly language source file to be compiled. If you specify more than one file, separate the file specifications with plus signs (+). File specifications separated by plus signs are concatenated into one input file and produce a single object file and, if indicated, a listing file.

Note

Unlike the VAX assembler, the MACRO-32 compiler does not support the creation of separate object files when the source files are separated by a comma (,).

You cannot include a wildcard character in a file specification. For each file specification, the compiler command supplies a default file type of MAR.

The compiler creates output files of one version higher than the highest version existing in the target directory.

Description

The qualifiers to the MACRO/MIGRATION command serve as either command (global) qualifiers or positional qualifiers. A **command qualifier** affects all the files specified in the command. A **positional qualifier** affects only the file that it qualifies. All MACRO/MIGRATION qualifiers except /LIBRARY are usable as either command or positional qualifiers. The /LIBRARY qualifier is a positional qualifier only.

Many of the qualifiers take one or more arguments. If you specify only one argument, you can omit the parentheses.

The compiler supports most of the standard MACRO qualifiers. Some of these qualifiers have additional options unique to the compiler and some of them are missing one or more VAX MACRO options. The compiler also supports several new qualifiers, unique to the compiler. All of these qualifiers are shown in Table A–1.

Table A–1 Compiler Qualifiers

Standard MACRO Qualifiers	New Qualifiers
/DEBUG	/FLAG
/DIAGNOSTICS	/MACHINE
/DISABLE[1]	/OPTIMIZE
/ENABLE[1]	/PRESERVE
/LIBRARY	/RETRY_COUNT
/LIST	/SYMBOLS
/OBJECT	/TIE
/SHOW	/UNALIGNED
	/WARN

[1]With additional options unique to the compiler and some VAX MACRO options missing

Qualifiers

/DEBUG=(option[,...])
/NODEBUG

Includes or excludes local symbols in the symbol table or traceback information in the object module. You can specify one or more of the following options:

Option	Description
ALL	Makes local symbols and traceback information in the object module available to the debugger. This qualifier is equivalent to /ENABLE=(DEBUG,TRACEBACK).
NONE	Makes local symbols and traceback information in the object module unavailable to the debugger. This qualifier is equivalent to /DISABLE=(DEBUG,TRACEBACK).
SYMBOLS	Makes all local symbols in the object module available and all traceback information unavailable to the debugger. This qualifier is equivalent to /ENABLE=SYMBOLS.
TRACEBACK	Makes traceback information in the object module available and local symbols unavailable to the debugger. This qualifier is equivalent to /ENABLE=TRACEBACK.

The default value for /DEBUG is ALL. The /DEBUG qualifier overrides /ENABLE=(DEBUG,TRACEBACK) or /DISABLE=(DEBUG,TRACEBACK), regardless of their order on the command line.

Note

Debugging can be simplified by specifying /NOOPTIMIZE. This qualifier prevents the movement of generated code across source line boundaries.

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

/DIAGNOSTICS[=filespec]
/NODIAGNOSTICS (default)

Creates a file containing assembler messages and diagnostic information. If you omit the file specification, the default file name is the same as the source program; the default file type is DIA.

No wildcard characters are allowed in the file specification.

The diagnostics file is reserved for use with Compaq layered products, such as the VAX Language-Sensitive Editor (LSE).

/DISABLE=(option[,...])
/NODISABLE

Provides initial settings for the compiler functions that can be controlled by the .DISABLE and .ENABLE MACRO directives.

You can specify one or more of the following functions:

Option	Description
DEBUG	Excludes local symbol table information in the object file for use with the debugger. If the /DEBUG qualifier is also specified, it overrides /DISABLE=(DEBUG,TRACEBACK) or /ENABLE=(DEBUG,TRACEBACK), regardless of their order on the command line.
FLAGGING	Deactivates compiler flagging.
GLOBAL	Disables the assumption that undefined symbols are external symbols.
OVERFLOW	Deactivates production of overflow trap code for the following opcodes: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (where x is greater than y, for example CVTLB), AOBxx, ACBL, and SOBxx.
QUADWORD	Disables support for quadword literal and address expressions.
SUPPRESSION	Prevents the listing of unreferenced symbols in the symbol table.
TRACEBACK	Disables the provision of traceback information to the debugger. If the /DEBUG qualifier is also specified, it overrides /DISABLE=(DEBUG,TRACEBACK) or /ENABLE=(DEBUG,TRACEBACK), regardless of their order on the command line.

By default, at compiler activation, FLAGGING, GLOBAL, and SUPPRESSION are enabled, and DEBUG, OVERFLOW, QUADWORD, and TRACEBACK are disabled.

The /NODISABLE qualifier has the same effect as omitting the /DISABLE qualifier. It can also be used to negate the effects of any /DISABLE qualifiers specified earlier in the command line.

Note

If /DISABLE is used two or more times in the command line, the last /DISABLE will override all previous uses of /DISABLE. The options not specified in the final /DISABLE will revert to their default values.

Furthermore, if /ENABLE and /DISABLE are used in the same command line for the same option, /DISABLE will always prevail, regardless of its position in the command line.

Workaround—If you want to disable two or more options, specify them in the following way:

```
/DISABLE=(xxxx, yyyy)
```

/ENABLE=(option[,...])
/NOENABLE

Provides initial settings for the compiler functions that can be controlled by the .DISABLE and .ENABLE MACRO directives.

You can specify one or more of the following functions:

Option	Description
DEBUG	Includes local symbol table information in the object file for use with the debugger. If the /DEBUG qualifier is also specified, it overrides /ENABLE=(DEBUG,TRACEBACK) or /DISABLE=(DEBUG,TRACEBACK), regardless of their order on the command line.
FLAGGING	Activates compiler flagging.
GLOBAL	Assumes undefined symbols are external symbols.
OVERFLOW	Activates production of overflow trap code for the following opcodes: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (where x is greater than y, for example CVTLB), AOBxx, ACBL, and SOBxx.
QUADWORD	Provides support for quadword literal and address expressions.
SUPPRESSION	Provides listing of unreferenced symbols in the symbol table.
TRACEBACK	Provides traceback information to the debugger. If the /DEBUG qualifier is also specified, it overrides /ENABLE=(DEBUG,TRACEBACK) or /DISABLE=(DEBUG,TRACEBACK), regardless of their order on the command line.

By default, at compiler activation, FLAGGING, GLOBAL, TRACEBACK, and SUPPRESSION are enabled, and DEBUG, OVERFLOW, and QUADWORD are disabled.

The /NOENABLE qualifier has the same effect as not specifying the /ENABLE qualifier. It can also be used to negate the effects of any /ENABLE qualifiers specified earlier in the command line.

Note

For every option of the /ENABLE qualifier, if /ENABLE and /DISABLE are used in the same command line for the same option, /DISABLE will always prevail, regardless of its position in the command line.

You may want to enable an option previously disabled through the use of a symbol. For example, you may have incorporated the following frequently used options into the DCL symbol MAC, as follows:

```
MAC::== MACRO/MIGRATION/NOTIE/DISABLE=FLAGGING
```

To enable FLAGGING using the symbol MAC, issue the following command:

```
$ MAC /NODISABLE/ENABLE=FLAGGING
```

/FLAG=(option[,...])
/NOFLAG

Specifies which classes of informational messages the compiler reports. The options are:

Option	Description
ALIGNMENT	Reports unaligned stack and memory references.
ALL	Enables all options.
ARGLIST	Reports that the argument list has been homed. (See Section 2.3.1.)
CODEGEN	Reports run-time code generation, such as self-modifying code. (See Section 3.2.2.)
DIRECTIVES	Reports unsupported directives.
HINTS	Reports input/output/auto-preserved register hints.
INSTRUCTIONS	Reports instructions that use absolute addresses that may compile correctly, but should be examined anyway, because the desired absolute address may be different on an Alpha computer.
JUMPS	Reports branches between routines.
NONE	Disables all options.
STACK	Reports all messages caused by user stack manipulation.

At compiler activation, flagging is enabled by default for all options except **HINTS**.

Note

Use of the /NOFLAG and /FLAG qualifiers together to activate a specific subset of cross-compiler messages does not work as expected. When used together, as in /NOFLAG/FLAG=(keyword,keyword), instead of activating only the messages specified by the keywords, all cross-compiler messages are activated. However, use of /FLAG=(none,keyword) activates only those messages specified by the keyword.

Note that specifying /NOFLAG or /FLAG=NONE does *not* disable the reporting of coding constructs that would prevent a successful compilation. The compiler continues to report code that you *must* change, such as an up-level stack reference.

/LIBRARY
/NOLIBRARY
Positional qualifier.

The associated input file to the /LIBRARY qualifier must be a macro library. The default file type is MLB. The /NOLIBRARY qualifier has the same effect as not specifying the /LIBRARY qualifier, or negates the effects of any /LIBRARY qualifiers specified earlier in the command line.

The compiler can search up to 16 libraries, one of which is always STARLET.MLB. This number applies to a particular compilation, not necessarily to a particular MACRO command. If you enter the MACRO command so that more than one source file is compiled, but the source files are compiled *separately*, you can specify up to 16 macro libraries for each separate compilation. More than one macro library in a compilation causes the libraries to be searched in reverse order of their specification.

A macro call in a source program causes the compiler to begin the following sequence of searches:

1. An initial search of the libraries specified with the .LIBRARY directive. The compiler searches these libraries in the reverse order of that in which they were declared.
2. If the macro definition is not found in any of the libraries specified with the .LIBRARY directive, a search of the libraries specified in the MACRO command line (in the reverse order in which they were specified).
3. If the macro definition is not found in any of the libraries specified in the command line, a search of STARLET.MLB.

/LIST[=filespec]
/NOLIST
Creates or omits an output listing, and optionally provides an output file specification for it. The default file type for the listing file is LIS. No wildcard characters are allowed in the file specification.

An interactive MACRO command does not produce a listing file by default. The /NOLIST qualifier, present either explicitly or by default, causes errors to be reported on the current output device.

The /LIST qualifier is the default for a MACRO command in a batch job. The /LIST qualifier allows you to control the defaults applied to the output file specification by the placement of the qualifier in the command line.

/MACHINE
/NOMACHINE (default)
Enables machine code listing, if it and the /LIST qualifier are both specified in the command line.

/OBJECT[=filespec]
/NOOBJECT
Creates or omits an object module. It also defines the file specification. By default, the compiler creates an object module with the same file name as the first input file. The default file type for object files is OBJ. No wildcard characters are allowed in the file specification.

The /OBJECT qualifier controls the defaults applied to the output file specification by the placement of the qualifier in the command line.

/OPTIMIZE[=(option[,...])]
/NOOPTIMIZE
Enables or disables optimization options. All options are enabled by default except VAXREGS. (See Section 4.3.1.)

Option	Description
[NO]PEEPHOLE	Peephole optimization
[NO]SCHEDULE	Code scheduling
[NO]ADDRESSES	Common base address loading
[NO]REFERENCES	Common data referencing
[NO]VAXREGS	Allow the use of VAX registers (R0 through R12) as temporary registers when they appear to be unused
ALL	All optimizations
NONE	No optimizations

Note that /ALL turns on VAXREGS, which may generate incorrect code unless all register usage of all routines in the module have been correctly declared.

/PRESERVE[=(option[,...])]
/NOPRESERVE (default)
Directs the compiler to generate special OpenVMS Alpha assembly code throughout a module for all VAX MACRO instructions that rely on VAX guarantees of operation atomicity or granularity. (See Section 2.10.) The options are:

Option	Description
GRANULARITY	Preserves the rules of VAX granularity of writes. Specifying /PRESERVE=GRANULARITY causes the compiler to use Alpha Load-locked and Store-conditional instruction sequences in code it generates for VAX instructions that perform byte, word, or unaligned longword writes.

Option	Description
ATOMICITY	Preserves atomicity of VAX modify operations. Specifying /PRESERVE=ATOMICITY causes the compiler to use Load...Locked and Store...Conditional instruction sequences in code it generates for instructions with modify operands.

/PRESERVE and /PRESERVE=(GRANULARITY,ATOMICITY) are equivalent. When preservation of both granularity and atomicity is enabled, and the compiler encounters a VAX coding construct that requires both granularity and atomicity guarantees, it enforces atomicity over granularity.

If you are aware of specific sections of VAX MACRO code that require VAX granularity and atomicity guarantees, you can forego compiler enforcement of these guarantees for the entire module and use the .PRESERVE and .NOPRESERVE directives (see Appendix B) to indicate those sections. Therefore, if you can isolate that code where granularity and atomicity *must* apply, these directives allow you to optimize the generated code by preventing the compiler from generating expanded Alpha code unnecessarily.

Atomicity is guaranteed on *multiprocessing* systems as well as uniprocessing systems when you specify /PRESERVE=ATOMICITY.

When the /PRESERVE qualifier is present, you can control the number of times compiler-generated code retries a granular or atomic update by specifying the /RETRY_COUNT qualifier.

Warning

If /PRESERVE=ATOMICITY is turned on, any unaligned data references will result in a fatal reserved operand fault. See Section 2.10.5. If /PRESERVE=GRANULARITY is turned on, unaligned word references to addresses assumed aligned will also cause a fatal reserved operand fault.

/RETRY_COUNT=count

Specifies to the compiler the number of times the following operations should be performed in generated code:

- Retries of operations performed in source by a VAX interlocked instruction
- Retries of atomic or granular updates if the /PRESERVE qualifier or .PRESERVE directive is present

If the /RETRY_COUNT qualifier is not present, the compiler generates code that performs an infinite number of retries of these operations.

/SHOW[=(function[,...])]
/NOSHOW[=(function[,...])]

Provides initial settings for the functions controlled by the compiler directives .SHOW and .NOSHOW.

You can specify one or more of the following functions:

CONDITIONALS	Lists unsatisfied conditional code associated with .IF and .ENDC MACRO directives.

CALLS	Lists macro calls and repeat range expansions.
DEFINITIONS	Lists macro definitions.
EXPANSIONS	Lists macro expansions.
BINARY	Lists binary code generated by the expansion of macro calls.

/SYMBOLS
/NOSYMBOLS (default)
Generates a symbol table and psect synopsis table for the listing file if it and the /LIST qualifier are both specified in the command line.

/TIE (default)
/NOTIE
Ensures that proper external callouts are generated for translated images. Translated images are images that were translated with the DECMigrate (also known as VEST) facility. The Translated Image Environment (TIE) allows translated images to execute as if on an OpenVMS VAX system. Use /NOTIE for better performance if you do not make calls to translated images.

/UNALIGNED
/NOUNALIGNED (default)
Forces the compiler to use unaligned loads and stores for *all* register-based memory references (except those that are FP–based or SP–based or are references to local aligned static data).

By default, the compiler assumes that addresses in registers used as base pointers (except those that are FP–based or SP–based) are longword-aligned at routine entry, and generates code to load BYTE, WORD, and LONG data accordingly. This can result in run-time alignment faults, with significant performance impact, if the assumption is incorrect. Specifying /UNALIGNED causes the compiler to generate code assuming pointers are unaligned. This code is significantly larger, but is more efficient than handling an alignment fault.

Note

The compiler does *not* track quadword register alignment. For quadword memory references (such as in VAX MOVQ instructions), the compiler assumes the base address is quadword aligned, *unless* it has determined the address may not be longword-aligned in its register tracking code. Quadword references in OpenVMS Alpha built-in uses are always assumed to be quadword aligned. Since these must be in new code, the data should be properly aligned.

The /UNALIGNED qualifier is generally appropriate only for modules where data is often unaligned, but which are not sufficiently performance sensitive to merit the correction of the data alignment in the source.

/WARN=[[option]...]
/NOWARN
Turns off all informational level or warning level messages. Both are on by default.

Option	Description
INFO	Turns on all informational level messages
NOINFO	Turns off all informational level messages
WARN	Turns on all informational *and* warning level messages
NOWARN	Turns off all informational *and* warning level messages

B
Compiler Directives

This appendix begins by briefly describing the compiler’s support of the VAX MACRO assembler directives. Then it lists the specialized directives of the MACRO-32 Compiler for OpenVMS Alpha and describes each one in detail.

B.1 Support of VAX MACRO Assembler Directives

The compiler supports most of the standard VAX MACRO assembler directives discussed in the *VAX MACRO and Instruction Set Reference Manual*. However, the following directives that are supported by the VAX MACRO assembler do not make sense for compiled code. Consequently, the compiler flags them and continues execution. You can disable the flagging of these directives by specifying /NOFLAG=DIRECTIVES.

- .ENABLE and .DISABLE ABSOLUTE—for forcing absolute addressing modes
- .ENABLE and .DISABLE TRUNCATION—for enabling floating point truncation
- .LINK—for specifying linker options in a linker options file
- .DEFAULT—for setting displacement lengths
- .OPDEF and .REF*n* —for defining opcodes
- Alignment directives (.ALIGN, .EVEN, and .ODD) in code psects
- .TRANSFER (see Section 3.7)
- .MASK

Note

The length of the argument to a .ASCID directive is limited to 996 characters when using the MACRO-32 Compiler for OpenVMS Alpha. No such restriction exists in the VAX MACRO Assembler.

B.2 Compiler Directives

The MACRO-32 Compiler for OpenVMS Alpha provides the following specialized directives:

- .BRANCH_LIKELY
- .BRANCH_UNLIKELY
- .CALL_ENTRY
- .DEFINE_PAL
- .DISABLE

- .ENABLE
- .EXCEPTION_ENTRY
- .GLOBAL_LABEL
- .JSB_ENTRY
- .JSB32_ENTRY
- .LINKAGE_PSECT
- .PRESERVE
- .SET_REGISTERS
- .SYMBOL_ALIGNMENT

You can use certain arguments to these directives to indicate register sets. You express a register set by listing the registers, separated by commas, within angle brackets. For example:

```
<R1,R2,R3>
```

If only one register is in the set, no angle brackets are used. For example:

```
R1
```

.BRANCH_LIKELY

Instructs the compiler that the following branch will likely be taken. Therefore, the compiler generates code that incorporates that assumption.

Format

.BRANCH_LIKELY

There are no parameters for this directive.

Example

```
MOVL (R0),R1
.BRANCH_LIKELY
BNEQ    10$
  .
  .
  .
10$
```

The compiler will move the code between the BNEQ instruction and label 10$ to the end of the module, and change the BNEQ 10$ to a BEQL to the moved code. It will then continue immediately following the BEQL instruction with generation of the code starting at label 10$. This will eliminate the delay that occurs on Alpha systems for a mispredicted branch.

.BRANCH_UNLIKELY

Instructs the compiler that the following branch will likely *not* be taken. Therefore, the compiler generates code that incorporates that assumption.

Format

.BRANCH_UNLIKELY

There are no parameters for this directive.

Description

The .BRANCH_UNLIKELY directive instructs the compiler that the following conditional branch will likely not be taken. The compiler then generates code that incorporates that assumption.

Alpha system hardware predicts that forward conditional branches are not taken. Therefore, if a .BRANCH_UNLIKELY directive precedes a branch that will be in the forward direction, it has no effect. However, if it precedes a branch that would be backward, code is generated to force the branch to be forward, to an out-of-line branch back to the actual branch destination.

.BRANCH_UNLIKELY should be used only in cases where the branch is very unlikely, not just less frequent than the fall-through case.

Example

```
        MOVL    #QUEUE,R0             ;Get queue header
10$:    MOVL    (R0),R0               ;Get entry from queue
        BEQL    20$                   ;Forward branch assumed unlikely
        .
        .                             ;Process queue entry
        .
        TSTL    (R0)                  ;More than one entry (known to be unlikely)
        .BRANCH_UNLIKELY
        BNEQ    10$                   ;This branch made into forward
20$:                                  ;conditional branch
```

The .BRANCH_UNLIKELY directive is used here because the Alpha hardware would predict a backward branch to 10$ as likely to be taken. The programmer knows it is a rare case, so the directive is used to change the branch to a forward branch, which is predicted not taken.

.CALL_ENTRY

Declares the entry point of a called routine to the compiler. This entry declaration will save and restore the full 64 bits of any registers (except R0 and R1) that are modified by the routine and are not declared as **scratch** or **output**.

Format

```
.CALL_ENTRY   [max_args] [,home_args=TRUE|FALSE]
              [,quad_args=TRUE|FALSE] [,input] [,output] [,scratch]
              [,preserve] [,label]
```

Parameters

max_args
Maximum number of arguments the called procedure expects. The compiler uses this value as the number of longwords it allocates in the fixed temporary region of the stack frame, if the argument list must be homed. If homing is not necessary, the **max_args** count is not required. The compiler flags procedure entry points, where **max_args** has not been specified, that require homed argument lists.

Note that, for .CALL_ENTRY routines in which **max_args** exceeds 14, the compiler uses the received argument count, or **max_args**, whichever is smaller, when homing the argument list.

home_args=TRUE | FALSE
Indication to the compiler that the called procedure's argument list should or should not be homed. The **home_args** argument overrides the compiler's default logic, as explained in Section 2.3.1, for determining the circumstances under which an argument list must be homed.

quad_args=TRUE | FALSE
Indication to the compiler that the called procedure's argument list will have quadword references.

input=<>
Register set that indicates those registers from which the routine receives input values.

This register set informs the compiler that the registers specified have meaningful values at routine entry and are unavailable for use as temporary registers even before the first compiler-detected use of the registers. Specifying registers in this register set affects compiler temporary register usage in two cases:

- If you are using the VAXREGS optimization switch. This optimization allows the compiler to use as temporary registers any of the VAX registers which are not explicitly being used by the VAX MACRO code.
- If you are explicitly using any of the Alpha registers (R13 and above).

In either of these cases, if you do not specify a register that is being used as input in the **input** argument, the compiler may use the register as a temporary register, corrupting the input value.

This register set has no effect on the compiler's default register preservation behavior. If you are not using the VAXREGS optimization switch or any of the Alpha registers, the input mask is used only to document your routine.

output=<>
Register set that indicates those registers to which the routine assigns values that are returned to the routine's caller. Registers included in this register set are not saved and restored by the compiler, even if they are modified by the routine.

This register set also informs the compiler that the registers specified have meaningful values at routine exit and are unavailable for use as temporary registers even after the last compiler-detected use of the registers. Specifying registers in this register set affects compiler temporary register usage in two cases:

- If you are using the VAXREGS optimization switch. This optimization allows the compiler to use as temporary registers any of the VAX registers which are not explicitly being used by the VAX MACRO code.
- If you are explicitly using any of the Alpha registers (R13 and above).

In either of these cases, if you do not specify a register that is being used as output in the **output** argument, the compiler may use the register as a temporary register, corrupting the output value.

scratch=<>
Register set that indicates registers that are used within the routine but which should not be saved and restored at routine entry and exit. The caller of the routine does not expect to receive output values nor does it expect the registers to be preserved. Registers included in this register set are not saved and restored by the compiler, even if they are modified by the routine.

This also pertains to the compiler's temporary register usage. The compiler may use registers R13 and above as temporary registers if they are unused in the routine source code. Because R13 through R15 must be preserved, if modified, according to the OpenVMS Alpha calling standard, the compiler preserves those registers if it uses them.

However, if they appear in the **scratch** register set declaration, the compiler will *not* preserve them if it uses them as temporary registers. As a result, these registers may be scratched at routine exit, even if they were not used in the routine source but are in the **scratch** set. If the VAXREGS optimization is used, this applies to registers R2 through R12, as well.

preserve=<>
Register set that indicates those registers that should be preserved over the routine call. This should include only those registers that are modified and whose full 64-bit contents should be saved and restored.

This register set causes registers to be preserved whether or not they would have been preserved automatically by the compiler. Note that because R0 and R1 are scratch registers, by calling standard definition, the compiler never saves and restores them unless you specify them in this register set.

This register set overrides the **output** and **scratch** register sets. If you specify a register both in the **preserve** register set and in the **output** or **scratch** register sets, the compiler will report the warning:

```
%AMAC-W-REGDECCON, register declaration conflict in routine A
```

label=*name*
Optionally specify a label as in a VAX MACRO .ENTRY directive. This can be used if a module is to be common between VAX and Alpha, the VAX version needs to reference the entry with a .MASK directive, and the Alpha version needs to use one or more of the special .CALL_ENTRY parameters. When the **label** parameter is specified and the symbol VAX is defined, an .ENTRY directive is used. (See Section 1.6.3). If the symbol VAX is not defined, it creates the label and does a normal .CALL_ENTRY. Note that **label** is not the first parameter. Therefore, you cannot simply replace .ENTRY with .CALL_ENTRY. You must use the **label** parameter declaration.

.DEFINE_PAL

Defines an arbitrary PALcode function such that it can be called later in the MACRO source.

Format

.DEFINE_PAL   name, pal_opcode, [,operand_descriptor_list]

Parameters

name
Name of the PALcode function. The compiler applies the prefix EVAX_ to the specified name (for instance, EVAX_MTPR_USP).

pal_opcode
Opcode value of the PALcode function. PALcode opcodes are listed in the *Alpha Architecture Reference Manual*.

Be sure to use angle brackets around the function code when specifying it in hexadecimal format (^X). If you specify the function code in decimal format, angle brackets are not necessary.

operand_descriptor_list
A list of operand descriptors that specifies the number of operands and the type of each. Up to 6 operand descriptors are allowed in the list. Be careful to specify operands correctly so that the compiler can correctly track register and stack usage. Table B–1 lists the operand descriptors.

Table B–1 Operand Descriptors

Access Type	Data Type			
	Byte	Word	Longword	Octaword
Address	AB	AW	AL	AQ
Read-only	RB	RW	RL	RQ
Modify	MB	MW	ML	MQ
Write-only	WB	WW	WL	WQ

Description

By default, the compiler defines many OpenVMS Alpha PALcode instructions as built-ins. These are listed in Appendix C. If you need to use an OpenVMS Alpha PALcode instruction that is not available as a compiler built-in, you must define the built-in yourself using the .DEFINE_PAL directive.

Example

```
.DEFINE_PAL MTPR_USP, <^X23>, RQ
```

Note

This is an example—the compiler compiles MTPR instructions directly to PAL calls.

.DISABLE

Disables compiler features over a range of source code.

Format

.DISABLE argument-list

Parameters

argument-list
You can use one or more of the symbolic arguments listed in the following table:

Option	Description
DEBUG	Excludes local symbol table information in the object file for use with the debugger.
FLAGGING	Deactivates compiler flagging.
GLOBAL	Disables the assumption that undefined symbols are external symbols.
OVERFLOW	Deactivates production of overflow trap code for the following opcodes: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (where x is greater than y, for example CVTLB), AOBxx, ACBL, and SOBxx.
QUADWORD	Disables support for quadword literal and address expressions.
SUPPRESSION	Stops the listing of unreferenced symbols in the symbol table.
TRACEBACK	Stops providing traceback information to the debugger.

.ENABLE

Enables compiler features over a range of source code.

Format

.ENABLE argument-list

Parameters

argument-list
You can use one or more of the symbolic arguments listed in the following table:

Option	Description
DEBUG[1]	Includes local symbol table information in the object file for use with the debugger.
FLAGGING	Activates compiler flagging.
GLOBAL	Assumes undefined symbols are external symbols.
OVERFLOW	Activates production of overflow trap code for the following opcodes: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (where x is greater than y, for example CVTLB), AOBxx, ACBL, and SOBxx.
QUADWORD	Provides support for quadword literal and address expressions.
SUPPRESSION	Provides a listing of unreferenced symbols in the symbol table.
TRACEBACK[2]	Provides traceback information to the debugger.

[1]To take effect, you must compile with /DEBUG or /ENABLE=DEBUG.
[2]To take effect, you must compile with /DEBUG or /ENABLE=TRACEBACK.

.EXCEPTION_ENTRY

Declares the entry point of an exception service routine to the compiler.

Format

.EXCEPTION_ENTRY [,stack_base]

Parameters

preserve=<>
Register set that forces the compiler to save and restore across the routine call the contents of registers. By default, the compiler saves at routine entry and restores at routine exit the full 64-bit contents of any register that is modified by a routine.

In the case of an .EXCEPTION_ENTRY routine, exception dispatching saves R2 through R7 on the stack (along with the PC and PSL) and the values of these registers are restored by the REI instruction executed by the routine itself. Other registers, if used, are saved in code generated by the compiler, and *all* other registers are saved if the routine issues a CALL or JSB instruction.

stack_base
Register into which the stack pointer (SP) value is moved at routine entry. At exception entry points, exception dispatching pushes onto the stack registers R2 through R7, the PC, and the PSL. Note that the Alpha counterpart for the VAX register known as the PSL is the processor status (PS) register. The value returned to the register specified in the **stack_base** helps an exception service routine locate the values of these registers.

You can use the macro $INTSTKDEF in SYS$LIBRARY:LIB.MLB to define symbols for the area on the stack where R2-R7, the PC, and the PSL are stored. The symbols are:

- INTSTK$Q_R2
- INTSTK$Q_R3
- INTSTK$Q_R4
- INTSTK$Q_R5
- INTSTK$Q_R6
- INTSTK$Q_R7
- INTSTK$Q_PC
- INTSTK$Q_PS

You can then use these symbols in the exception routine, as offsets to the stack_base value. By using the appropriate symbolic offset with the stack_base value, the exception routine can access the saved contents of any of these registers. For example, the exception routine could examine the PSL to see what access mode was in effect when the exception was taken.

Description

The .EXCEPTION_ENTRY directive indicates the entry point of an exception service routine. At routine entry, R3 must contain the address of the procedure descriptor. The routine must exit with an REI instruction.

You should declare with the .EXCEPTION_ENTRY directive all of the following interrupt service routines:

- Interval clock
- Interprocessor interrupt
- System/processor correctable error
- Power failure
- System/processor machine abort
- Software interrupt

.GLOBAL_LABEL

Declares a global label in a routine that is not an entry point to the routine.

Format

Label: .GLOBAL_LABEL

There are no parameters for this directive.

Description

The .GLOBAL_LABEL directive declares a global label within a routine that is not a routine entry point. Unless declared with .GLOBAL_LABEL, global labels in code (specified with "::") are assumed to be entry point labels, which require declaration. If they are not declared, they are flagged as errors.

The compiler also allows the address of a global label to be stored (for instance, by means of PUSHAL instruction). (The compiler flags as an error any attempt to store a label that has not been declared as a global label or an entry point.)

By using the .GLOBAL_LABEL directive, the user is acknowledging that the stored code address will not be the target of a CALL or JSB instruction. Global labels must appear inside routine boundaries.

Labels declared with the .GLOBAL_LABEL directive can be used as the **newpc** argument in calls to the $UNWIND (Unwind Call Stack) system service because it allows the address of the label to be stored.

However, there is no provision in the compiler to automatically adjust the stack pointer at such labels to remove arguments passed on the stack or compensate for stack alignment. If the call stack is unwound back to an alternate PC in the calling routine, the stack may still contain arguments and alignment bytes, and any stack-based references that expect this adjustment to the caller's original stack depth (which happened automatically on VAX) will be incorrect.

Code that contains labels declared with this directive that are to be used as alternate PC targets for $UNWIND must be examined carefully to ensure correct behavior, with particular emphasis on any references based on the stack pointer.

.JSB_ENTRY

Declares the entry point of a JSB routine to the compiler. This entry declaration will save and restore the full 64 bits of any registers (except R0 and R1) that are modified by the routine and are not declared as **scratch** or **output**. See also .JSB32_ENTRY.

Format

.JSB_ENTRY [input] [,output] [,scratch] [,preserve]

Parameters

input=<>

Register set that indicates those registers from which the routine receives input values.

This register set informs the compiler that the registers specified have meaningful values at routine entry and are unavailable for use as temporary registers even before the first compiler-detected use of the registers. Specifying registers in this register set affects compiler temporary register usage in two cases:

- If you are using the VAXREGS optimization switch. This optimization allows the compiler to use as temporary registers any of the VAX registers which are not explicitly being used by the VAX MACRO code.
- If you are explicitly using any of the Alpha registers (R13 and above).

In either of these cases, if you do not specify a register that is being used as input in the **input** argument, the compiler may use the register as a temporary register, corrupting the input value.

This register set has no effect on the compiler's default register preservation behavior. If you are not using the VAXREGS optimization switch or any of the Alpha registers, the input mask is used only to document your routine.

output=<>

Register set that indicates those registers to which the routine assigns values that are returned to the routine's caller. Registers included in this register set are not saved and restored by the compiler, even if they are modified by the routine.

This register set also informs the compiler that the registers specified have meaningful values at routine exit and are unavailable for use as temporary registers even after the last compiler-detected use of the registers. Specifying registers in this register set affects compiler temporary register usage in two cases:

- If you are using the VAXREGS optimization switch. This optimization allows the compiler to use as temporary registers any of the VAX registers which are not explicitly being used by the VAX MACRO code.
- If you are explicitly using any of the Alpha registers (R13 and above).

In either of these cases, if you do not specify a register that is being used as output in the **output** argument, the compiler may use the register as a temporary register, corrupting the output value.

scratch=<>

Register set that indicates registers that are used within the routine but which should not be saved and restored at routine entry and exit. The caller of the routine does not expect to receive output values nor does it expect the registers to be preserved. Registers included in this register set are not saved and restored by the compiler, even if they are modified by the routine.

The compiler may use registers R13 and above as temporary registers if they are unused in the routine source code. Because R13 through R15 must be preserved, if modified, according to the OpenVMS Alpha calling standard, the compiler preserves those registers if it uses them.

However, if they appear in the **scratch** register set declaration, the compiler will *not* preserve them if it uses them as temporary registers. As a result, these registers may be scratched at routine exit, even if they were not used in the routine source but are in the **scratch** set. If the VAXREGS optimization is used, this applies to registers R2 through R12, as well.

preserve=<>
Register set that indicates those registers that should be preserved over the routine call. This should include only those registers that are modified and whose full 64-bit contents should be saved and restored.

This register set causes registers to be preserved whether or not they would have been preserved automatically by the compiler. Note that because R0 and R1 are scratch registers, by calling standard definition, the compiler never saves and restores them unless you specify them in this register set.

This register set overrides the **output** and **scratch** register sets. If you specify a register both in the **preserve** register set and in the **output** or **scratch** register sets, the compiler will report the following warning:

```
%AMAC-W-REGDECCON, register declaration conflict in routine A
```

Note

For procedures declared with the .JSB_ENTRY directive, the MACRO-32 compiler automatically generates a null frame procedure descriptor.

Because no new context is set up by a null frame procedure, a side effect is that there is no guarantee of completely accurate debugger information about such procedures in response to SHOW CALLS and SHOW STACK commands. For example, the line number in the called null procedure (to which a JSB is done) may be reported as the line number in the calling procedure from which the JSB is issued.

.JSB32_ENTRY

Declares the entry point of a JSB routine to the compiler. This directive does not preserve any VAX register values (R2 through R12) unless the PRESERVE parameter is specified. The routine itself may save and restore registers by pushing them on the stack, but this will not preserve the upper 32 bits of the registers. See also .JSB_ENTRY.

Warning

The .JSB32_ENTRY directive can be a great time-saver *if* you are sure that you can use it. If you use .JSB32_ENTRY in a situation where the upper 32 bits of a register *are* being used, it may cause very obscure and difficult-to-track bugs by corrupting a 64-bit value that may be several calling levels above the offending routine.

.JSB32_ENTRY should *never* be used in an AST routine, condition handler, or any other code that can be executed asynchronously.

Format

.JSB32_ENTRY [input] [,output] [,scratch] [,preserve]

Parameters

input=<>
Register set that indicates those registers from which the routine receives input values.

For the .JSB32_ENTRY directive, this register set is used only to document your code.

output=<>
Register set that indicates those registers to which the routine assigns values that are returned to the routine's caller.

For the .JSB32_ENTRY directive, this register set is used only to document your code.

scratch=<>
Register set that indicates registers that are used within the routine but which should not be saved and restored at routine entry and exit. The caller of the routine does not expect to receive output values nor does it expect the registers to be preserved.

The **scratch** argument also pertains to the compiler's temporary register usage. The compiler may use registers R13 and above as temporary registers if they are unused in the routine source code. Because R13 through R15 must be preserved, if modified, according to the OpenVMS Alpha calling standard, the compiler preserves those registers if it uses them.

However, if they appear in the **scratch** register set declaration, the compiler will *not* preserve them if it uses them as temporary registers. As a result, these registers may be scratched at routine exit, even if they were not used in the routine source but are in the **scratch** set.

Because R2 through R12 are not preserved by default, their inclusion in the **scratch** is for documentation purposes only.

preserve=<>
Register set that indicates those registers that should be preserved over the routine call. This should include only those registers that are modified and whose full 64-bit contents should be saved and restored.

This register set causes registers to be preserved by the compiler. By default, no registers are preserved by the .JSB32_ENTRY directive.

This register set overrides the **output** and **scratch** register sets. If you specify a register both in the **preserve** register set and in the **output** or **scratch** register sets, the compiler will report the warning:

%AMAC-W-REGDECCON, register declaration conflict in routine A

Description

The .JSB32_ENTRY directive is an alternative way of declaring a JSB entry point. It is designed to streamline the declaration of VAX MACRO routines that operate within a well-defined, bounded application environment, such as that of a single application or a self-contained subsystem. For any routine declared with the .JSB32_ENTRY directive, the compiler does not automatically save or restore any VAX registers (R2 through R12), therefore leaving the current 32-bit operation untouched. When you use the .JSB32_ENTRY directive to declare a JSB entry point, you are responsible for declaring and saving registers which must be preserved.

If the externally visible entry points of a subsystem can be called from the 64-bit environment, those entry points should not be declared with .JSB32_ENTRY. Instead, .JSB_ENTRY (or .CALL_ENTRY) should be used so that the full 64-bit register values are saved, if necessary.

.LINKAGE_PSECT

Allows the name of the linkage section psect to be changed.

Format

.LINKAGE_PSECT   program-section-name

Parameters

program_section_name
Name of the program section. The name can contain up to 31 characters, including any alphanumeric character and the special characters underline (_), dollar sign ($), and period (.).

Description

The .LINKAGE_PSECT directive allows you to locate a routine's linkage section by reference to other psects within the routine. This facilitates such operations as locking code within memory (see Section 3.10) and forcing code location. An example of forcing code location is to explicitly place the psect in the image created by the linker, using linker options. This would let you use adjacent psects to find their bounds.

You can use the .LINKAGE_PSECT directive multiple times within a single source module to set different linkage sections for different routines. However, note that a routine's linkage section remains the same for the entire routine. The name of the routine's linkage section is the one specified in the last .LINKAGE_PSECT directive before the routine's entry point directive.

The compiler reports a fatal error if different linkage sections are specified for routines that share code paths.

The .LINKAGE_PSECT directive sets the psect attributes to be the same as the default linkage psect $LINKAGE. The attributes are the same as the normal psect default attributes except the linkage psect is set NOEXE NOWRT.

You can change the linkage section psect attributes using the .PSECT directive after declaring the psect with .LINKAGE_PSECT.

Example

```
          .LINKAGE_PSECT LINK_001
          .PSECT LINK_000
LS_START:
          .PSECT LINK_002
LS_END:
```

This code allows a program to use LS_START and LS_END in computations to determine the location and size of the linkage section (LINK_001) of the routine.

.PRESERVE

Directs the compiler to generate special Alpha assembly code for VAX MACRO instructions, within portions of the source module, that rely on VAX guarantees of operation atomicity or granularity.

Format

.[NO]PRESERVE   argument-list

Parameters

argument-list
One or more of the symbolic arguments listed in the following table:

Option	Description
GRANULARITY	Preserves the rules of VAX granularity of writes. Specifying .PRESERVE GRANULARITY causes the compiler to use Alpha Load-locked and Store-conditional instruction sequences in code it generates for VAX instructions that perform byte, word, or unaligned longword writes.
ATOMICITY	Preserves atomicity of VAX modify operations. Specifying .PRESERVE ATOMICITY causes the compiler to use Load-locked and Store-conditional instruction sequences in code it generates for instructions with modify operands.

Description

The .PRESERVE and .NOPRESERVE directives cause the compiler to generate special Alpha assembly code for VAX MACRO instructions, within portions of the source module, that rely on VAX guarantees of operation atomicity or granularity (see Section 2.10).

Use of .PRESERVE or .NOPRESERVE without specifying GRANULARITY or ATOMICITY will affect both options. When preservation of both granularity and atomicity is enabled, and the compiler encounters a VAX coding construct that requires both granularity and atomicity guarantees, it enforces atomicity over granularity.

Alternatively, you can use the /PRESERVE and /NOPRESERVE compiler qualifiers to affect the atomicity and granularity in generated code throughout an entire MACRO source module.

Atomicity is guaranteed for *multiprocessing* systems as well as uniprocessing systems when you specify .PRESERVE ATOMICITY.

When the .PRESERVE directive is present, you can use the /RETRY_COUNT qualifier on the command line to control the number of times the compiler-generated code retries a granular or atomic update.

Warning

If .PRESERVE ATOMICITY is turned on, any unaligned data references will result in a fatal reserved operand fault. See Section 2.10.5. If .PRESERVE GRANULARITY is turned on, unaligned word references to addresses assumed aligned will also cause a fatal reserved operand fault.

Example

```
INCW 1(R0)
```

This instruction, when compiled with .PRESERVE GRANULARITY, retries the insertion of the new word value, if it is interrupted. However, when compiled with .PRESERVE ATOMICITY, it will also refetch the initial value and increment it, if interrupted. If both options are specified, it will do the latter.

.SET_REGISTERS

This directive allows the user to override the compiler's alignment assumptions, and also allows implicit reads/writes of registers to be declared.

Format

.SET_REGISTERS argument-list

Parameters

argument-list
One or more of the arguments listed in the following table. For each argument, you can specify one or more registers.

Option	Description
aligned=<>	Declares one or more registers to be aligned on longword boundaries.
unaligned=<>	Declares one or more registers to be unaligned. Because this is an explicit declaration, this unaligned condition will not produce a fault at run time.
read=<>	Declares one or more registers, which otherwise the compiler could not detect as input registers, to be read.

Option	Description
written=<>	Declares one or more registers, which otherwise the compiler could not detect as output registers, to be written to.

Description

The **aligned** and **unaligned** qualifiers to this directive allow the user to override the compiler's alignment assumptions. Using the directive for this purpose in certain cases can produce more efficient code (see Section 4.1).

The **read** and **written** qualifiers to this directive allow implicit reads and writes of registers to be declared. They are generally used to declare the register usage of called routines and are useful for documenting your program.

With one exception, the .SET_REGISTERS directive remains in effect (ensuring proper alignment processing) until the routine ends, unless you change the value in the register. The exception can occur under certain conditions when a flow path joins the code following a .SET_REGISTERS directive.

The following example illustrates such an exception. R2 is declared **aligned**, and at a subsequent label, 10$, which is before the next write access to the register, a flow path joins the code. R2 will be treated as **unaligned** following the label, because it is unaligned from the other path.

```
        INCL R2         ; R2 is now unaligned
          .
          .
          .
        BLBC R0, 10$
          .
          .
          .
        MOVL R5, R2
        .SET_REGISTERS ALIGNED=R2
        MOVL R0, 4(R2)
  10$:  MOVL 4(R2), R3   ; R2 considered unaligned
                         ; due to BLBC branch
```

The .SET_REGISTERS directive and its **read** and **written** qualifiers are required on every routine call that passes or returns data in any register from R2 through R12, if you specify the command line qualifier and option **/OPTIMIZE=VAXREGS**. That is because the compiler allows the use of unused VAX registers as temporary registers when you specify **/OPTIMIZE=VAXREGS**.

Examples

1.
```
DIVLR0,R1
.SET_REGISTERS ALIGNED=R1
MOVL     8(R1), R2          ; Compiler will use aligned load.
```

In this example, the compiler would normally consider R1 *unaligned* after the division. Any memory references using R1 as a base register (until it is changed again) would use unaligned load/stores. If it is known that the actual value will always be aligned, performance could be improved by adding a .SET_REGISTERS directive, as shown.

2.
```
MOV1     4(R0), R1          ; Stored memory addresses assumed
.SET_REGISTERS UNALIGNED=R1 ; aligned so explicitly set it unaligned
MOVL     4(R1), R2          ; to avoid run-time fault.
```

In this example, R1 would be considered longword aligned after the MOVL. If it is actually unaligned, an alignment fault would occur on memory reference that follows at run time. To prevent this, the .SET_REGISTERS directive can be used, as shown.

3.
```
.SET_REGISTERS READ=<R3,R4>, WRITTEN=R5
JSB     DO_SOMETHING_USEFUL
```

In this example, the read/written attributes are used to explicitly declare register uses which the compiler cannot detect. R3 and R4 are input registers to the JSB target routine, and R5 is an output register. This is particularly useful if the routine containing this JSB does not use these registers itself, or if the SET_REGISTERS directive and JSB are embedded in a macro. When compiled with /FLAG=HINTS, routines which use the macro would then have R3 and R4 listed as possible input registers, even if they are not used in that routine.

.SYMBOL_ALIGNMENT

This directive associates an alignment attribute with a symbol definition for a register offset. You can use this directive when you know the alignment of the base register. This attribute guarantees to the compiler that the base register has the same alignment, which enables the compiler to generate optimal code.

Format

.SYMBOL_ALIGNMENT   argument-list

Parameters

argument-list
One of the arguments listed in the following table.

Option	Description
long	Declares longword alignment for any symbol that you declare after this directive.
quad	Declares quadword alignment for any symbol that you declare after this directive.
none	Turns off the alignment specified by the preceding .SYMBOL_ ALIGNMENT directive.

Description

The .SYMBOL_ALIGNMENT directive is used to associate an alignment attribute with the fields in a structure when you know the base alignment. It is used in pairs. The first .SYMBOL_ALIGNMENT directive associates either longword (**long**) or quadword (**quad**) alignment with the symbol or symbols that follow. The second directive, .SYMBOL_ALIGNMENT **none**, turns it off.

Any time a reference is made with a symbol with an alignment attribute, the base register of that reference, in effect, inherits the symbol's alignment. The compiler also resets the base register's alignment to longword for subsequent alignment tracking. This alignment guarantee enables the compiler to produce more efficient code sequences.

Example

```
OFFSET1 = 4
.SYMBOL_ALIGNMENT LONG
OFFSET2 = 8
OFFSET3 = 12
.SYMBOL_ALIGNMENT QUAD
OFFSET4 = 16
.SYMBOL_ALIGNMENT NONE
OFFSET5 = 20
    .
    .
    .
CLR1 OFFSET2(R8)
    .
    .
    .
MOVL R2, OFFSET4(R6)
```

For OFFSET1 and OFFSET5, the compiler will use only its tracking information for deciding if R*n* in OFFSET1(R*n*) is aligned or not. For the other references, the base register will be treated as longword (OFFSET2 and OFFSET3) or quadword (OFFSET4) aligned.

After each use of OFFSET2 or OFFSET4, the base register in the reference is reset to longword alignment. In this example, the alignment of R8 and R6 will be reset to longword, although the reference to OFFSET4 will use the stronger quadword alignment.

C Compiler Built-Ins

This appendix describes the two sets of built-ins provided with the MACRO-32 Compiler for OpenVMS Alpha. They are:

- Alpha instruction built-ins which are used to access Alpha instructions for which there are no VAX equivalents.
- Alpha PALcode built-ins which are used to emulate the VAX instructions for which there are no Alpha equivalents and to perform other functions such as quadword queue manipulations.

Both sets of built-ins are presented in tables. The second column of each table specifies the operands the built-in expects, where:

WL = write longword
ML = modify longword
AL = address of longword
WQ = write quadword
RQ = read quadword
MQ = modify quadword
AQ = address of quadword
AB = address of byte
AW = address of word
WB = write byte
WW = write word

Note

Be careful when mixing built-ins with VAX MACRO instructions on the same registers. The code generated by the compiler expects registers to contain 32-bit sign extended values, but it is possible to create 64-bit register values that are not in this format. Subsequent longword operations on these registers could produce incorrect results.

Therefore, make sure to return registers to 32-bit sign extended format before using them in VAX MACRO instructions as source operands. (Loading the register with a new value using a VAX MACRO instruction (such as MOVL) returns it to this format.)

C.1 Alpha Instruction Built-Ins

Ported VAX MACRO code sometimes requires access to Alpha native instructions to deal directly with a 64-bit quantity or to include an Alpha instruction that has no VAX equivalent. The compiler provides built-ins to allow you access to these instructions.

You use these built-ins in the same way that you use native VAX instructions, using any VAX operand mode. For example, EVAX_ADDQ 8(R0),(SP)+,R1 is legal. The only exception is that the first operand of any Alpha load/store built-in (EVAX_LD*, EVAX_ST*) *must* be a register.

It is recommended that you place any built-in within an ".IF DF,EVAX" conditional code block unless the module is Alpha specific. They can appear in Alpha specific portions of the macro definitions described in Appendix D.

The following byte and word built-ins are included in the MACRO-32 compiler, starting with OpenVMS Alpha Version 7.1:

- EVAX_LDBU
- EVAX_LDWU
- EVAX_STB
- EVAX_STW
- EVAX_SEXTB
- EVAX_SEXTW

The best environment in which to run code that contains the byte and word built-ins is on an Alpha computer that implements these instructions in hardware. If you run such code on an OpenVMS Alpha system that implements them by software emulation, the following limitations exist:

- Significant performance loss

  The overhead of handling the exception to trigger the software emulation causes a significant performance loss. If software emulation is in effect, you will see the following message:

  ```
  %SYSTEM-I-EMULATED, an instruction not implemented on this processor was emulated
  ```

- Some capabilities not present in the software emulation

  The software emulation is not capable of providing all the capabilities that would be present on a system that implemented the the instructions in hardware. Code that executes in inner access modes and at elevated IPL is allowed to use these instructions. For example, activation of the software emulator above IPL 2 will not cause a bugcheck. However, certain applications where these instructions might be useful, such as direct writes to hardware control registers, will be impossible, because such applications require the presence of address lines whose function cannot be emulated.

Furthermore, if the code with these built-ins executes on a system without either the byte and word software emulator or a processor that implements the byte and word instructions in hardware, it will incur a fatal exception, such as the following:

```
%SYSTEM-F-OPCDEC, opcode reserved to Digital fault at
PC=0000000000020068,PS=0000001B
```

Table C–1 summarizes the Alpha built-ins supported by the compiler.

Note

Memory references in the MACRO-32 compiler built-ins are always assumed to be quadword aligned except in EVAX_SEXTB, EVAX_SEXTW, EVAX_LDBU, EVAX_LDWU, EVAX_STB, EVAX_STW, EVAX_LDQU, and EVAX_STQU.

Table C–1 Alpha Instruction Built-Ins

Built-in	Operands	Description
EVAX_SEXTB	<RQ,WB>	Sign extend byte
EVAX_SEXTW	<RQ,WW>	Sign extend word
EVAX_SEXTL	<RQ,WL>	Sign extend longword
EVAX_LDBU	<WQ,AB>	Load zero-extended byte from memory
EVAX_LDWU	<WQ,AQ>	Load zero-extended word from memory
EVAX_LDLL	<WL,AL>	Load longword locked
EVAX_LDAQ	<WQ,AQ>	Load address of quadword
EVAX_LDQ	<WQ,AQ>	Load quadword
EVAX_LDQL	<WQ,AQ>	Load quadword locked
EVAX_LDQU	<WQ,AQ>	Load unaligned quadword
EVAX_STB	<RQ,AB>	Store byte from register to memory
EVAX_STW	<RQ,AW>	Store word from register to memory
EVAX_STLC	<ML,AL>	Store longword conditional
EVAX_STQ	<RQ,AQ>	Store quadword
EVAX_STQC	<MQ,AQ>	Store quadword conditional
EVAX_STQU	<RQ,AQ>	Store unaligned quadword
EVAX_ADDQ	<RQ,RQ,WQ>	Quadword add
EVAX_SUBQ	<RQ,RQ,WQ>	Quadword subtract
EVAX_MULQ	<RQ,RQ,WQ>	Quadword multiply
EVAX_UMULH	<RQ,RQ,WQ>	Unsigned quadword multiply high
EVAX_AND	<RQ,RQ,WQ>	Logical product
EVAX_OR	<RQ,RQ,WQ>	Logical sum
EVAX_XOR	<RQ,RQ,WQ>	Logical difference
EVAX_BIC	<RQ,RQ,WQ>	Bit clear
EVAX_ORNOT	<RQ,RQ,WQ>	Logical sum with complement
EVAX_EQV	<RQ,RQ,WQ>	Logical equivalence
EVAX_SLL	<RQ,RQ,WQ>	Shift left logical
EVAX_SRL	<RQ,RQ,WQ>	Shift right logical
EVAX_SRA	<RQ,RQ,WQ>	Shift right arithmetic

(continued on next page)

Table C–1 (Cont.) Alpha Instruction Built-Ins

Built-in	Operands	Description
EVAX_EXTBL	<RQ,RQ,WQ>	Extract byte low
EVAX_EXTWL	<RQ,RQ,WQ>	Extract word low
EVAX_EXTLL	<RQ,RQ,WQ>	Extract longword low
EVAX_EXTQL	<RQ,RQ,WQ>	Extract quadword low
EVAX_EXTBH	<RQ,RQ,WQ>	Extract byte high
EVAX_EXTWH	<RQ,RQ,WQ>	Extract word high
EVAX_EXTLH	<RQ,RQ,WQ>	Extract longword high
EVAX_EXTQH	<RQ,RQ,WQ>	Extract quadword high
EVAX_INSBL	<RQ,RQ,WQ>	Insert byte low
EVAX_INSWL	<RQ,RQ,WQ>	Insert word low
EVAX_INSLL	<RQ,RQ,WQ>	Insert longword low
EVAX_INSQL	<RQ,RQ,WQ>	Insert quadword low
EVAX_INSBH	<RQ,RQ,WQ>	Insert byte high
EVAX_INSWH	<RQ,RQ,WQ>	Insert word high
EVAX_INSLH	<RQ,RQ,WQ>	Insert longword high
EVAX_INSQH	<RQ,RQ,WQ>	Insert quadword high
EVAX_TRAPB	<>	Trap barrier
EVAX_MB	<>	Memory barrier
EVAX_RPCC	<WQ>	Read process cycle counter
EVAX_CMPEQ	<RQ,RQ,WQ>	Integer signed compare, equal
EVAX_CMPLT	<RQ,RQ,WQ>	Integer signed compare, less than
EVAX_CMPLE	<RQ,RQ,WQ>	Integer signed compare, less equal
EVAX_CMPULT	<RQ,RQ,WQ>	Integer unsigned compare, less than
EVAX_CMPULE	<RQ,RQ,WQ>	Integer unsigned compare, less equal
EVAX_BEQ	<RQ,AQ>	Branch equal
EVAX_BLT	<RQ,AQ>	Branch less than
EVAX_BNE	<RQ,AQ>	Branch not equal
EVAX_CMOVEQ	<RQ,RQ,WQ>	Conditional move/equal
EVAX_CMOVNE	<RQ,RQ,WQ>	Conditional move/not equal
EVAX_CMOVLT	<RQ,RQ,WQ>	Conditional move/less than
EVAX_CMOVLE	<RQ,RQ,WQ>	Conditional move/less or equal
EVAX_CMOVGT	<RQ,RQ,WQ>	Conditional move/greater than
EVAX_CMOVGE	<RQ,RQ,WQ>	Conditional move/greater or equal

(continued on next page)

Table C–1 (Cont.) Alpha Instruction Built-Ins

Built-in	Operands	Description
EVAX_CMOVLBC	<RQ,RQ,WQ>	Conditional move/low bit clear
EVAX_CMOVLBS	<RQ,RQ,WQ>	Conditional move/low bit set
EVAX_MF_FPCR	<WQ>	Move from floating-point control register
EVAX_MT_FPCR	<WQ,RQ>	Move to floating-point control register
EVAX_ZAP	<RQ,RQ,WQ>	Zero bytes
EVAX_ZAPNOT	<RQ,RQ,WQ>	Zero bytes with NOT mask

C.2 Alpha PALcode Built-Ins

Alpha PALcode built-ins, primarily for privileged code, are used in the same way that Alpha instruction built-ins are used with two exceptions:

- For the queue PAL functions, the compiler does not move the input arguments to the Alpha registers before issuing the PAL call as it does for all other functions. Therefore, you must supply the code to do that.
- When using a built-in for a PAL call that returns a value, the source code must explicitly read R0 for the return value.

Certain Alpha PALcode built-ins, EVAX_INSQHIQR, EVAX_INSQTIQR, EVAX_REMQHIQR, and EVAX_REMQHITR, support the manipulation of quadword queues, a function that VAX MACRO-32 does not support. If you use these built-ins, you must supply the code to move the input arguments to R16 (and R17, for EVAX_INSQ*xxxx*), as shown in the following example:

```
MOVAB   Q_header, R16   ; Set up address of queue header for PAL call
EVAX_REMQHIQR           ; Remove quadword queue entry
EVAX_STQ  R0, entry     ; Save entry address returned in R0
```

The Alpha PALcode built-ins are listed in Table C–2.

Note

You can use the .DEFINE_PAL compiler directive to custom-define a built-in for an Alpha PALcode operation that is not listed in this table. See Appendix B for additional information.

Table C–2 Alpha PALcode Built-Ins

Built-in	Operands	Description
EVAX_CFLUSH	<RQ>	Cache flush
EVAX_DRAINA	<>	Drain aborts
EVAX_LDQP	<AQ>	Load quadword physical
EVAX_STQP	<AQ,RQ>	Store quadword physical

(continued on next page)

Table C–2 (Cont.) Alpha PALcode Built-Ins

Built-in	Operands	Description
EVAX_SWPCTX	<AQ>	Swap privileged context
EVAX_BUGCHK	<RQ>	Bugcheck
EVAX_CHMS	<>	Change mode supervisor
EVAX_CHMU	<>	Change mode user
EVAX_IMB	<>	Instruction memory barrier
EVAX_SWASTEN	<RQ>	Swap AST enable
EVAX_WR_PS_SW	<RQ>	Write processor status software field
EVAX_MTPR_ASTEN	<RQ>	Move to processor register ASTEN
EVAX_MTPR_ASTSR	<RQ>	Move to processor register ASTSR
EVAX_MTPR_AT	<RQ>	Move to processor register AT
EVAX_MTPR_FEN	<RQ>	Move to processor register FEN
EVAX_MTPR_IPIR	<RQ>	Move to processor register IPIR
EVAX_MTPR_IPL	<RQ>	Move to processor register IPL
EVAX_MTPR_PRBR	<RQ>	Move to processor register PRBR
EVAX_MTPR_SCBB	<RQ>	Move to processor register SCBB
EVAX_MTPR_SIRR	<RQ>	Move to processor register SIRR
EVAX_MTPR_TBIA	<>	Move to processor register TBIA
EVAX_MTPR_TBIAP	<>	Move to processor register TBIAP
EVAX_MTPR_TBIS	<AQ>	Move to processor register TBIS
EVAX_MTPR_TBISD	<AQ>	Move to processor register, TB invalidate single DATA
EVAX_MTPR_TBISI	<AQ>	Move to processor register, TB invalidate single ISTREAM
EVAX_MTPR_ESP	<AQ>	Move to processor register ESP
EVAX_MTPR_SSP	<AQ>	Move to processor register SSP
EVAX_MTPR_USP	<AQ>	Move to processor register USP
EVAX_MFPR_ASN	<>	Move from processor register ASN
EVAX_MFPR_AT	<>	Move from processor register AT
EVAX_MFPR_FEN	<>	Move from processor register FEN
EVAX_MFPR_IPL	<>	Move from processor register IPL
EVAX_MFPR_MCES	<>	Move from processor register MCES
EVAX_MFPR_PCBB	<>	Move from processor register PCBB
EVAX_MFPR_PRBR	<>	Move from processor register PRBR
EVAX_MFPR_PTBR	<>	Move from processor register PTBR
EVAX_MFPR_SCBB	<>	Move from processor register SCBB
EVAX_MFPR_SISR	<>	Move from processor register SISR
EVAX_MFPR_TBCHK	<AQ>	Move from processor register TBCHK

(continued on next page)

Table C–2 (Cont.) Alpha PALcode Built-Ins

Built-in	Operands	Description
EVAX_MFPR_ESP	<>	Move from processor register ESP
EVAX_MFPR_SSP	<>	Move from processor register SSP
EVAX_MFPR_USP	<>	Move from processor register USP
EVAX_MFPR_WHAMI	<>	Move from processor register WHAMI
EVAX_INSQHILR	<>	Insert entry into longword queue at head interlocked-resident
EVAX_INSQTILR	<>	Insert entry into longword queue at tail interlocked-resident
EVAX_INSQHIQR	<>	Insert entry into quadword queue at head interlocked-resident
EVAX_INSQTIQR	<>	Insert entry into quadword queue at tail interlocked-resident
EVAX_REMQHILR	<>	Remove entry from longword queue at head interlocked-resident
EVAX_REMQTILR	<>	Remove entry from longword queue at tail interlocked-resident
EVAX_REMQHIQR	<>	Remove entry from quadword queue at head interlocked-resident
EVAX_REMQTIQR	<>	Remove entry from quadword queue at tail interlocked-resident
EVAX_GENTRAP	<>	Generate trap exception
EVAX_READ_UNQ	<>	Read unique context
EVAX_WRITE_UNQ	<RQ>	Write unique context

D
Macros for Porting to OpenVMS Alpha

This appendix describes macros designed to facilitate the porting of VAX MACRO code to an OpenVMS Alpha system. They are grouped according to their function, as follows:

- Page-related calculations (see Section D.1)
- Saving and restoring Alpha registers (see Section D.2)
- Locking pages into a working set (see Section D.3)

Note that you can use certain arguments to the macros described in this appendix to indicate register sets. To express a register set, list the registers, separated by commas, within angle brackets. For instance:

```
<R1,R2,R3>
```

If the set contains only one register, omit the angle brackets:

```
R1
```

D.1 Page-Related Calculations

This section describes the following macros that provide a standard, architecture-independent means for calculating page-size dependent values:

- $BYTES_TO_PAGES
- $NEXT_PAGE
- $PAGES_TO_BYTES
- $PREVIOUS_PAGE
- $ROUND_RETADR
- $START_OF_PAGE

These macros reside in the directory SYS$LIBRARY:STARLET.MLB and can be used by both application code and system code. Because application code does not have access to SYSTEM_DATA_CELLS, the user must supply the relevant masks, shift values, and so on.

The shift values are correlated with the page size of the processor. The **rightshift** values are negative; the **leftshift** values are positive, as shown in Table D–1.

Table D–1 Shift Values

Page size	rightshift	leftshift
512 bytes (VAX)	-9	9
8K (Alpha)	-13	13
16K[1]	-14	14
32K[1]	-15	15
64K[1]	-16	16

[1]If a future Alpha system implements this architecturally-permitted larger page size.

Typically, the application issues a call to $GETSYI (specifying the SYI$_PAGESIZE item descriptor) to obtain the CPU-specific page size and then compute other values from the page size that is returned.

The following conventions apply to the macros described in this section:

- If the destination operand is blank, the source operand is used as the destination.
- All macros conditionalize code on the symbols VAXPAGE and BIGPAGE.
- Several macros allow for page-size-independent code on VAX systems with the **independent**=**YES** argument. These macros generate code in which I-stream fetches are changed to memory accesses. Because this is inherently slower on a VAX system, the default value of the **independent** argument is **NO**.

$BYTES_TO_PAGES

Converts a byte count to a page count.

Format

```
$BYTES_TO_PAGES   source_bytcnt, dest_pagcnt, rightshift, roundup=YES,
                  quad=YES
```

Parameters

source_bytcnt
Source byte count.

dest_pagcnt
Destination of page count.

rightshift
Location of application-provided value to shift (in place of multiply). This value is a function of the page size, as shown in Table D–1.

roundup=YES
If **YES**, page-size–1 is added to byte count before shifting; if **NO**, page count is truncated. Any other value is treated as the user-specified address of the page-size–1 value. Note that **roundup**=**YES** is incompatible with the presence of the

rightshift argument; invoking the macro with both these arguments generates a compile-time warning.

quad=YES
If **YES**, the conversion supports 64-bit addressing. If **NO**, the conversion does not support 64-bit addressing.

$NEXT_PAGE

Computes the virtual address of the first byte in the next page.

Format

```
$NEXT_PAGE   source_va, dest_va, clearbwp=NO, user_pagesize_addr,
             user_mask_addr, quad=YES
```

Parameters

source_va
Source virtual address.

dest_va
Destination of virtual address within next page.

clearbwp=NO
If **YES**, masks the byte-within-page portion of the source virtual address.
The **clearbwp**=**NO** option is a performance enhancement, avoiding unnecessary instructions if you know you are starting on a page boundary or you are intending to divide by page-size anyway.

user_pagesize_addr
Location of the page-size value (returned by a call to the $GETSYI system service specifying the SYI$_PAGESIZE item descriptor) in the application data area.
If this argument is blank, the macro uses MMG$GL_PAGESIZE (bigpage) or MMG$C-VAX-PAGE-SIZE (vaxpage).

user_mask_addr
Location of the application-provided byte-within-page mask. If this argument is blank, the macro uses MMG$GL_BWP_MASK if **user_pagesize_addr** is also blank. Otherwise, it subtracts one from the contents of the **user_pagesize_addr** and uses that value.

quad=YES
If **YES**, the conversion supports 64-bit addressing. If **NO**, the conversion does not support 64-bit addressing.

$PAGES_TO_BYTES

Converts a page count to a byte count.

Format

$PAGES_TO_BYTES source_pagcnt, dest_bytcnt, leftshift, quad=YES

Parameters

source_pagcnt
Source page count.

dest_bytcnt
Destination of byte count.

leftshift
Location of application-provided value to shift (in place of multiply). This value is a function of the page size, as shown in Table D–1.

quad=YES
If **YES**, the conversion supports 64-bit addressing. If **NO**, the conversion does not support 64-bit addressing.

$PREVIOUS_PAGE

Computes the virtual address of the first byte in the previous page.

Format

$PREVIOUS_PAGE source_va, dest_va, clearbwp=NO, user_pagesize_addr, user_mask_addr, quad=YES

Parameters

source_va
Source virtual address.

dest_va
Destination of virtual address within previous page.

clearbwp=NO
If **YES**, masks the byte-within-page portion of the source virtual address.
The **clearbwp**=**NO** option is a performance enhancement, avoiding unnecessary instructions if you know you are starting on a page boundary or you are intending to divide by page-size anyway.

user_pagesize_addr
Location of the page-size value (returned by a call to the $GETSYI system service specifying the SYI$_PAGESIZE item descriptor) in the application data area.
If this argument is blank, the macro uses MMG$GL_PAGESIZE (bigpage) or MMG$C-VAX-PAGE-SIZE (vaxpage).

user_mask_addr
Location of the application-provided byte-within-page mask. If this argument is blank, the macro uses MMG$GL_BWP_MASK if **user_pagesize_addr** is also blank. Otherwise, it subtracts one from the contents of the **user_pagesize_addr** and uses that value.

quad=YES
If **YES**, the conversion supports 64-bit addressing. If **NO**, the conversion does not support 64-bit addressing.

$ROUND_RETADR

Rounds the range implied by the virtual addresses in a **retadr** array returned from a memory management system service to a range that is the factor of CPU-specific pages. The return value can be supplied as an **inadr** array in a subsequent call to another memory management system service.

Format

$ROUND_RETADR   retadr, full_range, user_mask_addr, direction=ASCENDING

Parameters

retadr
Address of array of two 32-bit addresses, typically returned from $CRMPSC or a similar service. This value can be in the form of either "label" or "(Rx)".

full_range
Output array of two longwords. FULL_RANGE[0] is **retadr**[0] rounded down to a CPU-specific page boundary, and FULL_RANGE[1] is **retadr**[1] rounded up to one less than a CPU-specific page boundary (that is, to the last byte in the page).

user_mask_addr
Location of application-provided byte-within-page mask. If this argument is blank, the macro uses MMG$GL_BWP_MASK on an OpenVMS Alpha system and VA$M_BYTE on an OpenVMS VAX system.

direction=ASCENDING
Direction of rounding. The keywords are defined in the following table:

ASCENDING	**retadr**[0] < **retadr**[1]
DESCENDING	**retadr**[1] < **retadr**[0]
UNKNOWN	Values are compared at run time, then proper rounding is performed

$START_OF_PAGE

Converts a virtual address to the address of the first byte within that page.

Format

$START_OF_PAGE   source_va, dest_va, user_mask_addr, quad=YES

Parameters

source_va
Source virtual address.

dest_va
Destination of virtual address of first byte within page.

user_mask_addr
Location of application-provided byte-within-page mask. If this argument is blank, the macro uses MMG$GL_BWP_MASK in OpenVMS Alpha systems and MMG$C-VAX-PAGE-SIZE - 1 (defined in $pagedef) in OpenVMS VAX systems.

quad=YES
If **YES**, the conversion supports 64-bit addressing. If **NO**, the conversion does not support 64-bit addressing.

D.2 Saving and Restoring Alpha Registers

Frequently, VAX MACRO source code must save and restore register values, either because that is part of the defined interface or the code requires work registers. On OpenVMS VAX, code may invoke any number of macros to do this. On OpenVMS Alpha, you cannot simply replace these macros with 64-bit pushes and pops to and from the stack, because there is no guarantee that the macro caller has a quadword-aligned stack. Instead, you should replace such macro invocations with $PUSH64 and $POP64 macros. These macros, located in STARLET.MLB, preserve all 64 bits of a register but use longword references to do so.

$POP64

Pops the 64-bit value on the top of the stack into an Alpha register.

Format

$POP64   reg

Parameters

reg
Register into which the macro places the 64-bit value from the top of the stack.

Description

$POP64 takes the 64-bit value at the top of the stack and places it in an Alpha register using longword instructions. This is to avoid using quadword instructions when an alignment fault should be avoided, but restoring all 64 bits is necessary.

$PUSH64

Pushes the contents of an Alpha 64-bit register onto the stack.

Format

$PUSH64 reg

Parameters

reg
Register to be pushed onto the stack.

Description

$PUSH64 takes an Alpha 64-bit register and puts it on the stack using longword instructions. This is to avoid using quadword instructions when an alignment fault should be avoided, but saving all 64 bits is necessary.

D.3 Locking Pages into a Working Set

Five macros are provided for locking pages into a working set. These macros reside in SYS$LIBRARY:LIB.MLB. For a complete description of how to use these macros, see Section 3.10.

Three macros are used for image initialization-time lockdown, and two macros are used for *on-the-fly* lockdown.

Note

If the code is being locked because the IPL will be raised above 2, where page faults cannot occur, make sure that the delimited code does not call run-time library or other procedures. The VAX MACRO compiler generates calls to routines to emulate certain VAX instructions. An image that uses these macros must link against the system base image so that references to these routines are resolved by code in a nonpageable executive image.

D.3.1 Image Initialization-Time Lockdown

The three macros for image initialization-time lockdown follow:

- $LOCK_PAGE_INIT
- $LOCKED_PAGE_END
- $LOCKED_PAGE_START

$LOCK_PAGE_INIT

Required in the initialization routines of an image that is using $LOCKED_ PAGE_START and $LOCKED_PAGE_END to delineate areas to be locked at initialization time.

Format

$LOCK_PAGE_INIT [error]

Parameters

[error]
Address to which to branch if one of the $LKWSET calls fail. If this address is reached, R0 reflects the status of the failed call, and R1 contains 0 if the call to lock the code failed, or 1 if that call succeeded but the call to lock the linkage section failed.

Description

$LOCK_PAGE_INIT creates the necessary psects and issues the $LWKSET calls to lock into the working set the code and linkage sections that were declared by $LOCKED_PAGE_START and $LOCKED_PAGE_END. R0 and R1 are destroyed by this macro.

The psects locked by this macro are $LOCK_PAGE_2 and $LOCK_LINKAGE_2. If code sections in other modules, written in other languages, use these psects, they will be locked by an invocation of this macro in a VAX MACRO module.

$LOCKED_PAGE_END

Marks the end of a section of code that may be locked at image initialization time by the $LOCK_PAGE_INIT macro.

Format

$LOCKED_PAGE_END [link_sect]

Parameters

[link_sect]
Psect to return to if the linkage psect in effect when the $LOCKED_PAGE_START macro was executed was not the default linkage psect, $LINKAGE.

Description

$LOCKED_PAGE_END is used with $LOCKED_PAGE_START to delineate code that may be locked at image initialization time by the $LOCK_PAGE_INIT macro. The code delineated by these macros must contain complete routines—execution cannot fall through either macro, nor can you branch into or out of the locked code. Any attempt to branch into or out of the locked code section or to fall through the macros will be flagged by the compiler with an error.

$LOCKED_PAGE_START

Marks the start of a section of code that may be locked at image initialization time by the $LOCK_PAGE_INIT macro.

Format

$LOCKED_PAGE_START

There are no parameters for this macro.

Description

$LOCKED_PAGE_START is used with $LOCKED_PAGE_END to delineate code that may be locked at image initialization time by the $LOCK_PAGE_INIT macro. The code delineated by these macros must contain complete routines—execution may not fall through either macro, nor may the locked code be branched into or out of. Any attempt to branch into or out of the locked code section or to fall through the macros will be flagged by the compiler with an error.

D.3.2 On-the-Fly Lockdown

The two macros for *on-the-fly* lockdown follow:

- $LOCK_PAGE
- $UNLOCK_PAGE

$LOCK_PAGE

Marks the beginning of a section of code to be locked *on the fly*.

Format

$LOCK_PAGE [error]

Parameters

[error]
Address to branch to if one of the $LKWSET calls fail.

Description

This macro is placed inline in executable code and must be followed by the $UNLOCK_PAGE macro. The $LOCK_PAGE/$UNLOCK_PAGE macro pair creates a separate routine in a separate psect. $LOCK_PAGE locks the pages and linkage section of this separate routine into the working set and JSRs to it. All code between this macro and the matching $UNLOCK_PAGE macro is included in the locked routine and is locked down.

All registers are preserved by this macro unless the error address parameter is present and one of the calls fail. If that happens, R0 reflects the status of the failed call. R1 then contains 0 if the call to lock the code failed or 1 if that call succeeded but the call to lock the linkage section failed.

If the ERROR parameter is used, the ERROR label must be placed outside the scope of the $LOCK_PAGE and $UNLOCK_PAGE pair. This is because the error routine is branched to before calling the subroutine that the $LOCK_PAGE and $UNLOCK_PAGE routines create.

Note that since the locked code is made into a separate routine, any references to local stack storage within the routine will have to be changed, as the stack context is no longer the same. Also, you cannot branch into or out of the locked code from the rest of the routine.

$UNLOCK_PAGE

Marks the end of a section of code to be locked *on the fly*.

Format

$UNLOCK_PAGE [error][,LINK_SECT]

Parameters

[error]
An error address to which to branch if one of the $ULKWSET calls fail.

[link_sect]
Linkage psect to return to if the linkage psect in effect when the $LOCK_PAGE macro was executed was not the default linkage psect, $LINKAGE.

Description

$UNLOCK_PAGE returns from the locked routine created by the $LOCK_PAGE and $UNLOCK_PAGE macro pair and then unlocks the pages and linkage section from the working set. This macro is placed inline in executable code after a $LOCK_PAGE macro.

All registers are preserved by this macro unless the error address parameter is present and one of the calls fail. If that happens, R0 reflects the status of the failed call. R1 then contains 0 if the call to unlock the code failed or 1 if that call succeeded but the call to unlock the linkage section failed.

If the **error** parameter is used, the **error** label must be placed outside the scope of the $LOCK_PAGE and $UNLOCK_PAGE pair. This is because the error routine is branched to after returning from the subroutine created by the $LOCK_PAGE and $UNLOCK_PAGE routines.

E MACRO-32 Macros for 64-Bit Addressing

This appendix describes the MACRO-32 macros for manipulating 64-bit addresses, for checking the sign extension of the low 32 bits of 64-bit values, and for checking descriptors for the 64-bit format.

These macros reside in the directory ALPHA$LIBRARY:STARLET.MLB (generally synonymous with SYS$LIBRARY:STARLET.MLB) and can be used by both application code and system code. The page macros have also been enhanced for 64-bit addresses. The support is provided by a new parameter, QUAD=NO/YES.

Note that you can use certain arguments to the macros described in this appendix to indicate register sets. To express a register set, list the registers, separated by commas, within angle brackets. For example:

```
<R1,R2,R3>
```

If the set contains only one register, the angle brackets are not required.

E.1 Macros for Manipulating 64-Bit Addresses

This section describes the following macros, designed to manipulate 64-bit addresses:

- $SETUP_CALL64
- $PUSH_ARG64
- $CALL64

$SETUP_CALL64

Initializes the call sequence.

Format

$SETUP_CALL64 arg_count, inline=true or false

Parameters

arg_count
The number of arguments in the call.

inline
Forces inline expansion, rather than creation of a JSB routine, when set to TRUE. If there are six or fewer arguments, the default is INLINE=FALSE.

Description

This macro initializes the state for a 64-bit call. It *must* be used before using $PUSH_ARG64 and $CALL64.

If there are six or fewer arguments, the code is always in line.

By default, if there are more than six arguments, this macro creates a JSB routine that is invoked to perform the actual call. However, if the inline option is specified as INLINE=TRUE, the code is generated in line. This option should be enabled *only* if the code in which it appears has a fixed stack depth. A fixed stack depth can be assumed if no RUNTIMSTK or VARSIZSTK messages have been reported. Otherwise, if the stack alignment is not at least quadword, there might be many alignment faults in the called routine and in anything the called routine calls. The default behavior (INLINE=FALSE) does not have this problem.

If there are more than six arguments, there can be no references to AP or SP between a $SETUP_CALL64 and the matching $CALL64, because the $CALL64 code may be in a separate JSB routine. In addition, temporary registers (R16 and above) may not survive the $SETUP_CALL64. However, they can be used *within* the range, except where R16 through R21 interfere with the argument registers already set up. In such cases, higher temporary registers should be used instead.

Note

The $SETUP_CALL64, $PUSH_ARG64, and $CALL64 macros are intended to be used in an inline sequence. That is, you cannot branch into the middle of a $SETUP_CALL64/$PUSH_ARG64/$CALL64 sequence, nor can you branch around $PUSH_ARG64 macros or branch out of the sequence to avoid the $CALL64.

$PUSH_ARG64

Does the equivalent of argument pushes for a call.

Format

$PUSH_ARG64 argument

Parameters

argument
The argument to be pushed.

Description

This macro pushes a 64-bit argument for a 64-bit call. The macro $SETUP_CALL64 *must* be used before you can use $PUSH_ARG64.

Arguments will be read as aligned quadwords. That is, $PUSH_ARG64 4(R0) will read the quadword at 4(R0), and push the quadword. Any indexed operations will be done in quadword mode.

To push a longword value from memory as a quadword, first move it into a register with a longword instruction, and then use $PUSH_ARG64 on the register. Similarly, to push a quadword value that you know is *not* aligned, move it to a temporary register first, and then use $PUSH_ARG64.

If the call contains more than six arguments, this macro checks for SP or AP references in the argument. If the call contains more than six arguments, SP references are not allowed, and AP references are allowed only if the inline option is used.

The macro also checks for references to argument registers that have already been set up for the current $CALL64. If it finds such references, a warning is reported to advise the user to be careful not to overwrite an argument register before it is used as the source in a $PUSH_ARG64.

The same checking is done for AP references when there are six or fewer arguments; they are allowed, but the compiler cannot prevent you from overwriting one before you use it. Therefore, if such references are found, an informational message is reported.

Note that if the operand uses a symbol whose name includes one of the strings R16 through R21, *not* as a register reference, this macro might report a spurious error. For example, if the invocation `$PUSH_ARG64 SAVED_R21` is made after R21 has been set up, this macro will unnecessarily report an informational message about overwriting argument registers.

Also note that $PUSH_ARG64 *cannot* be in conditional code. $PUSH_ARG64 updates several symbols, such as the remaining argument count. Attempting to write code that branches around a $PUSH_ARG64 in the middle of a $SETUP_ CALL64/$CALL64 sequence will not work properly.

$CALL64

Invokes the target routine.

Format

$CALL64 call_target

Parameters

call_target
The routine to be invoked.

Description

This macro calls the specified routine, assuming $SETUP_CALL64 has been used to specify the argument count, and $PUSH_ARG64 has been used to push the quadword arguments. This macro checks that the number of pushes matches what was specified in the setup call.

The call_target operand must not be AP- or SP-based.

E.2 Macros for Checking Sign Extension and Descriptor Format

The macros in this section are used for checking certain values and directing program flow based on the outcome of the check.

$IS_32BITS

Checks the sign extension of the low 32 bits of a 64-bit value and directs the program flow based on the outcome of the check.

Format

$IS_32BITS   quad_arg, leq_32bits, gtr_32bits, temp_reg=22

Parameters

quad_arg
A 64-bit quantity, either in a register or in an aligned quadword memory location.

leq_32bits
Label to branch to if quad_arg is a 32-bit sign-extended value.

gtr_32bits
Label to branch to if quad_arg is greater than 32 bits.

temp_reg=22
Register to use as a temporary register for holding the low longword of the source value—R22 is the default.

Description

$IS_32BITS checks the sign extension of the low 32 bits of a 64-bit value and directs the program flow based on the outcome of the check.

Examples

1. `$is_32bits  R9, 10$`

   In this example, the compiler checks the sign extension of the low 32 bits of the 64-bit value at R9 using the default temporary register, R22. Depending on the type of branch and the outcome of the test, the program either branches or continues in line.

2. `$is_32bits  4(R8), 20$, 30$, R28`

   In this example, the compiler checks the sign extension of the low 32 bits of the 64-bit value at 4(R8) using R28 as a temporary register and, based on the check, branches to either 20$ or 30$.

$IS_DESC64

Tests the specified descriptor to determine if it is a 64-bit format descriptor, and directs the program flow based on the outcome of the test.

Format

$IS_DESC   desc_addr, target, size=long or quad

Parameters

desc_addr
The address of the descriptor to test.

target
The label to branch to if the descriptor is in 64-bit format.

size=long
The size of the address pointing to the descriptor. Acceptable values are “long” (the default) and “quad”.

Description

$IS_DESC64 tests the fields which distinguish a 64-bit descriptor from a 32-bit descriptor. If it is in 64-bit form, a branch is taken to the specified target. The address to be tested is read as a longword, unless SIZE=QUAD is specified.

Examples

1. `$is_desc64 r9, 10$`

   In this example, the descriptor pointed to by R9 is tested, and if it is in 64-bit form, a branch to 10$ is taken.

2. `$is_desc64 8(r0), 20$, size=quad`

   In this example, the quadword at 8(R0) is read, and the descriptor it points to is tested. If it is in 64-bit form, a branch to 20$ is taken.

Index