HP OpenVMS MACRO コンパイラ ポーティングおよびユーザーズ・ガイド

注文番号: AA–PV64E–TE

2008 年 5 月

本書は，VAX MACROコンパイラ向けに書かれた Macro-32 ソース・コードを，HP OpenVMS Industry Standard 64 または HP OpenVMS Alpha システム上で動作する OpenVMS 用の MACRO コンパイラ向けに移植する方法について説明します。また，コンパイラの使用方法についても説明します。

改訂／更新情報: 本書は， OpenVMS Version 7.3 用の『OpenVMS MACRO-32 Porting and User's Guide』の改訂版です。

ソフトウェア・バージョン: OpenVMS I64 Version 8.2
OpenVMS Alpha Version 8.2

日本ヒューレット・パッカード株式会社

ZK5601

原典： HP OpenVMS MACRO Compiler Porting and User’s Guide

目次

例

表

まえがき

対象読者

本書は，アプリケーション・コードの移植を担当するソフトウェア・エンジニアを対象に，以下の移植について説明しています。

- VAX MACROから OpenVMS Alpha システム上の OpenVMS MACRO へ
- VAX MACROから OpenVMS I64 システム上の OpenVMS MACRO へ
- OpenVMS Alpha システム上の OpenVMS MACRO から OpenVMS I64 システム上の OpenVMS MACRO へ

読者は，プログラミングの技術を持ち，関係するオペレーティング・システムについて理解している必要があります。

本書の構成

本書は 2 つの部に分かれています。

第 1 部, 概念と方法は，以下の章で構成されます。

- 第 1 章, Macro-32 コードの移植の準備では，VAX MACROコードを OpenVMS Alpha または OpenVMS I64 システムに移植する方法について説明します。
- 第 2 章, MACRO コンパイラのプラットフォームごとの動作では，特別な指示文やマクロを含め，コンパイラの機能をいつどのように使用するかを説明します。
- 第 3 章, ソースに対する推奨される変更と必要な変更では， MACRO コンパイラでコンパイルできないコーディング構造を変更する方法について説明します。
- 第 4 章, 移植したコードの性能改善では，移植したコードの性能を向上させるために利用できるコンパイラの機能について説明します。
- 第 5 章, MACRO の 64 ビット・アドレッシングのサポートでは， MACRO コンパイラと関連する構成要素で提供される 64 ビット・アドレシングのサポートについて説明します。

第 2 部, リファレンスは，以下の付録で構成されます。

- 付録 A, MACRO コンパイラの修飾子
- 付録 B, 専用の指示文
- 付録 C, MACRO コンパイラ・ビルトイン

- 付録 D, VAX から Alpha または I64 への移植用のマクロ
- 付録 E, 64 ビット・アドレッシング用のマクロ

関連資料

本書では，一部の項目に関する追加情報として，以下のマニュアルを参照しています。

- OpenVMS Alpha から OpenVMS I64 へのアプリケーション・ポーティング・ガイドでは， OpenVMS Alpha システムから OpenVMS I64 システムにアプリケーションを移植する方法を説明しています。
- Migrating an Environment from OpenVMS VAX to OpenVMS Alpha[1]には，VAX から Alpha への移行手順と，移行に役立つ情報が記載されています。移行を計画する際に決定しなければならない事柄や，これらの判断を行うにあたって必要な情報を取得する方法について説明しています。また，移行の方法について説明し，それぞれの方法で必要な作業量を見積もり，アプリケーションにとって最適な方法を選択するために使用可能なツールについても説明しています。
- Migrating an Application from OpenVMS VAX to OpenVMS Alpha では，再コンパイルと再リンクによって OpenVMS VAX アプリケーションの OpenVMS Alpha 版を構築する方法について説明しています。アプリケーションのネイティブ OpenVMS Alpha 版を作成するにあたって変更が必要な， VAX アーキテクチャの機能 (ページ・サイズの想定，同期処理，条件処理など) への依存についての説明があります。また，アプリケーションの中で，ネイティブ OpenVMS Alpha コンポーネントと変換された OpenVMS VAX コンポーネントの間でやり取りする方法についても説明しています。
- OpenVMS Calling Standard では， OpenVMS VAX， OpenVMS Alpha，および OpenVMS I64 システムでプロシージャ呼び出しを可能とするメカニズムについて説明しています。
- VAX MACRO and Instruction Set Reference Manual[1]では， VAX 命令と標準のVAX MACROアセンブリ言語の指示文について説明しています。
- OpenVMS System Messages: Companion Guide for Help Message Users では， Help Message ユーティリティを使用して， MACRO アセンブラ・メッセージと MACRO コンパイラ・メッセージの情報を取得する方法について説明しています。

[1] このマニュアルはアーカイブ扱いになっています。このマニュアルの保守は行われておらず， OpenVMS ドキュメント・セットにも含まれていません。ただし，http://www.hp.com/go/openvms/docでオンラインで参照することができます。

第1部
概念と方法

1

Macro-32 コードの移植の準備

この章では， OpenVMS VAX システムから OpenVMS Alpha または OpenVMS Industry Standard 64 システムへの Macro-32 コードの移植を計画する際に使用できる手順について説明します。

この章の主なトピックは以下のとおりです。

- 第 1.1 節, MACRO コンパイラの機能
- 第 1.2 節, コンパイラとアセンブラの違い
- 第 1.3 節, OpenVMS VAX から OpenVMS Alpha または OpenVMS I64 への段階的な移植手順
- 第 1.4 節, OpenVMS Alpha から OpenVMS I64 への段階的な移植手順
- 第 1.5 節, 移植性のないVAX MACROのコーディング・スタイルの特定
- 第 1.6 節, 有効なコーディング規約の作成
- 第 1.7 節, 共通ソースの保守

注意

コンパイラの起動方法については，付録 A を参照してください。

1.1 MACRO コンパイラの機能

OpenVMS MACRO コンパイラは， OpenVMS VAX システム (VAX MACROアセンブラ) 向けに記述された Macro-32 ソース・コードをコンパイルして， OpenVMS Alpha システムや OpenVMS I64 システムで動作する機械語コードを生成します。

注意

このコンパイラは， Macro-32 **コードを** OpenVMS Alpha **システムおよび** OpenVMS I64 **システムに移植するために提供されています。新規に開発する場合は，中級言語または高級言語を使用することをお勧めします。**

変更なしでコンパイルできるVAX MACROコードもありますが，ほとんどのコード・モジュールでは， OpenVMS Alpha または OpenVMS I64 に移植する際に，エントリ・ポイント指示文の追加が必要となります。多くの場合それ以外の変更も必要となります。

最初のコンパイルでは，コンパイラによってモジュールの問題が数個しか検出されず，それらの問題を修正した後でさらに問題が検出されることがあります。これは，1 つの問題を解決したことによって，コンパイラがさらにコードを調べ，最初の問題の陰に隠れていたその他の問題を検出できるようになったためです。

コンパイラには，この手順を容易にするための機能が多数含まれています。以下のような機能があります。

- コンパイラが生成するメッセージの種類を制御したり，生成されたコードで VAX と同じ動作をさせるための修飾子。たとえば，/FLAG 修飾子を使用すると，コンパイラが報告する情報メッセージの種類を指定することができます。これらのメッセージの多くは， VAX アーキテクチャへの依存など，移植上の問題を示しています。/FLAG 修飾子で指定できるオプションには，アラインされていないスタックやメモリの参照の報告や，サポートされていない指示文の報告などがあります。/FLAG 修飾子についての詳細は，付録 A, MACRO コンパイラの修飾子を参照してください。
- ルーチンのエントリ・ポイントを示してコンパイラに指示する指示文や，コードのセクションに対して VAX と同じ動作を強制する指示文。たとえば.CALL_ ENTRY は，呼び出されるルーチンのエントリ・ポイントをコンパイラに対して宣言します。第 2.3 節，第 2.5 節，第 3 章では，コンパイラが特別な指示文を必要とする状況について説明しています (付録 B, 専用の指示文を参照)。
- OpenVMS Alpha プラットフォームでは， 64 ビット演算を実行する Alpha 命令と Alpha PALcode 命令にアクセスできるようにするためのビルトイン (PALcode は， Privileged Architecture Library code の略)。たとえば， EVAX_ADDQ を適切なオペランドとともに使用すると，クォドワード加算命令が実行されます (付録 C, MACRO コンパイラ・ビルトインを参照)。
- OpenVMS I64 プラットフォームの場合，特定の I64 命令にアクセスできるようにするためのビルトイン。また， Alpha から I64 へのコードの移植を支援するために， EVAX_ビルトインの多くが I64 用に再実装されています (付録 C, MACRO コンパイラ・ビルトインを参照)。

コンパイラは， 64 ビット・アドレスをサポートしています。これについては，第 5 章と付録 E に説明があります。 64 ビット・アドレスのサポートは， OpenVMS Alpha Version 7.0 で追加されました。このサポートは， OpenVMS Alpha や OpenVMS I64 でサポートされている高級言語を使用せずに，VAX MACROを使用して 64 ビットのアドレス空間にアクセスするのが望ましいまれなケース向けに提供されています。

1.2 コンパイラとアセンブラの違い

MACRO コンパイラは，アセンブラではなくコンパイラであることに注意してください。そのため，入力コードに完全に一致する出力コードを作成するわけではありません。最適化処理で，コードが移動，複製，削除され，命令がインターリーブされることがあります。さらに，移植されたコードに誤りがあった場合の動作は，それに対応する VAX コードの動作と同じにならないことがあります。これらの違いについては，以降の項で説明します。

1.2.1 コードの移動

OpenVMS Alpha システムや OpenVMS I64 システムでは，分岐予測が間違っていた場合のコストが高くなります。コンパイラは，モジュール内の最も可能性が高いコード・パスを判断し，そのコード・パスをつなげたコードを生成します。可能性が低いと判断されたコード・パスは，モジュールの後ろの方に移動されます。次の例を考えます。

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

この例で，コンパイラは HALT を可能性が低いコード・パスと判断し，また 2 つのコード・ストリームが 10$で再結合されないことを検出します。これらの条件により，この分岐が実行される可能性が高いと判断されます。次に，間にある命令をモジュールの後ろに移動し， BLBS 命令を，移動したコードに分岐する BLBC 命令に変更し，次のようにラベル 10$の位置からインライン・コードの生成を続行します。

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

コンパイラ指示文.BRANCH_LIKELY および.BRANCH_UNLIKELYを使用することで，コンパイラによる条件分岐の可能性の判断を変更することができます (第 4.2 節を参照)。

1.2.2 コードの複製

コンパイラは，小さいコード・セクションを何度も複製して，過剰な分岐をなくすことがあります。たとえば，次の VAX コードへの分岐をコンパイルするときに，コンパイラは ERROR1 へ分岐するそれぞれの場所に MOVL を複製し，直接 COMMON_ERROR に分岐するように変更することがあります。

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

1.2.3 コードの削除

コンパイラの最適化処理によって，一部の命令がコードの流れに無関係と判断されることがあります。そのような場合，命令は削除されます。このような例としては，次のように，後続の条件分岐がない CMP 命令や TST 命令があります。

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

この CMPB 命令を削除すると，命令の目的が，2 箇所のメモリ位置にアクセスし，ルーチンを呼び出す前にメモリ・ページをページ・インさせることだった場合には，問題が発生します。このような部分は，OpenVMS Alpha や OpenVMS I64 に移植する際には，VAX と Alpha または I64 のページ・サイズが異なるため，いずれにせよ変更が必要となる可能性があります。ページ・サイズの変更に加えて，次のように命令を MOVx命令に置き換える必要があります。

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

この 2 つの MOVB 命令では，2 つの異なるレジスタを対象としている点に注意してください。現在コンパイラは，レジスタに値をロードする命令の後で，一度も読み込まれることなくそのレジスタが上書きされる場合でも，その命令を削除しません。しかし，将来この最適化が実装される可能性があります。

注意

一般に，メモリの読み取り参照で実際にメモリがアクセスされる必要があるコードの存在は，慎重に調べる必要があります。現在または将来の最適化で，メモリ参照が移動または削除される可能性があるためです。

1.2.4 命令のインターリーブ

デフォルトで実行される命令のスケジューリング (第 4.3 節を参照) により，1 つの VAX 命令から生成された Alpha または Itanium の命令と，周りの VAX 命令から生成された Alpha または Itanium の命令がインターリーブされます。

1.2.5 予約オペランド・フォルト

VAX システムでは，一部のVAX MACRO命令で，特定のオペランドが必要な範囲にないと予約オペランド・フォルトが発生することがあります。たとえば，INSV などのビット操作命令では，サイズ・オペランドが 32 よりも大きいと，VAX システムは実行時予約オペランド・フォルトを生成します。

OpenVMS Alpha システムと OpenVMS I64 システムでは，範囲外のオペランドがコンパイル時の定数の場合は，コンパイラがこの条件を検出してエラー・メッセージを出力します。しかし，このオペランドが実行時の変数の場合は，コンパイラは実行時の範囲チェックを生成しません。オペランドが範囲外の場合，この命令の実行によって誤った結果が生成されますが，フォルトにはなりません。

1.3 OpenVMS VAX から OpenVMS Alpha または OpenVMS I64 への段階的な移植手順

以下の手順は，VAX MACROコードを OpenVMS Alpha または OpenVMS I64 に移植するための効率の良い方法であることが実証されています。

1. 移植する各モジュールを最初から最後まで調べ，正常な移植を阻害するようなコーディング・スタイルになっていないことを確認します。このような調査が必要なのは，VAX アーキテクチャについての広範な知識を利用した，無数の創意に富んだ用法を備えたVAX MACROコードを扱うことは，コンパイラにとって不可能なためです。このようなコードが発見されずにそのままになっていると，VAX MACROコードを OpenVMS Alpha または OpenVMS I64 に移植するという努力が根底から覆される可能性があります。
2. モジュールの各エントリ・ポイントに，適切なエントリ・ポイント指示文を追加してください。

   このような指示文には次のものがあります。

   - .CALL_ENTRY
   - .JSB_ENTRY
   - .JSB32_ENTRY
   - .EXCEPTION_ENTRY (Alpha システムのみ)

   .CALL_ENTRY 句を使用したい場合以外は，VAX MACROのエントリ・ポイント.ENTRY を.CALL_ENTRY に変更する必要はありません。また，ここでレジスタ引数を追加する必要もありません。

   これらの指示文を正しく配置するためのガイドラインについては，第 2.3 節を参照してください。

   それぞれの指示文の完全な構文の説明については，付録 B を参照してください。

3. コンパイラを起動してモジュールをコンパイルします。これを実行するために推奨されるコマンド・プロシージャを第 2.12 節に示します。

   アラインされていないスタックおよびメモリの参照，ルーチンの分岐，問題が起きる可能性がある命令，自己変更コードがあると，デフォルトでは，コンパイラがエラーを出力します。コマンド行で/FLAG=HINTS を指定すると，コンパイラは，手順 2 で挿入したエントリ・ポイント指示文のinputおよびoutputレジスタ引数に対する忠告を出力します。

4. コンパイラによって報告される問題に注目します。

   通常 MACRO コンパイラは， OpenVMS Alpha システムと OpenVMS I64 システムで同じメッセージを生成しますが，以下の接頭辞だけは例外です。

   - OpenVMS Alpha システム上の AMAC
   - OpenVMS I64 システム上の IMAC

   例

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

   または

   ```
   IMAC-I-ARGLISHOME, argument list must be homed in caller
   ```

   Help Message ユーティリティを呼び出して，表示された各コンパイラ・メッセージに対する説明やユーザの対処を参照することができます。

   Help Message ユーティリティについては， DCL ヘルプまたは OpenVMS System Messages: Companion Guide for Help Message Users を参照してください。また，第 1.5 節と第 3 章では， OpenVMS Alpha や OpenVMS I64 に直接変換できないVAX MACROのコーディング・スタイルについて詳細に説明しています。

   このパスで見つかった問題に対処することで，コンパイラは以降のパスでその他の問題を発見できるようになります。

5. Macro-32 ソース・コードを編集します。コンパイラによって指摘された問題を修正し，コンパイラが見逃した問題が他にないか探します。

   ただし，コンパイラが情報として出力する診断メッセージが出ないようにするためだけにコードを変更することはしないでください。情報レベルのメッセージのほとんどは，正常にコンパイルできるものの， OpenVMS Alpha や OpenVMS I64 では最適な性能が発揮できないコードを指摘するためにあります。ソース・コード中の問題のある命令の調査を終え，問題がなくなったと確信したら，コードの修正を終了します。コマンド行修飾子/FLAG と/WARN を使用して，診断メッセージの生成を制御できる点に注意してください。また，.ENABLE 指示文と.DISABLE 指示文を使用すると，モジュール内のコードの一部分で情報レベルのメッセージの表示をオフにすることができます。

6. 手順 2 で追加したエントリ・ポイント指示文に，input，output，preserve，およびscratch引数のうち適切なものを追加し，指定した各引数に該当するレジスタのリストを指定します。指定するレジスタの決定方法については，第 2.6 節を参照してください。
7. OpenVMS I64 システムの場合は，リンケージ指示文 (.CALL_LINKAGE，.DEFINE_LINKAGE) を追加し， R0/R1 以外のレジスタに値を返すルーチンと， Macro-32 以外の言語で記述された， JSB 命令で呼び出されるルーチンについて，コンパイラに指示します。また，ルーチンの戻りが標準的でない場合は，間接的な CALLS/CALLG 命令のそれぞれに.USE_LINKAGE 指示文を追加します。さらに，ルーチンが Macro-32 以外の言語で記述されている場合は，それぞれの間接 JSB 命令に.USE_LINKAGE 指示文を追加します。詳細は，付録 B を参照してください。
8. Macro-32 ソース・コードに対して，コンパイラから情報メッセージ以外が出力されず， OpenVMS Alpha または OpenVMS I64 の正しいオブジェクト・コードが生成されるようになるまで，手順 3 ～ 7 を繰り返します。
9. モジュールが OpenVMS VAX と OpenVMS Alpha または OpenVMS I64 システムで共通な場合は (第 1.7 節に示すコーディング規約を参照)，VAX MACROアセンブラとコンパイラの両方でエラーがなくなるまで移植作業は終わりではありません。

VAX MACROモジュールの移植に関してある程度経験を積むと，ソースを調べて問題を発見し，コンパイラを実行する前にそれを修正するのは，比較的容易になります。

1.4 OpenVMS Alpha から OpenVMS I64 への段階的な移植手順

すでに OpenVMS Alpha で動作しているVAX MACROコードを OpenVMS I64 に移植するのは，比較的簡単な作業です。 OpenVMS VAX からの移植と違い，ほとんどのVAX MACROコードは変更なしで再コンパイルすることができます。

OpenVMS Alpha と OpenVMS I64 の間の呼び出し規則の違いにより，コンパイラは以下の点について知る必要があります。

- JSB 命令または BSBW 命令を使用して，VAX MACROで記述されていないコードにジャンプするような MACRO コード
- CALLS 命令または CALLG 命令を使用して， R0 や R1 以外のレジスタに値を返すルーチンを呼び出す MACRO コード

R0/R1 以外のレジスタに値を返すルーチンと， JSB/BSBW 命令で呼び出される Macro-32 以外の言語で記述されたルーチンについてコンパイラに指示するために，リンケージ指示文 (.CALL_LINKAGE，.DEFINE_LINKAGE) を追加する必要があります。また，ルーチンが標準でない戻り方をする場合には，それぞれの間接 CALLS/CALLG 命令に.USE_LINKAGE 指示文を追加する必要があります。ルーチンが Macro-32 以外の言語で記述されている場合は，それぞれの間接 JSB/BSBW 命

令に.USE_LINKAGE 指示文を追加します。詳細は，付録 B, 専用の指示文を参照してください。

1.5 移植性のないVAX MACROのコーディング・スタイルの特定

OpenVMS Alpha または OpenVMS I64 のオブジェクト・コードにコンパイルしようとしているVAX MACROモジュールを調べるときには，移植性のないコーディング構造を探す必要があります。そのようなコードがモジュールにあると，そのようなコードがない場合と比較して移植に時間がかかります。コンパイラはこのようなコードの多くを検出できますが，自分で見つけることで移植作業が短縮できます。あまり頻繁に使用されず発見が難しいものも含め，移植性のない MACRO コーディング・スタイルの詳細については，第 3 章を参照してください。

以下の移植性のない MACRO コーディング・スタイルを探してください。

- 呼び出し元のさらに呼び出し元に戻るために，戻りアドレスをスタックから削除している場合 (第 3.3.4 項を参照)。以下に例を示します。

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

- スタック上に領域を割り当てるために，以下のような方法で戻りアドレスをスタックから一時的に削除している場合 (第 3.3.4 項を参照)。

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

または

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

- RSB 命令の戻りアドレスを設定するためなど，以下の例のようにラベルをスタックにプッシュしている場合 (第 3.3.3 項を参照)。

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

または

```
PUSHAL  routine_label
RSB
```

- フレーム・ポインタ (FP) の変更 (第 3.1.1 項を参照)。VAX MACROコードでフレーム・ポインタを変更する理由には，一般に次の 2 つがあります。
  - スタック上に呼び出しフレームを手動で作成するため。
  - フレーム・ポインタを使用して，.JSB_ENTRY ルーチン内で割り付けられたローカルなストレージを参照している場合。

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

- 以下の例のように REI ターゲットを構成している場合 (第 3.3.7 項を参照)。

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

または

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

または

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

- 以下の例のようにラベルとオフセットからなる飛び先に分岐している場合。VAX MACROコード中にこのようなコードがある場合，.CALL_ENTRY ルーチンの先頭のレジスタ保存マスクなどのように，コード・ストリーム中のいくつかのデータを超えた分岐を示しています (第 3.2.1 項を参照)。また，コードが VAX 命令のサイズを意識しており，それに依存していることを示していることもあります (第 3.2.3 項を参照)。

```
BRx     label+offset
```

- 以下の例のようにオペレーション・コードをスタックやデータ領域などに移動している場合。このようなコーディング・スタイルは，コードの生成またはコードの自己変更のどちらかを示し，第 3.2.2 項に示すように再設計が必要です。

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

または

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

- モジュールにまたがるジャンプ。アーキテクチャ上の要件により，コンパイラはモジュールにまたがったジャンプを JSB として扱う必要があります。そのため，次の例のような外部の分岐先は.JSB_ENTRY 指示文で宣言する必要があります (第 2.3.3 項を参照)。

```
JMP     G^external_routine
```

1.6 有効なコーディング規約の作成

第 1.3 節では，VAX MACROを OpenVMS Alpha や OpenVMS I64 に移植するための推奨される手順を説明しました。この手順に従うことでコードを効率よく移植することができますが，それだけでは移植を行う技術者の間で移植作業の一貫性が確保されるという保証がなく，移植されたコードを後からデバッグしてテストする必要がある技術者に分かりやすいという保証もありません。移植作業が均一に進み，ソース・コードの変更内容がきちんと文書化されるようにするためには，技術グループで，作業の目的に合ったコーディング規約を作成する必要があります。

当然，ツールの管理，ソース・コードの管理，共通コード，テストに関して従うべき手順を含め，技術グループが採用する方法論をグループの開発環境に合わせて調整する必要があります。技術グループが検討する必要があるコーディング規約には以下のものがあります。

- OpenVMS VAX システムおよび OpenVMS Alpha システムや OpenVMS I64 システムで共通なVAX MACROソース・モジュールの確立 (第 1.7 節を参照)。
- コンパイラのエントリ・ポイント指示文での，明確で一貫性のあるレジスタ宣言 (第 2.3 節を参照)。

1.7 共通ソースの保守

VAX MACROコードの移植作業を計画する際に， OpenVMS VAX システム，OpenVMS Alpha システム，および OpenVMS I64 システムで共通のソースを保守することのメリットを検討してください。技術グループにとっては，保守と拡張が必要なソースが 1 つですむというメリットがあり，また共通ソースはユーザ・インタフェースの共通化に役立ちます。ただし，条件付きのソース・コードが多くなりすぎてコードが分かりにくくなる場合は，アーキテクチャ固有の部分とアーキテクチャに依存しない部分に再構成してください。これらの部分が管理できないほど多くなったら，アーキテクチャ固有のモジュールを個別に作成してください。

1.7.1 コンパイラ指示文定義の追加

正常にコンパイルできると，ソース・ファイル中のVAX MACROコードは，VAX MACROアセンブラでも正常に処理されます。コードにコンパイラ指示文を追加した場合は，コードのアセンブル時にライブラリSYS$LIBRARY:STARLET.MLBによって解決されます。アセンブラは自動的にこのライブラリを検索し，未定義のマクロを探します。定義が見つかったら，アセンブラはコンパイラ指示文があったことを無視します。

ただし， OpenVMS VAX Version 6.0 以降を使用している場合は，まず指示文の定義を OpenVMS Alpha または OpenVMS I64 上の SYS$LIBRARY:STARLET.MLB から抽出し， OpenVMS VAX 上のSYS$LIBRARY:STARLET.MLBに挿入する必要があります。以下に例を示します。

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

コンパイラ指示文の定義の多くは，他のマクロを参照している点に注意してください。コード中で使用されているコンパイラ指示文のすべての定義だけでなく，関連するすべてのマクロを抽出し， OpenVMS VAX システム上の SYS$LIBRARY:STARLET.MLB に挿入してください。

1.7.2 VAX への依存の除去

Alpha または Itanium のコードに直接コンパイルできないコーディング・スタイルが含まれているために，ソース・ファイルを変更する必要がある場合でも，共通のVAX MACROソースから OpenVMS VAX および OpenVMS Alpha や OpenVMS I64 用のイメージを生成できることがあります。

同じコードが OpenVMS VAX および OpenVMS Alpha や OpenVMS I64 で動作するようにするために，このような VAX への依存を除去することで，移植作業の際に大きなメリットが生まれることがあります。 VAX 環境または Alpha 環境で動作するようにコードの移植を終えている場合は，その環境でモジュールを構築してテストすることで，移植したコードの単一のモジュールまたはモジュールのグループをデバッグできます。これによりデバッグ作業が大幅に短縮できます。

場合によっては，ソース・ファイルを条件付きにしたり (第 1.7.3 項を参照), OpenVMS VAX システムおよび OpenVMS Alpha システムや OpenVMS I64 システムで別のソースを提供するために，プロシージャを定義して実装しなくてはならないことがあります。

コードが内部モードで動作する場合は，共通のVAX MACROソースから OpenVMS VAX および OpenVMS Alpha や OpenVMS I64 のイメージを生成しようとしても，完全な成功は見込めません。内部モードのコードは，エグゼクティブと相互動作するため， OpenVMS VAX および OpenVMS Alpha や OpenVMS I64 の間のインタフェースおよびエグゼクティブ・データ構造の違いによる影響を受けやすくなっています。しかし，ユーザ・モードのコードは一般にアーキテクチャへの依存の影響を受けにくいため，より容易にコードの共通化ができます。

1.7.3 アーキテクチャ固有のコードの条件付け

VAX MACROコードを条件付きにするには， SYS$LIBRARY:ARCH_DEFS.MAR ファイルと以下のアーキテクチャ固有のシンボルを使用する必要があります。

- VAX プロセッサ上でVAX MACROソース・コードとともにアセンブルされるシンボル
- Alpha プロセッサ上でVAX MACROソース・コードとともにコンパイルされるシンボル
- Itanium プロセッサ上でVAX MACROソース・コードとともにコンパイルされるシンボル

コードのソース・モジュールを共通化し，アーキテクチャの種類による条件付きとする場合は，アセンブルおよびコンパイル時のコマンド行で ARCH_DEFS.MAR を追加して，.IF DF VAX，.IF DF ALPHA，.IF DF IA64を使用します。

以下のコード・パターンの使用は避け，移植済みのコード中にある場合は変更してください。

```
.IF DF VAX
  . . . ; VAX code
.IFF
  . . . ; Alpha code
  . . . ; will it work for IA64
.ENDC
```

または

```
.IF DF ALPHA
  . . . ; Alpha code
.IFF
  . . . ; VAX code, will likely fail for IA64
.ENDC
```

最も良いコードは次のものです。

```
.IF DF VAX
  . . . ; VAX code
.IFF
.IF DF ALPHA
  . . . ; Alpha code
.IFF
.IF DF IA64
  . . . ; IA64 code
.IFF
.ERROR  . . .
.ENDC
.ENDC
.ENDC
```

ARCH_DEFS.MAR ファイルは， OpenVMS Alpha および OpenVMS I64 で提供されています。 OpenVMS VAX では対応するファイルを作成する必要があります。以下に示すのは，作成するファイルの例です。

```
; 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
```

Alpha 版は SYS$LIBRARY に存在し，以下のシンボルの定義が含まれています。

- ALPHA— ソース・コードが Alpha アーキテクチャ固有であることを示します。
- BIGPAGE— ソース・コードが，可変メモリ・ページ・サイズを想定していることを示します。

I64 版は SYS$LIBRARY に存在し，次のシンボルの定義が含まれています。

- IA64— ソース・コードが Itanium アーキテクチャ固有であることを示します。

警告

OpenVMS Alpha や OpenVMS I64 の ARCH_DEFS.MAR で定義されているその他のシンボルは OpenVMS Alpha または OpenVMS I64 のソース・コード固有であり，すべてのリリースに含まれているという保証はありません。

2

MACRO コンパイラのプラットフォームごとの動作

OpenVMS システムの MACRO コンパイラは，VAX MACROソース・コードを OpenVMS VAX から OpenVMS Alpha や OpenVMS I64 に容易に移植できるように設計されています。本書で説明する制限やガイドラインに従ったVAX MACROコードであれば，コンパイラは，オリジナルのVAX MACROソースの意味を保ち，OpenVMS Alpha または OpenVMS I64 の呼び出し規則に従った， OpenVMS Alpha または OpenVMS I64 の正しいオブジェクト・モジュールを生成します。

この章のトピックは以下のとおりです。

- 第 2.1 節, Alpha と Itanium のレジスタの使用
- 第 2.2 節, Itanium アーキテクチャ，呼び出し規則，レジスタ・マッピング
- 第 2.3 節, ルーチンの呼び出しと宣言
- 第 2.4 節, CALL エントリ・ポイントの宣言
- 第 2.5 節, JSB ルーチンのエントリ・ポイントの宣言
- 第 2.6 節, ルーチンのレジスタ使用の宣言
- 第 2.7 節, ローカル・ルーチン間での分岐
- 第 2.8 節, 例外エントリ・ポイントの宣言 (OpenVMS Alpha のみ)
- 第 2.9 節, パック 10 進数命令の使用
- 第 2.10 節, 浮動小数点命令の使用
- 第 2.11 節, VAX の不可分性と細分性の維持
- 第 2.12 節, コンパイルとリンク
- 第 2.13 節, デバッグ

2.1 Alpha と Itanium のレジスタの使用

OpenVMS Alpha システムと OpenVMS I64 システムには， 32 個の整数レジスタ R0 ～ R31 があります。 OpenVMS Alpha コンパイラによって生成されたコードは，Alpha のレジスタ R0 ～ R12 を， VAX のレジスタであるかのように使用します。また， OpenVMS I64 コンパイラは， VAX のレジスタを，対応する Itanium のレジスタに自動的にマッピングします。そのため，VAX MACROコードのこれらレジスタを使用している箇所 (たとえば， JSB ルーチンに対する入力として使用している場合など) は，変更しなくても正しくコンパイルされます。VAX MACROの命令 (MOVL や ADDL など) は，演算に関係する Alpha または Itanium のレジスタの下位 32 ビット

を使用します。コンパイラは，符号拡張された 64 ビットの値をレジスタに保持します。

レジスタ R13 以降も， OpenVMS Alpha または OpenVMS I64 のオブジェクト形式にコンパイルされるVAX MACROコードで使用できます。これらのレジスタを使用する場合には，以下の制限を確認してください。

- 通常，既存のVAX MACROコードはレジスタ R0 ～ R11 だけを使用し， R12 ～ R14 は，それぞれ引数ポインタ (AP)，フレーム・ポインタ (FP)，スタック・ポインタ (SP) として定義されています。コンパイラは， AP， FP， SP の正規の使用を， VAX のレジスタと同様の機能を持つ Alpha または I64 のレジスタへの参照としてコンパイルします。VAX MACROソースが AP をスクラッチ・レジスタとして参照している場合は，コンパイラはこの参照を R12 に対する参照に変換します。これが望ましくない場合は，別のスクラッチ・レジスタを使用するようにコードを変更する必要があります。
- コンパイラがVAX MACROソース内で R12， R13， R14 への参照を検出すると，AP， FP， SP に相当する Alpha または I64 のレジスタへの参照に変換しません。代わりに，対応する Alpha の整数レジスタへの参照と見なします。
- R13 以降を使用するコードは，VAX MACROアセンブラでアセンブルできません。そのため， OpenVMS Alpha または OpenVMS I64 の条件付きとするか，OpenVMS Alpha または OpenVMS I64 専用のモジュールにします。
- ビルトイン EVAX_LDQ と EVAX_STQ を使用することで，レジスタの 64 ビット全体にアクセスできます。この機能については，付録 C を参照してください。
- OpenVMS Calling Standard で規定されているように，特定の使用目的を持ったレジスタを参照する場合には，特に注意する必要があります。
- OpenVMS Calling Standard で規定されているように， R16 以降のレジスタはスクラッチ・レジスタです。ルーチン呼び出しにまたがって値が保持されると仮定しないでください。
- ルーチンのソース・コードでレジスタ R13 以降が使用されていない場合，コンパイラはこれらのレジスタを一時的なレジスタとして使用します。 R13 ～ R15 を使用する場合は，保存と復元が行われます。
- Alpha システムでは，保存したレジスタはスタックに書き込まれます。 I64 システムでは，保存したレジスタはレジスタ 32 ～ 127 にコピーされます。

2.2 Itanium アーキテクチャ，呼び出し規則，レジスタ・マッピング

Itanium アーキテクチャと OpenVMS I64 には， Alpha アーキテクチャと OpenVMS Alpha に比べて重要な違いがいくつかあります。

OpenVMS I64 システムでは， 32 個の整数レジスタ R0 ～ R31 があり， R0 は読み取り専用のレジスタで， 0 を保持しています。これに対し OpenVMS Alpha では， R31 は読み書きレジスタで， 0 を保持しています。

また， OpenVMS I64 の呼び出し規則は， Intel の呼び出し規則と互換性を持つように作成されており， OpenVMS Alpha の呼び出し規則とは大きく異なっています。たとえば， OpenVMS I64 の標準の戻りレジスタは R8/R9 ですが， OpenVMS Alpha では R0/R1 です。 OpenVMS I64 の呼び出し規則では， R1 は GP (グローバル・ポインタ) として予約されており，標準化された FP (フレーム・ポインタ) が含まれていません。また，呼び出しにまたがって保持されるのは R4 ～ R7 だけです。これに対し OpenVMS Alpha では， R2 ～ R15 が保持されます。

Macro-32 ソース・コードは， OpenVMS VAX と OpenVMS Alpha の呼び出し規則を念頭に記述されているため，既存のコードを OpenVMS I64 のコンパイラで変更なしにコンパイルできるようにするために，コンパイラはいくつかの変換を行います。

まず，コンパイラは Macro-32 ソース・プログラムのレジスタを I64 ハードウェア上の異なるレジスタにマッピングします。これにより既存のプログラムが“MOVL SS$_NORMAL, R0”を使用できるようになり，生成されたコードが呼び出し規則どおりに R8 に値を返すようになります。表 2–1 に， OpenVMS VAX/OpenVMS Alpha から OpenVMS I64 へのレジスタのマッピングを示します。

表 2–1 OpenVMS VAX/OpenVMS Alpha から OpenVMS I64 へのレジスタ・マッピング

ソース・コード中の OpenVMS VAX/OpenVMS Alpha のレジスタ	生成されたコードで使用される OpenVMS I64 のレジスタ
R0	R8
R1	R9
R2	R28
R3	R3
R4	R4
R5	R5
R6	R6
R7	R7
R8	R26
R9	R27
R10	R10
R11	R11
R12	R30
R13	R31
R14	R20
R15	R21
R16	R14
R17	R15
R18	R16
R19	R17
R20	R18
R21	R19
R22	R22
R23	R23
R24	R24
R25	R25
R26	Itanium スタック汎用レジスタ
R27	Itanium スタック汎用レジスタ
R28	Itanium スタック汎用レジスタ
R29	R29
R30	R12
R31	R0

レジスタのマッピングは，呼び出しにまたがって保護されるレジスタ，呼び出しにまたがって保護されないレジスタ，呼び出し時や戻り時に値が変わる揮発性レジスタなど，レジスタの特性に基づいて慎重に選択されています。

OpenVMS Alpha では， AP に対する Macro-32 の参照は，引数がスタックに格納されているかどうかに応じて，コンパイラによって適切な場所にマッピングされます。FP への参照をサポートするため，コンパイラは必要に応じて FP の値を作成します。OpenVMS VAX および OpenVMS Alpha と同様に条件ハンドラを確立するために，コンパイラは 0(FP) への参照をサポートします。

本書の他の箇所に記載されているレジスタ番号は，その番号の VAX/Alpha レジスタを参照しています。実際の Itanium レジスタへのマッピングは， Macro-32 ソース・コード (およびコンパイラの大半の部分) では意識する必要がありません。

コンパイラには，マッピング・テーブルを介さずに Itanium レジスタに直接アクセスするための構文がありません。

コンパイラによる自動的なレジスタのマッピングにより，ほとんどの Macro-32 プログラム (Alpha のレジスタ R16 ～ R31 にアクセスするものを含む) は，変更なしでコンパイルできます。

ただし， OpenVMS Alpha でのルーチン・パラメータとしてレジスタ R16 ～ R21 を使用している場合は， OpenVMS I64 へのポータビリティがありません。 CALL にパラメータを渡すには PUSHL を使用し，呼び出された側のルーチンでは 4(AP), 8(AP) などを使用してそのパラメータを参照してください。コンパイラは， VAX のオペランドで暗黙的に使用されているスタック参照の代わりに，正しいレジスタ参照を生成します。

OpenVMS I64 システムでは， OpenVMS Alpha システム上で Alpha の命令に直接アクセスするためのビルトイン EVAX_* の多くが引き続きサポートされます。これらのビルトインでは，同じ論理演算を行う Itanium の命令が生成されます。 I64 システムでもサポートされる EVAX_*ビルトインの一覧については，付録 C を参照してください。

2.3 ルーチンの呼び出しと宣言

OpenVMS Calling Standard では， OpenVMS VAX システム， OpenVMS Alpha システム，および OpenVMS I64 システムに対して，大きく異なる呼び出し規約が規定されています。

OpenVMS VAX システムでは， 2 種類の呼び出し形式 CALL と JSB があり， 5 つの命令 CALLS， CALLG， JSB， BSBW，および BSBB があります。 CALL 命令は，戻り情報，保存されたレジスタ，その他のルーチン情報が格納されるフレームをスタック上に作成します。 CALL 命令に対するパラメータは，スタック上 (CALLS) またはその他の場所 (CALLG) にある連続したロングワード・メモリ位置に格納して渡されます。 JSB 命令では，戻りアドレスがスタック上に格納されます。どちらの呼び出し形式でも，呼び出し命令のハードウェア処理によってこれらすべての機能が提供されます。

OpenVMS Alpha システムでは，呼び出し形式は 1 つしかなく，サブルーチン呼び出し命令も JSR だけです。パラメータを受け取るルーチンでは，最初の 6 つのパラメータが R16 ～ R21 で渡され，パラメータの個数が R25 で渡されます。残りのパラメータは，スタック上でクォドワードとして渡されます。JSR 命令のハードウェアによる実行では，単に制御をサブルーチンに渡し，指定されたレジスタに戻りアドレスを格納します。スタック領域を割り当てることも，呼び出しフレームを作成することも，パラメータ操作を行うこともしません。これらすべての機能は，呼び出し側のルーチンで行うか，呼び出される側のルーチンで行う必要があります。

OpenVMS I64 システムでは，呼び出し形式は 1 つしかなく，サブルーチン呼び出し命令も br.call だけです。パラメータを受け取るルーチンでは，最初の 8 つのパラメータは R32 ～ R39 で渡され，パラメータの個数が R25 で渡されます。残りのパラメータは，スタック上でクォドワードとして渡されます。(OpenVMS I64 上の MACRO コンパイラでは，レジスタ・スタックを直接操作することはできず，R31 より大きな番号のレジスタにアクセスするための構文もありません。) br.call 命令のハードウェアによる実行では，単に制御をサブルーチンに渡し，指定されたレジスタに戻りアドレスを格納し，ハードウェア・レジスタ・スタックを調整します。スタック領域を割り当てることも，呼び出しフレームを作成することも，パラメータ操作を行うこともしません。これらすべての機能は，呼び出し側のルーチンで行うか，呼び出される側のルーチンで行う必要があります。

コンパイラは，OpenVMS Calling Standard に準拠しつつ，さまざまなVAX MACRO命令の機能をエミュレートするコードを生成する必要があります。そのためには，呼び出し命令で入力パラメータを操作するコードと，ルーチン自体のエントリ・ポイントでスタック・フレームなどのコンテキストを作成するための特別なコードが必要になります。

コンパイラは，宣言されたルーチンの一部となっているソース命令に対するコードだけを生成します。コンパイラが正しいリンケージ情報と正しいルーチン・コードを生成するように，VAX MACROソースのエントリ・ポイントにコンパイラ指示文を挿入する必要があります。

2.3.1 リンケージ・セクション (OpenVMS Alpha のみ)

Alpha システムでは，すべての外部 (モジュール外) 参照は**リンケージ・セクション**を通じて行われます。リンケージ・セクションは，以下のものが格納されたプログラム・セクション (psect) です。

- 外部変数のアドレス
- コード・ストリームに直接格納できない大きな定数
- リンケージ・ペア
- プロシージャ記述子

リンケージ・ペアは，外部モジュールを呼び出すときに使用するデータ構造です。リンケージ・ペアには，呼び出されるプロシージャ記述子のアドレスと，エントリ・ポイント・アドレスが格納されます。リンケージ・ペアは呼び出し側のリンケージ・セクションにあります。そのため，あるルーチンに対して，呼び出し側のリンケージ・セクションに 1 つずつ，いくつものリンケージ・ペアがあります。モジュール・ローカルな呼び出しの場合は，呼び出される側のフレーム記述子をコンパイラが直接参照できるため，リンケージ・ペアは使用されません。

プロシージャ記述子は，レジスタとスタックの使用など，ルーチンに関する基本的な情報を提供するデータ構造です。各ルーチンには固有のプロシージャ記述子があり，リンケージ・セクションの中にあります。通常，プロシージャ記述子は，実行時にはアクセスされません。主に，例外が発生した場合や，デバッガなどによって，フレームを解釈するために使用されます。

これらを含む各種のデータ構造についての詳細は， OpenVMS Calling Standard を参照してください。

2.3.2 プロローグ・コードとエピローグ・コード

VAX のサブルーチンの動作をまねるために，コンパイラは各ルーチンの入口と出口でコードを生成する必要があります。このコードは， VAX ハードウェアによって実行される機能をエミュレートし，プロローグ・コードおよびエピローグ・コードと呼ばれます。

プロローグ・コードでは，コンパイラは，必要なコンテキストを保存するために，スタック領域 (および OpenVMS I64 システムではスタック汎用レジスタ) を割り当てる必要があります。これには，ルーチンが保護しなければならないレジスタの保存と，戻りアドレスの保存が含まれます。 CALL ルーチンの場合は，新しいスタック・フレームの作成と，フレーム・ポインタ (FP) の変更も含まれ (OpenVMS Alpha システムと OpenVMS I64 システムで必要に応じて)，さらに入力パラメータの操作と保存が含まれることもあります (第 2.4 節を参照)。

エピローグ・コードでは，コンパイラは必要なレジスタ，スタック・フレーム情報，戻りアドレスを復元し，スタックを元の位置まで切り捨てます。

2.3.3 エントリ・ポイントの宣言が必要な場合

CALLS， CALLG， JSB， BSBW， BSBB 命令のいずれかのターゲットとして使用される可能性があるコード・ラベルは，エントリ・ポイントとして宣言する必要があります。また，以下の条件に該当するコード・ラベルは，すべて.JSB_ENTRY 指示文または.JSB32_ENTRY 指示文を使用して，エントリ・ポイントとして宣言する必要があります。

- ラベルが，グローバルな (モジュールにまたがった) JMP 命令， BRB 命令，または BRW 命令のターゲットとなる可能性がある場合。

- ラベルが，不確定な分岐 (BRB @(R10)など) のターゲットとなる可能性があり，参照とラベルが同じモジュール内にある場合でもラベルのアドレスが R10 に格納されている場合。
- 現在のモジュールからラベルがアクセスされるかどうかにかかわらず，ラベルのアドレスがレジスタまたはメモリ位置に格納される場合。

I64 と Alpha の OpenVMS の呼び出し規則では，不確定なコード・アドレスに直接アクセスする手段は提供されていません。そのようなアクセスは，すべてルーチンとコード・アドレスを記述するプロシージャ記述子を通じて実現されます。コード・ラベル・アドレスが格納されるときには，コンパイラはそのアドレスが現在のモジュールからだけ参照されるのか，別の MACRO モジュールや別の言語で記述された別のモジュールからも参照されるのかが分かりません。ソース命令がコード・アドレスを格納するときには必ず， MACRO コンパイラは，他のコードがそのコードに正しくアクセスできるように，代わりにそのコード・アドレスのプロシージャ記述子アドレスを格納します。プロシージャ記述子が存在するために，ラベルをエントリ・ポイントとして宣言する必要があります。格納されているアドレスを分岐先として使用する際，コンパイラはそのアドレスがどこから来たかを知らないため，格納されているアドレスを常にプロシージャ記述子のアドレスであると見なして，その記述子を使用してルーチンに制御を渡します。

OpenVMS I64 システムは， OpenVMS Alpha システムと同じように振る舞いますが，唯一の違いは，呼び出し規則では用語“関数記述子”が使用され，関数記述子はリンカによって作成される点です。

2.3.4 ルーチンのエントリ・ポイントを指定するための指示文

STARLET.MLB 内のマクロは，ルーチンにエントリ・ポイントを指定するための指示文を生成します。これら各マクロの形式についての詳細は，付録 B を参照してください。

OpenVMS VAX システム用にアセンブルした場合，.CALL_ENTRY 以外のマクロはヌルです。 OpenVMS Alpha システムまたは OpenVMS I64 システム用にコンパイルした場合，マクロは，引数を確認しコンパイラ指示文を生成するために展開されます。そのため，以下のマクロを共通のソース・モジュールで使用できます。

- .CALL_ENTRY 指示文は， Macro-32 コード中の.ENTRY 指示文を論理的に置き換えます。ただし，いずれかの.CALL_ENTRY 句を使用する場合以外は，.ENTRY 指示文を置き換える必要はありません。.ENTRY 指示文を置き換える場合は，引数が一致しないため，エディタで一括置換できない点に注意してください。

注意

.ENTRY 指示文は OpenVMS Alpha や OpenVMS I64 の.CALL_ENTRY 指示文に変換されるため，.ENTRY で宣言されたルーチンで変更されるレジスタは

自動的に保護されます。これは，VAX MACROの動作と異なります。第 2.4.2 項を参照してください。

- .JSB_ENTRY 指示文は， JSB， BSBB， BSBW のいずれかの命令で呼び出されるルーチンを示します。.JSB32_ENTRY 指示文は， Alpha または Itanium のレジスタ値の上位 32 ビットの保護が不要な特別な環境で使用されます。

  .JSB_ENTRY 指示文は，他のモジュールからの JMP 命令または分岐命令のターゲットとなるエントリ・ポイントを宣言するためにも使用する必要があります。これは，このようなモジュールにまたがった分岐は，コンパイラによって JSB として実装されるためです。

  JMP (R0) のように保存されたコード・アドレスへの分岐も JSB 命令として扱われます。そのため，ターゲットとなる可能性のあるエントリ・ポイントを.JSB_ENTRY で宣言する必要があります。コード・ラベルを保存しようとした場合や，このような分岐を使用した場合に，コンパイラはこれに関する警告を表示します。

- .EXCEPTION_ENTRY 指示文 (OpenVMS Alpha のみ) は，例外サービス・ルーチンを指定します。

- .CALL_LINKAGE 指示文 (OpenVMS I64 のみ) は，名前付きリンケージまたは匿名リンケージをルーチン名に関連付けます。コンパイラが，ルーチン名をターゲットとする CALLS， CALLG， JSB， BSBB， BSBW のいずれかの命令を見つけると，関連付けられているリンケージを使用して，呼び出しの前後で保存と復元が必要なレジスタを決定します。

- .USE_LINKAGE 指示文 (OpenVMS I64 のみ) は，一時的な名前付きリンケージまたは匿名リンケージを確立します。これは，コンパイラによって，字句解析の順で次に処理される CALLS， CALLG， JSB， BSBB， BSBW 命令に対して使用されます。

- .DEFINE_LINKAGE 指示文 (OpenVMS I64 のみ) は，以降の.CALL_LINKAGE 指示文または.USE_LINKAG 指示文で使用される名前付きリンケージを定義します。

これらの指示文についての詳細は，付録 B を参照してください。

2.3.5 ルーチン呼び出しのコード生成

VAX の JSB， BSBW， BSBB 命令のコード生成は，パラメータ操作が不要のため非常に単純です。しかし， CALLS 命令と CALLG 命令のコード生成では， OpenVMS VAX の呼び出し規則のパラメータ処理を， OpenVMS Alpha または OpenVMS I64 の呼び出し規則に変換する必要があるため，より複雑です。

引数の個数が固定の CALLS 命令を処理する際には，コンパイラはルーチンにパラメータを渡すためのスタックへのプッシュを自動的に検出し， OpenVMS Alpha の呼び出し規則または OpenVMS I64 の呼び出し規則で使用されているレジスタに，パラメータを直接移動するコードを生成します。

VAX で次の呼び出しがあるとします。

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

VAX でのこの呼び出しは， OpenVMS Alpha では次のようにコンパイルされます。

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

VAX でのこの呼び出しは， OpenVMS I64 では基本的に次のようにコンパイルされます。

```
sxt4    r46 = r28                 ; Load 2nd output register
adds    r45 = 1, r0               ; Load 1st output register
adds    r25 = 2, r0               ; 2 parameters
adds    r12 = -16, r12            ; Extra octaword per calling standard
br.call.sptk.many br0 = XYZ ;;    ; Call the routine
adds    r12 = 16, R16             ; Pop extra octaword
mov     r1 = r32                  ; Restore saved GP
```

生成されるコードは， OpenVMS Alpha と OpenVMS I64 の呼び出し規則の違いに合わせるために，正確には異なる可能性があります。

引数の数が可変の CALLS 命令または CALLG 命令の場合，コンパイラは，引数リストを OpenVMS Alpha 形式または OpenVMS I64 形式にアンパックするエミュレーション・ルーチンを呼び出す必要があります。これには 2 つの副作用があります。 OpenVMS VAX システムでは， CALLG 命令が指す引数リストの一部がアクセス不能でも，呼び出されたルーチンが引数リストのその部分にアクセスしなければ，ルーチンはアクセス違反にならずに正常に完了します。 OpenVMS Alpha システムと OpenVMS I64 システムでは，ルーチンを呼び出す前に引数リスト全体が処理されるため，引数リストのいずれかの部分がアクセス不能の場合， CALLG は常にアクセス違反となります。また，引数の個数が可変な CALLG 命令または CALLS 命令の引数の個数が 255 を超えている場合，エミュレートされた呼び出しがエラー状態を返し，ターゲット・ルーチンは呼び出されません。

2.4 CALL エントリ・ポイントの宣言

OpenVMS VAX， OpenVMS Alpha， OpenVMS I64 の呼び出し規則が違うため (第 2.3 節を参照)，コンパイラは AP 相対の VAX パラメータ参照をすべて，以下のように変換する必要があります。

- 呼び出されるルーチン内の AP 相対のパラメータ参照を， OpenVMS Alpha または OpenVMS I64 の新しいパラメータ位置に対する，レジスタおよびスタックの直接参照に変換します。
- AP 相対の参照のうち，パラメータ・リストへの別名参照となるものや，コンパイル時に解決できない変数オフセットを使用しているものを検出します。コンパイラは，クォドワード・レジスタとスタック引数をスタック上のロングワード引数リストにパックすることで， VAX の引数リストをまねます。.CALL_ENTRY 指示文に対する 2 つの引数は，このために用意されています (第 2.4.1 項を参照)。

後者の場合，この方法は**引数リストのホーミング**と呼ばれます。その結果得られる**ホーミングされた引数リスト**は，スタック・フレーム中の固定位置にあるため，すべての参照は FP ベースとなります。

2.4.1 ホーミングされた引数リスト

コンパイラが引数を自動的にホーミングする AP 参照の例としては，以下のものがあります。

- AP または AP ベースのアドレスの別レジスタへの保存
- AP または AP ベースのアドレスのパラメータとしての引き渡し
- パラメータを参照するための AP への可変オフセットの加算
- パラメータを参照するための AP 相対の可変インデックスの使用
- 6(AP) のように，引数リスト中でのロングワードにアラインされていないオフセットの使用

コンパイラは，スタック上のプロシージャ・フレームの固定的な一時領域に，ホーミングされた引数リストを設定します。

必須ではありませんが，.CALL_ENTRY 指示文に対する 2 つの引数home_args=TRUEおよびmax_args=nを使用して，ホーミングを指定することができます。

引数max_args=nは，コンパイラが固定的な一時領域に割り当てるロングワードの最大数を表します。max_args=nを使用する主な理由は，ソース・コード中で明示的に使用されている数よりも多くの引数をプログラムが受け取った場合に備えることです。home_args=TRUEを必要としないルーチンにこのパラメータを指定する一般的な理由は， AP を参照する JSB ルーチンを呼び出すためです。このような参照では，

コンパイラは引数リストが最後の.CALL_ENTRY ルーチンでホーミングされていると仮定し，FP ベースの参照を使用します。この仮定は常に正しいとは限らないため，.JSB_ENTRY ルーチンの内部で AP が使用されると，コンパイラは情報メッセージを報告します。

コンパイラが報告する情報メッセージを抑制したい場合は，両方の引数を使用します。

引数のうちのどちらかまたは両方を省略した場合，ホーミングが必要なコードをコンパイラが見つけると，コンパイラの処理内容を示すメッセージが表示されます。これを参考に，引数を追加したり，max_args引数の値を変更するなど，コードに対して必要な変更を行います。

home_args=TRUEだけを指定すると，次のメッセージが表示されます。

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

このメッセージが表示されるのは，ホーミングを明示的に指示しているにもかかわらず，max_argsで引数の数を指定していないためです。明示的な引数参照が見つからない場合は，OpenVMS Alpha システムでは 6 個，OpenVMS I64 システムでは 8 個のロングワードが割り当てられます。明示的な引数参照が見つかった場合は，見つかった最大の引数参照と同じ数のロングワードが割り当てられます。(このメッセージは，ホーミング対象としてコンパイラが見つけた引数の数よりも小さな数をmax_argsの値として指定した場合にも出力されます。)

max_argsだけを指定した場合，ホーミングが必要な参照が見つかると，次のメッセージが出力されます。

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

どちらの引数も指定していないときに，ホーミングが必要な参照が見つかると，次のメッセージが出力されます。

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

ここでnは，コンパイラが検出した，参照されている引数の最大数を表します。

2.4.2 変更されたレジスタの保存

仕様どおりに作成されたVAX MACRO CALL ルーチンは，引数リストを通じてすべてのパラメータを渡し，必要なすべてのレジスタを.ENTRY 宣言のレジスタ・マスクを通じて保存します。ただし，ルーチンによってはこれらの規則に従わず，レジスタでパラメータを渡したり，ルーチン内で PUSHL や POPL などの命令を使用して，必要なレジスタの内容の保存や復元を行っている場合があります。

OpenVMS Alpha システムや OpenVMS I64 システムでは， PUSHL や POPL を使用してレジスタの保存や復元を行っても，レジスタの下位 32 ビットしか保存されないため，これらの命令を使用するだけでは不十分です。レジスタの破壊を防ぐために，コンパイラは，レジスタ保存マスクで指定されたレジスタに加え，ルーチン内で変更されるすべてのレジスタ (R0 と R1 を除く) の 64 ビット全体の値を自動的に保存および復元します。これは，呼び出されるルーチンからの出力値を返すために任意のレジスタを使用しても，その目的は達成できないことを意味します。出力は標準の引数リストで渡すか，出力するレジスタを，.CALL_ENTRY 宣言のoutputパラメータで宣言する必要があります (第 2.6 節を参照)。 OpenVMS I64 システムでは，このようなルーチンを呼び出す場合，.CALL_LINKAGE 指示文を使用する必要があります。I64 の呼び出し規則では，呼び出しの前後でさまざまなレジスタを保存する必要があるためです。

2.4.3 引数ポインタの変更

ルーチンが AP を変更すると，コンパイラはそのルーチン内での AP のすべての使用を R12 に変更し，このような変更をすべて報告します。このようにして引数リスト全体を参照する手法はサポートされていませんが，アドレス 0(AP) を R12 に明示的に移動し，エントリ・ポイントでhome_args=TRUEを指定することで，同様の結果を得ることができます。 R12 は VAX 形式の引数リストを指します。

2.4.4 呼び出されたルーチン内での動的な条件ハンドラの設定

呼び出されたルーチン内で動的な条件ハンドラを設定するには，条件ハンドラのアドレスをフレームに格納する必要があります。コンパイラは， 0(FP) を変更する.CALL_ENTRY ルーチンに対して，静的な条件ハンドラを生成します。静的なハンドラは動的なハンドラを呼び出すか，条件ハンドラのアドレスがフレームに格納されていない場合には戻ります。

OpenVMS Alpha システムでは，性能上の理由から，コンパイラは，条件ハンドラを設定するすべてのルーチンの最後に自動的に TRAPB 命令を挿入するようなことはしません。 Alpha システムでのトラップは不確定であるため，これにより，ルーチンの最後の付近にある命令でのトラップが，ルーチンのエピローグ・コードでフレーム・ポインタが変更された後で処理されたために，宣言されたハンドラでトラップが処理されないことがあります。ルーチン内で生成されるすべてのトラップを，宣言されたハンドラで処理する必要がある場合は，ルーチンの実行を終了させる RET 命令の直前に EVAX_TRAPB ビルトインを挿入します。

2.5 JSB ルーチンのエントリ・ポイントの宣言

アセンブルされたVAX MACROコードとコンパイルされた OpenVMS Alpha および OpenVMS I64 のオブジェクト・コードでは，JSB ルーチンのパラメータは通常，レジスタで渡されます。

レジスタに書き込みを行う JSB ルーチンでは，レジスタの新しい内容を出力として呼び出し元に返すのでない場合は，これらのレジスタの内容の保存および復元を行うことがあります。しかし，VAX MACROモジュールでは，PUSHL や POPL などの命令を実行してレジスタ内容の保存および復元を行い，レジスタ内容の下位 32 ビットのみを保存することが非常に一般的であるため，コンパイラは 64 ビット・レジスタの保存と復元をルーチンの入口と出口に追加する必要があります。

コンパイラは，使用されているレジスタ・セットを算定できますが，ルーチンの呼び出し元(他のモジュールにある可能性もある)に関する詳しい知識がないと，どのレジスタに出力値を格納するつもりなのかを判断できません。そのため，JSB ルーチンの各エントリ・ポイントに，.JSB_ENTRY 指示文または.JSB32_ENTRY指示文を追加して，ルーチンでのレジスタの使用方法を宣言する必要があります。

2.5.1 .JSB_ENTRY と.JSB32_ENTRY の違い

JSB のエントリ・ポイントを宣言する方法は 2 つあります。.JSB_ENTRY 指示文は標準的な宣言であり，.JSB32_ENTRY 指示文は，ルーチンの呼び出し元が，64 ビット・レジスタの上位 32 ビットの保護を必要としないことが分かっている場合に使用します。

.JSB_ENTRY 指示文で宣言されたルーチンでは，コンパイラはデフォルトでルーチンの入口で R0 と R1 を除く，ルーチン内で変更されているすべてのレジスタの 64 ビット全体の内容を保存し，ルーチンの出口で内容を復元します。レジスタがルーチン内で明示的に変更されていない場合は，コンパイラはルーチンの呼び出しの前後でその内容を保護しません。第 2.6 節に示すように，.JSB_ENTRY 指示文のレジスタ引数を使用することで，コンパイラのデフォルトの動作を変更することができます。

.JSB32_ENTRY を使用すると，コンパイラはレジスタの保存および復元を自動的に行わないため，現在の 32 ビット動作はそのままになります。コンパイラは，preserve引数で明示的に指定されたレジスタの 64 ビット値だけを保存および復元します。

レジスタの上位 32 ビットを保護する必要がない環境でルーチンを移植する場合は，.JSB32_ENTRY 指示文を使用することをお勧めします。これにより，それぞれのルーチンでレジスタを使用しているかどうかを調べる必要がないため，コードをより迅速に移植することができます。.JSB32_ENTRY 指示文は引数なしで指定でき，必要なレジスタの値を保護するには，32 ビット・レジスタをプッシュ/ポップする既存のコードで十分です。

注意

OpenVMS Alpha や OpenVMS I64 の他の言語のコンパイラでは，64 ビットの値を使用することがあります。そのため，ルーチンが別の言語から呼び出される場合や，別の言語から呼び出される別の MACRO ルーチンから呼び出される場合は，.JSB32_ENTRY は使用できません。

2.5.2 .JSB32_ENTRY を使用する一般的な 2 つのケース

.JSB32_ENTRY を使用できる一般的な 2 つのケースがあります。

- ユーザ・モード・アプリケーションまたは自己完結型のサブシステム全体が Macro-32 で記述されている場合は，アプリケーション全体で.JSB32_ENTRY を使用できます。
- 別の言語，または 64 ビットのレジスタ保護が必要なソースから呼び出されるメジャー Macro-32 ルーチンが 1 つある場合で，そのルーチンがいくつかの他の Macro-32 ルーチンを呼び出している場合は， 64 ビット保護のバリアを設定できます。それには，メジャー・ルーチンで.JSB_ENTRY または.CALL_ENTRY を使用し，明示的な出力でないすべてのレジスタを保護します。このようにすれば，内部的なサブルーチンで.JSB32_ENTRY 指示文を使用できます。

  コンパイラは，デフォルトではメジャー・ルーチンで明示的に変更されているレジスタについてのみ保存および復元を行うため，すべてのレジスタを明示的に保存せずに.JSB_ENTRY または.CALL_ENTRY を使用するだけでは，バリアとして十分ではありません。.JSB32_ENTRY を使用する内部のサブルーチンで，保護されていないレジスタが変更されることがあります。バリアを迂回するような方法で，内部のサブルーチンを呼び出すことができないことも確認してください。

注意

.JSB32_ENTRY 指示文が使用できることが分かっていると，大幅に時間が節約できる場合があります。レジスタの上位 32 ビットが使用されている状況で.JSB32_ENTRY を使用すると，非常に分かりにくく追跡が難しいバグの原因となります。問題が発生しているルーチンの数段階上の呼び出しで 64 ビットの値が破壊されるためです。

.JSB32_ENTRY は，AST ルーチン，条件ハンドラなど，非同期に実行されるコードでは使用しないでください。

2.5.3 JSB ルーチン内の PUSHR 命令と POPR 命令

PUSHR/POPR を使用して一部のレジスタの保存と復元を行っているルーチンに.JSB_ENTRY を追加する場合があります。ルーチンの使い方によっては，いくつかのレジスタが，コンパイラにより 1 回， PUSHR/POPR によってもう 1 回，合計 2 回保存および復元されます。コードの性能がきわめて重要な場合以外はこれを最適化

しないでください。コンパイラはこの状況検出し，冗長な保存と復元をなくそうとします。

2.5.4 JSB ルーチン内での動的な条件ハンドラの設定

.JSB_ENTRY ルーチン内に 0(FP) を変更するコードがあると，コンパイラによってエラーが出力されます。

2.6 ルーチンのレジスタ使用の宣言

エントリ・ポイント指示文.CALL_ENTRY，.JSB_ENTRY，.JSB32_ENTRY，.CALL_LINKAGE (OpenVMS I64 のみ)，.USE_LINKAGE (OpenVMS I64 のみ)，.DEFINE_LINKAGE (OpenVMS I64 のみ) では，以下の 4 つのレジスタ宣言引数を指定することができます。

- input
- output
- scratch
- preserve

これらのレジスタ引数は，ルーチン内でのレジスタの使われ方を記述し，ルーチンをコンパイルする方法をコンパイラに指示するために使用します。以下の目的でレジスタ引数を使用できます。

- コンパイラのデフォルトのレジスタ保護動作を変更するため (第 2.4.2 項および第 2.5.1 項を参照)
- コンパイラが使用できる一時レジスタがないことを示すため
- ルーチンでのレジスタの使われ方を文書化するため

注意

OpenVMS Alpha **システムのみ**: /OPTIMIZE=VAXREGS **を指定して** VAX **レジスタを一時レジスタとして使用する場合，暗黙的に使用しているレジスタが一時レジスタとして使用されないように，**input**句および**output**句を使用して，そのようなレジスタの使用をすべて宣言する必要があります。この最適化が有効になっていると，コンパイラは，明示的に宣言されていないすべてのレジスタを一時レジスタとして使用します。**

2.6.1 エントリ・ポイントのレジスタ宣言の input 引数

input引数は，ルーチンが入力値を受け取るレジスタを示します。ルーチン自身はレジスタの内容を入力値として使用せず，それを使用するルーチンを呼び出す場合もあります。これは，**パススルー入力テクニック**と呼ばれ，呼び出される側のルーチンもルーチン・エントリ・マスクでそのレジスタをinputとして宣言する必要があります。

input引数を指定しても，コンパイラのデフォルトのレジスタ保護動作に影響はありません。input引数のみに指定されたレジスタでも，コンパイラによって第 2.4.2 項および第 2.5.1 項で説明したように扱われます。レジスタが入力として使用されているときに，デフォルトの保護動作を変更したい場合は，input引数に加えて，output引数，preserve引数，scratch引数のいずれかでもこのレジスタを指定する必要があります。

input引数は，ルーチンの入口で，指定されたレジスタに意味がある値が格納されており，コンパイラがそのレジスタの最初の使用を検出する前であっても，そのレジスタを一時レジスタとして使用できないことをコンパイラに対して通知します。通常，コンパイラは VAX レジスタ (R2 ～ R12) を一時レジスタとして使用しないため，input引数にレジスタを指定すると，以下の 2 つの場合にコンパイラの一時レジスタの使用が影響を受けます。

- OpenVMS Alpha システムのみ: 最適化オプション VAXREGS を使用している場合。この最適化を有効にすると，VAX MACROコードで明示的に使用されていないすべての VAX レジスタをコンパイラが一時レジスタとして使用できるようになります。なお，.JSB32_ENTRY 指示文では， VAXREGS 最適化を使用した場合，コンパイラは常にすべての VAX レジスタが入力として使用されていると仮定します。そのため， VAX レジスタをinput引数に指定し，コンパイラの一時レジスタとして使用されないようにする必要はありません。
- Alpha または Itanium のレジスタのいずれか (R13 以降) を明示的に使用している場合。

上記のいずれかのケースで，入力として使用されているレジスタをinput引数に指定しないと，そのレジスタはコンパイラによって一時レジスタとして使用され，入力値が壊れてしまいます。

最適化オプション VAXREGS を使用しておらず， Alpha や Itanium のレジスタも使用していない場合は，入力マスクはルーチンの文書化のためにのみ使用されます。

2.6.2 エントリ・ポイントのレジスタ宣言の output 引数

output引数は，ルーチンがその呼び出し元に返す値を格納するレジスタを示します。多くの場合はそのルーチン自体がレジスタを変更しますが，出力値を代入する別のルーチンをそのルーチンが呼び出す場合もあります。これは**パススルー出力テクニック**と呼ばれ，呼び出される側のルーチンもそのルーチン・エントリ・レジスタ・セットでそのレジスタをoutputとして宣言する必要があります。

この引数を使用することで，ルーチンの中で変更されるレジスタの自動的な保護が禁止されます。この引数に指定されているレジスタは，ルーチンによって変更されている場合でも，.CALL_ENTRY ルーチンや.JSB_ENTRY ルーチンで保護されません。

output引数は，コンパイラに対し，ルーチンの出口で，指定されたレジスタに意味のある値が格納されており，コンパイラが最後にそのレジスタの使用を検出した後でも，そのレジスタを一時レジスタとして使用できないことを知らせます。通常，コンパイラは VAX レジスタ (R2 ～ R12) を一時レジスタとして使用しないため，output引数にレジスタを指定すると，以下の 2 つの場合にコンパイラの一時レジスタの使用が影響を受けます。

- OpenVMS Alpha システムのみ: 最適化オプション VAXREGS を使用している場合。この最適化を有効にすると，VAX MACROコードで明示的に使用されていないすべての VAX レジスタをコンパイラが一時レジスタとして使用できるようになります。なお，.JSB32_ENTRY指示文では， VAXREGS 最適化を使用した場合，コンパイラは常にすべての VAX レジスタが出力として使用されていると仮定します。そのため， VAX レジスタをoutput引数に指定し，コンパイラの一時レジスタとして使用されないようにする必要はありません。
- Alpha または Itanium のレジスタのいずれか (R13 以降) を明示的に使用している場合。

上記のいずれかのケースで，出力として使用されているレジスタをoutput引数に指定しないと，そのレジスタはコンパイラによって一時レジスタとして使用され，出力値が壊れてしまいます。

.JSB32_ENTRY ルーチンでは，デフォルトではどのレジスタも保存されないため，output引数はコードの文書化のためだけに使用されます。

OpenVMS Alpha システムのみ: VAXREGS の最適化では，すべてのレジスタは出力であるものと見なされるため，output引数はコードの文書化のためにのみ使用されます。

2.6.3 エントリ・ポイントのレジスタ宣言の scratch 引数

scratch引数は，ルーチンの中で使用しているものの，ルーチンの入口と出口で保存および復元をする必要がないレジスタを示します。ルーチンの呼び出し元は，これらのレジスタに出力値が格納されることを期待せず，レジスタが保護されることも期待しません。

この引数を使用すると，ルーチン内で変更されるレジスタが保護されなくなります。この引数に指定されているレジスタは，ルーチン内で変更される場合でも保護されません。

scratch引数は，コンパイラの一時レジスタの使用にも関係します。レジスタ R13 以降がルーチンのソース・コードで使用されていなければ，コンパイラはこれらのレジスタを一時レジスタとして使用します。 OpenVMS Alpha システムと OpenVMS I64 システムでは， R13 ～ R15 が変更される場合は保護する必要があるため，コンパイラはこれらのレジスタを使用する場合には保護します。

しかし，これらのレジスタがscratchレジスタ・セット宣言に指定されている場合は，コンパイラは，一時レジスタとしてそれを使用する場合に保護しません。その結果，これらのレジスタは，ルーチンのソースで使用されていなくても，scratchセットに指定されていれば，ルーチンの出口では壊れている可能性があります。 VAXREGS による最適化を使用した場合は (Alpha システムのみ)，これはレジスタ R2 ～ R12 にも適用されます。

.JSB32_ENTRY ルーチンでは， R2 ～ R12 はデフォルトでは保存されないため，scratch宣言に指定しても，文書化の意味しかありません。

2.6.4 エントリ・ポイントのレジスタ宣言の preserve 引数

preserve引数は，ルーチン呼び出しの前後で保護する必要があるレジスタを指示します。これには，変更され， 64 ビットの内容全体を保存および復元する必要があるレジスタだけを含める必要があります。

preserve引数を指定すると，コンパイラの.CALL_ENTRY 指示文または.JSB_ENTRY 指示文の処理によってレジスタが自動的に保護されるかどうかにかかわらずレジスタは保護されます。.JSB32_ENTRY ルーチンでレジスタの 64 ビット全体を保存および復元する唯一の方法でもあります。 R0 と R1 はスクラッチ・レジスタであるため，ルーチンのエントリ・ポイントでpreserve引数に指定しないかぎり，コンパイラはこれらのレジスタの保存と復元を行いません。

この引数は，output引数とscratch引数より優先されます。preserve引数とoutput引数またはscratch引数の両方にレジスタを指定すると，コンパイラはレジスタを保護しますが，次の警告を出力します。

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

preserve引数は，コンパイラの一時レジスタの使用には影響を与えません。

2.6.5 レジスタ・セットを指定するためのヘルプ

コンパイラを起動する際，コマンド行で/FLAG=HINTS を指定すると，ルーチンのエントリ・ポイントに対してレジスタ・セットを構成するのに役立つメッセージが表示されます。コンパイラが出力するヒントの中には，以下の内容が出力されます。

- ルーチンの入力として使用されている可能性があるレジスタ。ただし，意図していないレジスタが表示されることがあります。レジスタに書き込む前に読み込むと，ここで出力されます。
- 出力値として使用されている可能性があるレジスタ。レジスタに書き込んだ後に読み込みを行っていないと，そのレジスタは出力値として使用されている可能性があるレジスタの一覧に追加されます。これも，意図していないレジスタが表示されることがあります。
- エントリ・ポイントのpreserve引数に指定されておらず，コンパイラが保存と復元を行うレジスタ。

.CALL_ENTRY，.JSB_ENTRY，および.JSB32_ENTRY のレジスタ引数に，ルーチンのテキスト・ヘッダに記述されているルーチンのインタフェースを反映させることをお勧めします。レジスタ引数input，output，scratch，およびpreserveは，できるだけすべてのルーチンで宣言してください。引数を指定するのは，宣言するレジスタがある場合だけでかまいません (たとえば，input=<>は必要ありません)。

2.7 ローカル・ルーチン間での分岐

あるルーチンの本体から，同じモジュールおよび psect にある別のルーチンの本体に分岐することができます。しかし，両方のルーチンでオーバヘッドが増えることになるため，コンパイラによって情報レベルのメッセージが出力されます。

注意

コンパイラは$EXIT の呼び出しをルーチンの終了として認識しません。ルーチンを終了させるには，$EXIT の後に RET または RSB のうち適切なものを追加してください。

CALL ルーチンが， RSB を実行するコード・パスに分岐する場合，エラー・メッセージが出力されます。このような CALL ルーチンは，修正しないと，実行時にエラーになります。

JSB ルーチンが， RET 命令を実行するコード・パスに分岐し， JSB ルーチンがレジスタを保護する場合，情報メッセージが出力されます。この構造は実行時に動作しますが， JSB ルーチンによって保存されたレジスタは復元されません。

コード・パスを共用する複数のルーチンのレジスタ宣言が異なっていると，レジスタの復元は条件付きで行われます。すなわち，ルーチンの入口で保存されるレジスタはどちらのルーチンでも同じですが，レジスタが復元されるかどうかは，どちらのエントリ・ポイントが実行されたかによって変わります。

以下に例を示します。

```
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
```

どちらのルーチンでも，ルーチンの入口で R3 は保存レジスタに含まれます。しかしルーチンの出口では， R3 を復元する前にマスク (同様に入口で保存されます) がテストされます。 R4 はどちらのエントリ・ポイントでも復元される必要があるため， R4 の復元の前にはマスクはテストされません。

コードを共用する 2 つのルーチンで書き込みが行われるレジスタを，一方ではscratchとして宣言し他方では宣言しないと，保存と復元を行うよりもコストが高くなります。この場合，両方でscratchとして宣言するか，一方のルーチンで保護が必要な場合は，両方でpreserveとして宣言します。

2.8 例外エントリ・ポイントの宣言 (OpenVMS Alpha のみ)

付録 B で説明するとおり，.EXCEPTION_ENTRY 指示文は，例外サービス・ルーチンのエントリ・ポイントを示します。.EXCEPTION_ENTRY 指示文は，以下のような割り込みを扱うルーチンのエントリ・ポイントを宣言するために使用します。

- インターバル・クロック
- プロセッサ間割り込み
- システムとプロセッサの訂正可能なエラー
- 電源障害
- システムとプロセッサのマシン・アボート
- ソフトウェア割り込み

ルーチンの入口で， R3 にはプロシージャ記述子のアドレスが格納されている必要があります。ルーチンは REI 命令で終了する必要があります。

例外エントリ・ポイントで，割り込みディスパッチャは，レジスタ R2 ～ R7, PC， PSL をスタックにプッシュします。これらのレジスタの内容にアクセスするには，.EXCEPTION_ENTRY 指示文でstack_base引数を指定します。コンパイラは，stack_baseに指定したレジスタに SP の値を設定するコードをルーチンの入口に

生成し，例外サービス・ルーチンがこのレジスタを使用して，スタック上のレジスタの値を使用できるようにします。

コンパイラは，ルーチンで使用されているその他すべてのレジスタを自動的に保存して復元します。さらに，サービス・ルーチンが CALL 命令または JSB 命令を実行する場合， R0 と R1 を含むすべてのスクラッチ・レジスタが自動的に保存および復元されます。

注意

フレームの 0(FP) にアドレスを格納することで設定されるエラー処理ルーチンは，.EXCEPTION_ENTRY ルーチンではありません。このようなエラー・ハンドラは，.CALL_ENTRY ルーチンとして宣言し， RET 命令で終了する必要があります。

2.9 パック 10 進数命令の使用

パック 10 進数指示文.PACKED と， EDITPC 以外のすべてのパック 10 進数命令は，コンパイルされたモジュールの外部にあるエミュレーション・ルーチンによって，MACRO コンパイラでサポートされます。

2.9.1 OpenVMS VAX と OpenVMS Alpha/I64 の実装の違い

OpenVMS VAX システムと OpenVMS Alpha/I64 システムの実装の違いを以下のリストに示します。

- パック 10 進数命令と不可分性

  パック 10 進数命令はすべてサブルーチン呼び出しによってエミュレートされるため，不可分でも再実行可能でもありません。また，.PRESERVE ATOMICITY 指示文や/PRESERVE=ATOMICITY オプションを使用して不可分にすることもできません。さらに OpenVMS Alpha システムや OpenVMS I64 システムでは，レジスタ R16 ～ R28 が命令の前後で保護されるという保証もありません。

- 引数レジスタの使用

  引数は，引数レジスタ (OpenVMS Alpha システムでは R16 ～ R21) によってエミュレーション・ルーチンに渡されます。パック 10 進数命令でこれらのレジスタを引数として使用すると，正常に動作しません。

  コンパイラは， VAX の引数ポインタ (AP) への参照を， OpenVMS Alpha の引数レジスタへの参照に変換します (2.3 項を参照)。たとえば，コンパイラは 4(AP) などの VAX の AP 相対のパラメータ参照を， Alpha の R16 に変換します。

OpenVMS I64 システムでは，最初のパラメータはレジスタ R16 ～ R21 ではなく，レジスタ R32 ～ R39 で渡されます。そのため，引数レジスタは 6 個ではなく 8 個あります。また， OpenVMS I64 では， MACRO コンパイラはこれらのレジスタを名前で参照する手段を提供していません。そのため，これらのレジスタの使用が競合するようなコードを記述する可能性はありません。

OpenVMS Alpha システムでは，パック 10 進数命令を使用する場合の AP ベースの参照の問題を防ぐには，これらの参照を含むルーチンのエントリ・ポイントで/HOME_ARGS=TRUE を指定するのが最も簡単です。これにより，すべての AP ベースの参照が，引数レジスタではなくプロシージャ・フレームから読み込まれるため，命令を変更する必要がありません。

/HOME_ARGS=TRUE を使用しないと， AP レジスタ・ベースのパラメータ参照をパック 10 進数命令のオペランドとして含むソース・コードは，変更する必要があります。まず AP ベースのオペランドを一時レジスタにコピーし，パック 10 進数命令ではその一時レジスタを使用します。たとえば，次のコードがあるとします。

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

これを次のように変更します。

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

OpenVMS I64 システムでは，入力引数レジスタは出力引数レジスタとは分離されていることに注意してください。

- オーバフロー・トラップ

  VAX のプログラム状態ワード (PSW) のビット 7 とビット 5 は，それぞれ DV (10 進オーバフロー・トラップ有効) ビットと IV (整数オーバフロー・トラップ有効) ビットです。これらのビットは， VAX PSW のエミュレーションの一環としてはエミュレートされず，パック 10 進数のサポートにより整数と 10 進数のオーバフローを有効または無効にできます。ただし，コンパイル時に静的に行う必要があります。

  10 進オーバフロー・フォルトを有効にするには，シンボル PD_DEC_OVF をゼロ以外として定義します。 PD_DEC_OVF が定義されていないかゼロが設定されている場合は，パック 10 進数エミュレーション・ルーチンは 10 進数オーバフロー・フォルトを生成しません。整数オーバフロー・フォルトを有効にするには，シンボル PD_INT_OVF をゼロ以外として定義します。 PD_INT_OVF が定義されていないかゼロが設定されている場合は，パック 10 進数エミュレーション・ルーチンは整数オーバフロー・フォルトを生成しません。

  MACRO の修飾子/ENABLE=OVERFLOW と指示文.ENABLE OVERFLOW は，パック 10 進数エミュレーション・ルーチンのオーバフロー・トラップの有効化には関係しません。 PD_DEC_OVF と PD_INT_OVF を使用する必要があります。

- 予約オペランド，10 進数のゼロ除算，整数オーバフロー，10 進数オーバフローのトラップ・ルーチン

  エミュレーション・ルーチンには，予約オペランド，10 進数のゼロ除算，整数オーバフロー，10 進数オーバフローに対して固有のトラップ・ルーチンがあります。予約オペランドと 10 進数のゼロ除算のトラップは，事象が発生すると必ず取得されます。オーバフロー・トラップは，明示的に有効にされた場合にだけ取得されます。トラップ・ルーチンは，重大度を回復不可能として LIB$SIGNAL を呼び出します。

- パック 10 進数エミュレーション・ルーチンからのメッセージ

  コンパイラのパック 10 進数エミュレーション・ルーチンからのすべてのメッセージでは，標準の OpenVMS シグナル値 SS$_ROPRAND，SS$_DECOVF，SS$_INTOVF，および SS$_FLTDIV が使用されます。

- 引数の形式についての制限事項

  これらの命令はマクロで実装されているため，引数の形式に 1 つの制限があります。マクロの呼び出しでは，先頭にサーカンフレックス(^)があると，パラメータが文字列であることを意味するものと解釈され，サーカンフレックスの直後の文字は文字列の区切り文字と解釈されます。そのため，^x20(SP) のようにオペランド型の指定で始まる引数は使用できません。#^XFF などのイミディエイト・モード引数は，先頭の文字がサーカンフレックスでないため，オペランド型指定を使用できます。

2.10 浮動小数点命令の使用

POLYx，EMODx，およびすべての H_floating 命令を除くすべての浮動小数点命令と指示文がサポートされています。

これらの命令は，サブルーチン呼び出しによってエミュレートされます。このサポートは，既存のほとんどのVAX MACROモジュールを自動的に移植できるようにするために提供されており，高速な浮動小数点性能を目的としたものではありません。

エミュレーション・ルーチン呼び出しのオーバヘッドに加えて，OpenVMS Alpha システムでは，すべての浮動小数点オペランドはメモリで渡す必要があります。これは，Alpha アーキテクチャには，整数レジスタから浮動小数点レジスタに直接値を移動する命令がないためです。また，実行される最初の浮動小数点命令で，プロセスの FEN (浮動小数点有効) ビットがオンになり，イメージが実行されているかぎり，浮動小数点レジスタ・セット全体がコンテキスト・スイッチのたびに保存および復元されます。

2.10.1 OpenVMS VAX と OpenVMS Alpha/I64 の実装の違い

OpenVMS VAX システムと OpenVMS Alpha/I64 システムの実装の違いを以下のリストに示します。

- 浮動小数点命令と不可分性

  浮動小数点命令はすべてサブルーチン呼び出しによってエミュレートされるため，不可分でも再実行可能でもありません。また，.PRESERVE ATOMICITY 指示文や/PRESERVE=ATOMICITY オプションを使用して不可分にすることもできません。さらにレジスタ R16 ～ R28 が命令の前後で保護されるという保証もありません

- 引数レジスタの使用

  引数は，引数レジスタ (OpenVMS Alpha システムでは R16 ～ R21) によってエミュレーション・ルーチンに渡されます。浮動小数点命令でこれらのレジスタを引数として使用すると，正常に動作しません。

  OpenVMS I64 システムでは，最初のパラメータはレジスタ R16 ～ R21 ではなく，レジスタ R32 ～ R39 で渡されます。そのため，引数レジスタは 6 個ではなく 8 個あります。また， OpenVMS I64 では， MACRO-32 コンパイラはこれらのレジスタを名前で参照する手段を提供していません。そのため，これらのレジスタの使用が競合するようなコードを記述する可能性はありません。

  OpenVMS Alpha システムでは，浮動小数点命令を使用する場合の AP ベースの参照の問題を防ぐには，これらの参照を含むルーチンのエントリ・ポイントで/HOME_ARGS=TRUE を指定するのが最も簡単です。これにより，すべての AP ベースの参照が，引数レジスタではなくプロシージャ・フレームから読み込まれるため，命令を変更する必要がありません。

  /HOME_ARGS=TRUE を使用しないと， AP レジスタ・ベースのパラメータ参照を浮動小数点命令のオペランドとして含むソース・コードは，変更する必要があります。まず AP ベースのオペランドを一時レジスタにコピーし，浮動小数点命令ではその一時レジスタを使用します。たとえば，次のコードがあるとします。

```
MOVF     @8(AP), @4(AP)
```

  これを次のように変更します。

```
MOVL    8(AP), R1
MOVL    4(AP), R2
MOVF    (R1),(R2)
```

  OpenVMS I64 システムでは，入力引数レジスタは出力引数レジスタとは分離されていることに注意してください。

- OpenVMS Alpha システムでの D_floating 形式

Alpha アーキテクチャでは， D_floating 形式は完全にはサポートされていません。すべての算術演算または変換は， D_floating 形式を G_floating 形式に変換し， G_floating 形式で演算を行った後に，元の D_floating 形式に変換する必要があります。その結果， D_floating 形式の仮数部が 3 ビット失われる上，変換に時間がかかります。そのため， D_floating 形式を使用しても得るものはありません。互換性のためだけに使用可能になっていますが，精度が少し低下するため注意が必要です。

- OpenVMS I64 システムでの VAX 浮動小数点形式

  Itanium アーキテクチャでは， IEEE の S 形式と T 形式だけがサポートされているため，すべての算術演算または変換は， D_floating 形式と G_floating 形式を T_floating 形式に変換し， T_floating 形式で演算を行ってから，元の D_floating 形式または G_floating 形式に戻すことで行われます。その結果， D_floating 形式の仮数部が 3 ビット失われる上，変換に時間がかかります。そのため， D_floating 形式を使用しても得るものはありません。互換性のためだけに使用可能になっていますが，精度が少し低下するため注意が必要です。同様に，すべての F_floating 演算は， S_floating に変換し，演算を行ってから F_floating に戻すことで実行されます。

- オーバフロー・トラップ

  整数オーバフローや浮動小数点アンダフローに対してトラップを有効にすることはできません。 Alpha システムは浮動小数点オーバフローに対して常にトラップを生成します。/ENABLE 修飾子と.ENABLE 指示文を使用しても，オーバフロー・トラップには影響を与えません。 OpenVMS VAX システム上で予測可能な結果になるオーバフロー・トラップは， OpenVMS Alpha システムや OpenVMS I64 システムでも同じ結果になります。

- ダーティ・ゼロ

  ダーティ・ゼロをなくすようにコードを変更する必要があります。 VAX のすべての浮動小数点形式における真のゼロは，すべてのビットにゼロが設定されています。指数ビットがすべてゼロでも，残りのいずれかのビットが 1 の場合，ダーティ・ゼロと呼ばれ， OpenVMS VAX システムではゼロとして扱われます。 OpenVMS Alpha システムでは，予約オペランド・トラップが発生します。

  OpenVMS I64 は， VAX 形式の浮動小数点をサポートしていません。しかし， VAX 浮動小数点エミュレーション・ルーチンにより VAX と同じ動作が保たれます。

- 引数の形式に関する制限

  これらの命令はマクロで実現されているため，引数の形式に関する制限が 1 つあります。マクロの呼び出しでは，先頭にサーカンフレックス(^)があると，パラメータが文字列であることを意味するものと解釈され，サーカンフレックスの直後の文字は文字列の区切り文字と解釈されます。そのため，^x20(SP) のようにオペランド型の指定で始まる引数は使用できません。#^XFF などのイミディエイト・モード引数は，先頭の文字がサーカンフレックスでないため，オペランド型指定を使用できます。

- 浮動小数点数の戻り値

  ルーチンを呼び出し，浮動小数点の戻り値が R0 に格納されることを期待している MACRO プログラムでは，呼び出し元のルーチンと呼び出されるルーチンの間に「ジャケット」が必要となります。ジャケットの目的は，浮動小数点レジスタ 0 から R0 に戻り値を移動させることです。

2.10.2 他の言語のルーチンに対する影響

このサポートでは，浮動小数点レジスタ・セットがコンパイラから見えるようになるわけではありません。整数レジスタで浮動小数点演算ができるようになるだけです。つまり，他の言語で記述されたルーチンがVAX MACROルーチンを呼び出したり，VAX MACROルーチンから呼び出される場合は，浮動小数点数値を入力または出力として使用できません。他の言語のコンパイラは，これらの値を浮動小数点レジスタで渡します。浮動小数点の引数をVAX MACROルーチンに渡したり，VAX MACROルーチンから受け取るには，必ずポインタを使用します。

他の言語の実行時ライブラリ (RTL) ルーチンを呼び出す場合にもこれが当てはまります。たとえば， MTH$RANDOM を呼び出すと，浮動小数点数値が浮動小数点レジスタ F0 に格納されて返されます。コンパイラは直接 F0 を読み取ることはできません。MTH$RANDOM を呼び出して結果を R0 に移動するジャケット・ルーチンを別の言語で作成するか，移動だけを行うルーチンを別に記述する必要があります。

2.11 VAX の不可分性と細分性の維持

VAX アーキテクチャには，ユニプロセシング・システムで，読み取り - 変更 - 書き込みメモリ操作を単一の割り込み不可能な操作として実行する命令が含まれています。**不可分性**は，メモリを一度の操作で変更できる能力のことを指します。このような命令は複雑で，性能が著しく落ちるため， Alpha システムまたは I64 システムの読み取り - 変更 - 書き込み操作は，不可分でなく，割り込み可能な命令シーケンスとしてのみ実行できます。

さらに， VAX の命令では，周囲のメモリに影響を与えることなく，メモリ中の単独のアラインされた，またはアラインされていないバイト，ワード，ロングワードをアドレス指定できます。(データ項目は，項目のアドレスがバイト単位のサイズの偶数倍になっている場合に**アラインされている**と見なされます。) **細分性**は，アラインされているロングワードの一部に対して独立に書き込みが行えることを示します。

バイト，ワード，アラインされていないロングワードのアクセスも性能を著しく低下させるため， OpenVMS Alpha システムでは，アラインされているロングワードとクォドワードにしかアクセスできません。そのため，単一のバイト，ワード，アラインされていないロングワードを書き込むための命令シーケンスを実行すると，周囲のバイトが読み込まれて書き戻されます。

Itanium にはバイトとワードにアクセスするための命令がありますが，アラインされていない場合，性能が低下します。

これらのアーキテクチャ上の違いにより，特定の条件ではデータが破壊される可能性があります。

OpenVMS Alpha システムでは，不可分性と細分性の維持は，他のスレッドがメモリを変更できないようにロックすることで実現されるのではなく，読み取り - 変更 - 書き込み操作の間にメモリが変更されたかどうかを判断する手段を提供することで実現されます。変更された場合は，読み取り - 変更 - 書き込み操作が再度実行されます。

OpenVMS I64 システムでは，不可分性は OpenVMS Alpha と同様に操作をリトライすることで実現されます。

データの一貫性を保証するため，コンパイラには，以降の項で説明する条件で使用する修飾子と指示文があります。

2.11.1 不可分性の維持

OpenVMS VAX， OpenVMS Alpha，および OpenVMS I64 のマルチプロセシング・システムでは，並列に動作する複数のスレッドが書き込み可能なグローバル・セクションにある共用データを変更するアプリケーションは，データに対するアクセスを同期させる何らかの手段を必要とします。 OpenVMS VAX のシングル・プロセッサ・システムでは，メモリの変更命令だけで共用データへのアクセスを同期させるのに十分です。しかし， OpenVMS Alpha システムや OpenVMS I64 システムでは，これだけでは不十分です。

メモリ変更オペランドを使用した VAX 命令に対して，読み取り - 変更 - 書き込み操作の一貫性を保証するために，コンパイラでは/PRESERVE=ATOMICITY オプションが使用できます。また，必要に応じてVAX MACROソース・コードのセクションに.PRESERVE ATOMICITY 指示文と.NOPRESERVE ATOMICITY 指示文を挿入し，不可分性の有効と無効を切り替えることもできます。

たとえば，以下の命令があるとします。この命令は， R1 が指すデータに対する読み込み，変更，書き込みシーケンスを要求します。

```
INCL (R1)
```

OpenVMS VAX システムでは，マイクロコードがこれら 3 つの操作を実行します。そのため，シーケンスが完全に完了するまで割り込みは発生しません。

OpenVMS Alpha システムでは，この 1 つの VAX 命令を実行するために，以下の 3 つの命令を実行する必要があります。

```
LDL     R27, (R1)
ADDL    R27, 1, R27
STL     R27, (R1)
```

同様に OpenVMS I64 システムでは，以下の 4 つの命令を実行する必要があります。

```
ld4     r22 = [r9]
sxt4    r22 = r22
adds    r22 = 1, r22
st4     [r9] = r22
```

この Alpha/Itanium コード・シーケンスの問題は，それぞれの命令の間で割り込みが発生する可能性があるという点です。割り込みにより AST ルーチンが実行されるか， LDL と STL の間に別のプロセスがスケジュールされ， AST または他のプロセスが R1 が指すデータを更新すると， STL は古いデータに基づく結果 (R1) を格納します。

不可分な操作が必要な場合で，/PRESERVE=ATOMICITY (または.PRESERVE ATOMICITY) を指定した場合は，コンパイラは INCL (R1) に対して次の Alpha 命令シーケンスを生成します。

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

Itanium での命令シーケンスは次のとおりです。

```
$L3:    ld4             r23 = [r9]
        mov.m           apccv = r23
        mov             r22 = r23
        sxt4            r23 = r23
        adds            r23 = 1, r23
        cmpxchg4.acq    r23, [r9] = r23
        cmp.eq          pr0, pr6 = r22, r23
  (pr6) br.cond.dpnt.few $L3
```

OpenVMS Alpha システムでは，このシーケンス中に現在のプロセッサまたは他のプロセッサ上の他のコード・スレッドによって (R1) が変更されると， STL_C (Store Longword Conditional) 命令が (R1) を更新せず， R28 に 0 を書き込むことでエラーになったことを通知します。この場合，コードは元に分岐し，干渉なく操作が完了するまで操作がリトライされます。

Alpha アーキテクチャの分岐予測ロジックでは，後方条件分岐が成立すると想定されるため， BEQ Retry の代わりに， BEQ Fail と BR Retry が実行されます。この操作はめったにリトライする必要がないため，分岐成立が想定されない前方条件分岐を行う方が効率が良くなります。

OpenVMS Alpha システムでは，不可分性を保持するための仕組みにより，ユニプロセッサシステムとマルチプロセッサ・システムの両方にこの不可分性の保証が適用されます。この保証は実際の変更命令だけに適用され，以降のメモリ・アクセスや以前のメモリ・アクセスにはインターロックが拡張されません (第 2.11.6 項を参照)。

OpenVMS I64 版のコードでは， compare-exchange 命令 (cmpxchg) を使用してロックされたアクセスが実現されますが，効果は同じです。変更しようとしているメモリが他のコードによって変更された場合，コードがループ・バックし，操作をリトライします。

OpenVMS Alpha システムまたは OpenVMS I64 システムにアプリケーションを移植する際に，書き込み可能なグローバル・セクションにある共用データを複数のプロセスが変更する場合は，アプリケーションがシングル・プロセッサ上でのみ動作する場合であっても，特に注意してください。また，メインライン・プロセス・ルーチンが変更するプロセス空間のデータが，非同期システム・トラップ (AST) ルーチンや条件ハンドラでも変更される可能性がある場合は，アプリケーションを調べる必要があります。 Alpha システムにおける読み取り - 変更 - 書き込み操作に関するプログラミング上の問題についての詳細は， Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications[1]を参照してください。

警告

不可分性を維持するときに，コンパイラは，Alpha PALcode の非アライン障害ハンドラで扱うことのできない，アラインされたメモリ命令を生成します。この命令では，アラインされていないアドレスに対して，回復不能な予約オペランド・フォルトが発生します。そのため，.PRESERVE ATOMICITY が指定されたすべてのメモリ参照は，アドレスがアラインされている必要があります (第 2.11.5 項を参照)。

2.11.2 細分性の維持

Alpha 上でVAX MACROのバイト，ワード，またはアラインされていないロングワードに対するメモリ書き込み命令の細分性を維持するということは，指定されたデータに対して命令が正常に実行され，周囲のデータの一貫性が維持されることを保証するということを意味します。

[1] このマニュアルはアーカイブ扱いになっています。このマニュアルの保守は行われておらず， OpenVMS ドキュメント・セットにも含まれていません。ただし，http://www.hp.com/go/openvms/docからオンラインで参照することができます。

VAX アーキテクチャには，メモリ内のバイト，ワード，アラインされていないロングワードに対して独立にアクセスする命令があるため，アラインされた同じロングワードの異なるバイトに 2 つのプロセスが同時に書き込んでも，互いに干渉が起こりません。

Alpha アーキテクチャのオリジナルの実装では，アラインされたロングワードおよびクォドワード・オペランドのみをアドレス指定できる命令が定義されていました。ただし，バイトとワードのオペランドのロードと格納が後で追加されました。

Alpha では，長さがロングワード未満のメモリや，アラインされていないメモリにデータ・フィールドを書き込むコードは，クォドワードのロード，変更したデータのクォドワードへの挿入，クォドワードの格納を行う割り込み可能な命令シーケンスを使用する必要があります。この場合，同じクォドワード内の異なるバイトに書き込もうとしている 2 つのプロセスは，実際にはクォドワード全体をロードし，演算を行い，保存します。ロード操作と格納操作のタイミングによっては，どちらかのバイト書き込みが失われる可能性があります。

Itanium アーキテクチャでは，バイト，ワード，ロングワード，クォドワードのアドレス指定が可能なため，アクセスの細分性が簡単に確保でき，オプションや宣言も必要ありません。

コンパイラには，バイト，ワード，アラインされていないロングワードの書き込みの一貫性を保証する/PRESERVE=GRANULARITY オプションがあります。/PRESERVE=GRANULARITY オプションを指定すると，バイト，ワード，アラインされていないロングワードへの書き込みを行うすべての VAX 命令に対して，細分性が保たれる Alpha 命令が生成されます。また，必要に応じてVAX MACROソース・コードのセクションに.PRESERVE GRANULARITY 指示文と.NOPRESERVE GRANULARITY 指示文を挿入し，細分性の維持を有効または無効にすることもできます。

たとえば，命令 MOVB R1, (R2) は，次の Alpha コード・シーケンスを生成します。

```
LDQ_U     R23, (R2)
INSBL     R1, R2, R22
MSKBL     R23, R2, R23
BIS       R23, R22, R23
STQ_U     R23, (R2)
```

LDQ_U 命令と STQ_U 命令の間で，他のコード・スレッドが (R2) が指すデータの一部を変更した場合，そのデータは上書きされて失われます。

次の Itanium コード・シーケンスが生成されます。

```
st1     [r28] = r9
```

コマンド修飾子または指示文で，同じ命令に対して細分性を維持するように指定した場合， Alpha コード・シーケンスは次のようになります。

```
          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
```

この場合， (R2) が指すデータが別のコード・スレッドによって変更されると，操作はリトライされます。

Itanium では，コードはすでに影響のあるメモリ位置にだけ書き込むようになっているため，コード・シーケンスは変わりません。

MOVW R1,(R2) 命令で，細分性を維持するために生成される Alpha コードは，コンパイラのレジスタ・アラインメント・トラッキング機能によって，レジスタ R2 が現在アラインされていると見なされているかどうかに依存します。 R2 がアラインされていると見なされている場合は，コンパイラは基本的に上の MOVB の例と同じコードを生成します。ただし， INSBL 命令と MSKBL 命令の代わりに INSWL 命令と MSKWL 命令が使用され， R2 のアドレスに対する BIC で#^B0110 を使用します。 R2 がアラインされていないと見なされている場合は，ワードが仮にクォドワード境界にまたがっている場合でも正しく書き込まれるように，コンパイラは 2 つの個別の LDQ_L/STQ_C ペアを生成します。

同様に Itanium でも，アドレスがワードにアラインされている場合は，コンパイラは単に`st2 [r28] = r9`を生成します。

注意

細分性の維持が有効になっている場合，アラインされたワードを書き込むために生成されるコードでは，アドレスがアラインされていないと，実行時に回復不可能な予約オペランド・フォルトが発生します。書き込み先のアドレスがアラインされていない可能性がある場合は，アラインされていないワードに書き込むことができるコードを生成するようにコンパイラに指示します。それには，書き込み命令の直前で，コンパイラ指示文.SET_REGISTERS UNALIGNED=Rn**を使用します。**

MOVL R1,(R2) 命令の細分性を維持するために，書き込み先のアドレスがアラインされていないと想定される場合でも，コンパイラは常に STL 命令を使用してロングワード全体を書き込みます。アドレスがアラインされていないと， STL 命令は非アライン・メモリ参照のフォルトになります。 PALcode の非アライン・フォルトのハン

ドラは，アラインされていないロングワードに書き込むために必要な，ロード，マスク，格納を行います。ただし， PALcode は割り込み可能でないため，これによって周囲のメモリが破壊されないことが保証されます。

アプリケーションを OpenVMS Alpha システムに移植する場合，ローカル・プロセッサで実行されているプロセスや，システム内の別のプロセッサで実行されているプロセス， AST ルーチンや条件ハンドラと共用しているメモリに対して，アプリケーションがバイト，ワード，またはアラインされていないロングワードの書き込みを行うかどうかを確認する必要があります。 OpenVMS Alpha システムでの細分性操作に関係する，プログラミング上の問題点についての詳細は， Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications を参照してください。

注意

INSV 命令は，細分性の維持がオンになっている場合に，細分性が正しく維持されるコードを生成しません。

2.11.3 不可分性と細分性の優先順位

細分性と不可分性の維持をどちらも有効にし，両方の維持が必要な VAX コードをコンパイラが発見すると，細分性よりも不可分性が優先されます。

たとえば，.PRESERVE=GRANULARITY を指定してコンパイルすると，命令 INCW 1(R0) は，割り込まれたら新しいワード値の書き込みをリトライします。しかし，.PRESERVE=ATOMICITY を指定してコンパイルすると，割り込まれたら初期値をフェッチしてインクリメントします。両方のオプションを指定した場合は，後者の動作になります。

また，コンパイラは，細分性を維持するアラインされていないワードおよびロングワードに対するコードを正常に生成できますが，不可分性を維持するアラインされていないワードおよびロングワードに対するコードは生成できません。両方のオプションを指定した場合には，すべてのメモリ参照はアラインされているアドレスに対するものである必要があります。

2.11.4 不可分性が保証できない場合

コンパイラの不可分性の保証は， VAX 命令のメモリ変更操作に対してだけ影響を与えるため， OpenVMS Alpha システムまたは OpenVMS I64 システムにおいて，/PRESERVE=ATOMICITY で解決できないVAX MACROソースのコーディング上の問題を調べる際には，特に注意する必要があります。

たとえば，次の VAX 命令があるとします。

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

この命令に対して，/PRESERVE=ATOMICITY (または.PRESERVE ATOMICITY) が指定されていると，コンパイラは次のような Alpha コード・シーケンスを生成します。

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

このコード・シーケンスで， STL_C が失敗すると，加算の前に変更オペランドだけが再読み込みされます。データ (R1) は再読み込みされません。この動作は VAX の動作とは少し異なります。 OpenVMS VAX システムでは，命令全体が割り込みなしで実行されます。 OpenVMS Alpha システムや OpenVMS I64 システムでは，変更オペランドだけが不可分に更新されます。

その結果，データ (R1) の読み込みが不可分であることが必要なコードでは，ロックなどの別の方法を使用して，同じレベルの同期を実現する必要があります。

この命令では，コンパイラは次の Itanium コード・シーケンスを生成します。

```
        ld4     r19 = [r9]
        sxt4    r19 = r19
        adds    r16 = 4, r9
$L4:    ld4     r17 = [r16]
        mov.m   apccv = r17
        mov     r15 = r17
        sxt4    r17 = r17
        add     r17 = r19, r17
        cmpxchg4.acq r17, [r16] = r17
        cmp.eq  pr0, pr7 = r15, r17
(pr7)   br.cond.dpnt.few $L4
```

別の VAX 命令について考えます。

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

この命令に対して，不可分性の維持がオンの場合もオフの場合も，コンパイラは次の Alpha コード・シーケンスを生成します。

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

この例の VAX 命令は単一のVAX CPU 上で不可分ですが， Alpha 命令シーケンスは単一の Alpha CPU 上で不可分ではありません。 4(R1) オペランドが書き込みオペランドであり，変更オペランドでないため，操作は LDL_L と STL_C を使用した不可分な操作にされません。

OpenVMS I64 システムでは，コード・シーケンスは次のようになります。

```
ld4     r14 = [r9]
sxt4    r14 = r14
adds    r24 = 4, r9
st4     [r24] = r14
```

最後に，より複雑な VAX INCL 命令について考えます。

```
INCL    @(R1)
```

この命令に対して，/PRESERVE=ATOMICITY (または.PRESERVE ATOMICITY) が指定されていると，コンパイラは次のような Alpha コード・シーケンスを生成します。

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

ここで，変更データの更新だけが不可分です。変更データのアドレスを取得するために必要なフェッチは，不可分シーケンスの一部になっていません。

OpenVMS I64 システムでは，コード・シーケンスは次のようになります。

```
        ld4     r16 = [r9]
        sxt4    r16 = r16
$L5:    ld4     r14 = [r16]
        mov.m   apccv = r14
        mov     r24 = r14
        sxt4    r14 = r14
        adds    r14 = 1, r14
        cmpxchg4.acq r14, [r16] = r14
        cmp.eq  pr0, pr8 = r24, r14
(pr8)   br.cond.dpnt.few $L5
```

2.11.5 不可分性のためのアラインメントの留意事項

不可分性を維持する場合，コンパイラは変更データがアラインされているものと想定する必要があります。クォドワード境界にまたがったフィールドは，2 つの読み取り - 変更 - 書き込みシーケンスが必要となるため，不可分に更新できません。

OpenVMS Alpha システムでは，通常のロード命令や格納命令と異なり，アラインされていない LDx_L 命令や STx_C 命令はソフトウェアで処理できないため，アライン

されていないアドレスに対する LDx_L 命令や STx_C 命令は，回復不可能な予約オペランド・フォルトを生成します。

OpenVMS I64 システムでは， compare-exchange (cmpxchg) 命令でのアラインされていないアドレスをソフトウェアで処理できないため，実行時に例外が発生します。

OpenVMS Alpha システムでは，/PRESERVE=ATOMICITY (または.PRESERVE ATOMICITY) が指定されていると， INCL (R1) 命令は LDL_L 命令および STL_C 命令を生成するため， R1 はアラインされたロングワードである必要があります。

次の命令があるとします。

```
INCW (R1)
```

OpenVMS Alpha システムでは，この命令に対し，コンパイラは次のようなコード・シーケンスを生成します。

```
        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
```

最初の BIC 命令で，#^B0111 ではなく#^B0110 が使用されている点に注意してください。これは，ワードがクォドワード境界にまたがらないようにするためです。そうしないと，メモリの更新が不完全になります。 R1 に格納されているアドレスがアラインされているワードを指していない場合は，ビット 0 がオンになり， BIC 命令ではそのビットがクリアされません。この場合， LDQ_L (Load Quadword Locked) 命令は，回復不可能な予約オペランド・フォルトを生成します。

すべてのバイトはアラインされているため， INCB 命令では#^B0111 を使用し，アラインされたアドレスを生成します。

OpenVMS I64 システムでは，INCW (R1) 命令に対して，コンパイラは次のようなコード・シーケンスを生成します。

```
$L5:    ld2           r19 = [r9]
        mov.m         apccv = r19
        mov           r18 = r19
        sxt2          r19 = r19
        adds          r19 = 1, r19
        cmpxchg2.acq  r19, [r9] = r19
        cmp.eq        pr0, pr8 = r18, r19
(pr8)   br.cond.dpnt.few  $L5
```

2.11.6 インターロックされる命令と不可分性

コンパイラが不可分性を維持する方法には，コンパイルされたVAX MACROコードにおいて興味深い副作用があります。

OpenVMS VAX システムでは，マルチプロセッサ・システムにおける共用データへのアクセスを同期させる方法としては，インターロックされる命令だけが正常に機能します。 OpenVMS Alpha のマルチプロセシング・システムでは，不可分性が維持される変更命令とインターロックされる命令をコンパイルした結果得られるコードは，どちらも正常に機能します。これは，コンパイラが両方の命令セットに対して生成する LDx_L と STx_C が，複数のプロセッサにわたって正しく動作するためです。同様に， OpenVMS I64 システムでは， compare-exchange (cmpxchg) 命令が，プロセッサにまたがったインターロック機能を提供します。

このコンパイラの副作用は OpenVMS Alpha システムと OpenVMS I64 システムに固有であり， OpenVMS VAX システムには該当しないため，VAX MACROコードを OpenVMS Alpha または OpenVMS I64 に移植する際に，コードを両方のシステムで動作させる予定の場合は，この副作用に頼らないようにしてください。

しかし，不可分性が維持されない他の命令に対するインターロックとしてメモリ変更が使用されている場合は，インターロックされる命令を使用する必要があります。これは， Alpha アーキテクチャと Itanium アーキテクチャでは，書き込み順序が厳密には保証されないためです。

たとえば，次のVAX MACROコード・シーケンスがあるとします。

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

このコード・シーケンスから，次の Alpha コード・シーケンスが生成されます。

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

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

Alpha アーキテクチャと Itanium アーキテクチャのデータ・フェッチ動作に起因して，(R1) への格納が処理されるよりも前に (R2) からデータが読み込まれる可能性があります。INCL (R1) 命令をロックとして使用し，ロックが設定される前に (R2) のデータがアクセスされないようにすると，(R1) のインクリメントよりも前に (R2) の読み込みが行われる可能性があり，保護されません。

VAX のインターロックされる命令では，インターロックされる命令の前後に Alpha の MB (メモリ・バリア) 命令または Itanium の mf (メモリ・フェンス) 命令が生成されます。これにより，インターロックされる命令をまたいでメモリのロードが移動されなくなります。

OpenVMS I64 では，コード・シーケンスは次のようになります。

```
$L7:    ld4     r16 = [r9]
        mov.m   apccv = r16
        mov     r15 = r16
        sxt4    r16 = r16
        adds    r16 = 1, r16
        cmpxchg4.acq r16, [r9] = r16
        cmp.eq  pr0, pr10 = r15, r16
(pr10)  br.cond.dpnt.few $L7
        ld4     r3 = [r28]
        sxt4    r3 = r3
```

次のコード・シーケンスがあるとします。

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

このコード・シーケンスは，次の Alpha コード・シーケンスを生成します。

```
        MB
Retry:  LDL_L   R28,(R1)
        ADDL    R28,#1,R28
        STL_C   R28,(R1)
        BEQ     R28, Fail
        MB
        LDL     R3, (R2)
         .
         .
         .
Fail:   BR      Retry
```

OpenVMS I64 では，次のようなコード・シーケンスが生成されます。

```
        mf
$L8:    ld2     r23 = [r9]
        mov.m   apccv = r23
        adds    r24 = 1, r23
        cmpxchg2.acq r14, [r9] = r24
        cmp.eq  pr0, pr11 = r23, r14
(pr11)  br.cond.dpnt.few $L8
        mf
        ld4     r3 = [r28]
        sxt4    r3 = r3
```

MB 命令または mf 命令があると，その前にあるすべてのメモリ操作が完了してからその後のメモリ操作が開始されるようになります。

2.12 コンパイルとリンク

コンパイラは以下のファイルを必要とします。

- SYS$LIBRARY:STARLET.MLB

  これは，コンパイラ指示文が定義されたマクロ・ライブラリです。コードをコンパイルする際，コンパイラは自動的に STARLET.MLB のコンパイラ指示文の定義を確認します。

- SYS$LIBRARY:STARLET.OLB

  これは，コンパイラが使用するエミュレーション・ルーチンとその他のルーチンが格納されているオブジェクト・ライブラリです。コードをリンクすると，リンカは STARLET.OLB とリンクして未定義のシンボルを解決します。

コンパイラ修飾子については，付録 A を参照してください。

2.12.1 リスト・ファイルの行番号

リスト・ファイル内でのマクロ展開の行番号は， Xnn/mmmとなります。ここで，Xnnはネストの深さを表し，mmmは最も外側のマクロからの相対行番号です。

例 2–1 に OpenVMS I64 のリスト・ファイルを示します。 OpenVMS Alpha のリスト・ファイルのソース部分は基本的に同じです。

例 2–1 OpenVMS I64 のリスト・ファイルでの行番号の例

(次ページに続く)

例 2–1 (続き) OpenVMS I64 のリスト・ファイルでの行番号の例

```
                    00000000       1 ;
                    00000000       2 ; This is the Itanium (previously called "IA-64") version of
                    00000000       3 ; ARCH_DEFS.MAR, which contains architectural definitions for
                    00000000       4 ; compiling VMS sources for VAX, Alpha, and I64 systems.
                    00000000       5 ;
                    00000000       6 ; Note: VAX, VAXPAGE, and IA64 should be left undefined,
                    00000000       7 ;       a lot of code checks for whether a symbol is
                    00000000       8 ;       defined (e.g. .IF DF VAX) vs. whether the value
                    00000000       9 ;       is of a expected value (e.g. .IF NE VAX).
                    00000000      10 ;
                    00000000      11 ;VAX   = 0
                    00000000      12 ;EVAX  = 0
                    00000000      13 ;ALPHA = 0
00000001            00000000      14 IA64   = 1
                    00000000      15 ;
                    00000000      16 ;VAXPAGE = 0
00000001            00000000      17 BIGPAGE  = 1
                    00000000      18 ;
00000020            00000000      19 ADDRESSBITS = 32
                    00000000      20  .TITLE ug_ex_listing /line numbering in the listing file/
                    00000000      21 ;
                    00000000      22  .MACRO test1
                    00000000      23  clrl r1
                    00000000      24  clrl r2
                    00000000      25  tstl 48(sp)  ; generate uplevel stack error
                    00000000      26  clrl r3
                    00000000      27  .ENDM test1
                    00000000      28  .MACRO test2
                    00000000      29  clrl r4
                    00000000      30  clrl r5
                    00000000      31  test1
                    00000000      32  clrl r6
                    00000000      33  .ENDM test2
                    00000000      34
                    00000000      35 foo: .jsb_entry
                    00000000      56  .show expansions
                    00000000      57  clrl r0
                    00000011      58  test2
    1.......
%IMAC-E-UPLEVSTK, (1) up-level stack reference in routine FOO
```

(次ページに続く)

例 2–1 (続き) OpenVMS I64 のリスト・ファイルでの行番号の例

```
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      59  rsb
             00000012      60  .noshow expansions
             00000012      61
             00000012      62  .END
```

2.13 デバッグ

コンパイラはデバッガを完全にサポートしています。コンパイルされたVAX MACROコードのデバッグ・セッションは，アセンブルされたVAX MACROコードのデバッグと同様です。ただし，ここで説明する重要な違いがいくつかあります。デバッグの詳しい説明については， OpenVMS Debugger Manual を参照してください。

2.13.1 コードの再配置

大きな違いの 1 つは，コードがアセンブルされるのではなくコンパイルされるという点です。 OpenVMS VAX システムでは，VAX MACROのそれぞれの命令は単一の機械語命令でした。 OpenVMS Alpha システムや OpenVMS I64 システムでは，VAX MACROのそれぞれの命令は多数の Alpha または Itanium の機械語命令にコンパイルされます。この違いの大きな副作用は，コンパイル・コマンドで/NOOPTIMIZE を指定しなかった場合のコードの再配置と再スケジューリングです。

デフォルトでは，いくつかの最適化が行われ，生成されたコードがソースの境界を超えて移動されます (第 1.2 節，第 4.3 節，および付録 A を参照)。ほとんどのコード・モジュールでは，/NOOPTIMIZE を指定してコンパイルすることでデバッグ作業が単純化されます。これを指定すると，再配置が行われなくなります。コードのデバッグを終えたら，/NOOPTIMIZE を指定せずに再コンパイルして性能を向上させることができます。

2.13.2 ルーチンの引数のシンボリック変数

コンパイルされたコードのデバッグとアセンブルされたコードのデバッグのもう 1 つの大きな違いは，VAX MACROに対する新しい概念である，ルーチンの引数を調べるためのシンボリック変数の定義です。 OpenVMS VAX システムでは，ルーチンをデバッグしていて引数を調べたくなったら，通常，次のような操作をします。

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

または

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

OpenVMS Alpha システムと OpenVMS I64 システムでは，引数は OpenVMS VAX システムと異なりメモリ内のベクタにありません。さらに， OpenVMS Alpha システムと OpenVMS I64 システムには AP レジスタがありません。VAX MACROのコンパイルされたコードをデバッグする際に EXAMINE @AP と入力すると，デバッガから AP が未定義のシンボルであると報告されます。

コンパイルされたコードでは，引数は以下の場所の組み合わせに置かれています。

- レジスタ
- ルーチンのスタック・フレームより上のスタック中
- 引数リストがホーミングされている場合 (第 2.4 節を参照) や，レジスタ引数を保存する必要があるルーチンを呼び出している場合は，スタック・フレーム内。

生成されたコードを参照してどこに引数があるかを探す必要はありません。代わりに，引数の正しい位置を指している$ARGnシンボルが用意されています。$ARG0 シンボルは， VAX システムでの@AP+0 と同じで引数の個数です。$ARG1 シンボルは最初の引数で，$ARG2 が 2 番目の引数となり，以下同様です。これらのシンボルは CALL_ENTRY 指示文と JSB_ENTRY 指示文では定義されていますが，EXCEPTION_ENTRY 指示文では定義されていません。

2.13.3 $ARGnシンボルを使用せずに引数を見つける

コンパイラが$ARGnシンボルを生成しない追加の引数がコードの中にある場合があります。.CALL_ENTRY ルーチンで定義される$ARGnシンボルの数は，コンパイラによって検出された最大数です (自動的に検出されるか， MAX_ARGS によって指定します)。.JSB_ENTRY ルーチンでは，引数は呼び出し元のスタック・フレームにホーミングされ，コンパイラは実際の数を検出できないため，常に 8 個の$ARGnシンボルが作成されます。

ほとんどの場合，追加の引数を容易に見つけることができますが，見つけられない場合もあります。

2.13.3.1 簡単に見つけることができる追加の引数

以下の場合は追加の引数を容易に見つけることができます。

- 引数リストがホーミングされておらず，$ARGnシンボルが OpenVMS Alpha では$ARG7 以降まで， OpenVMS I64 では$ARG9 以降まで定義されている場合。引数リストがホーミングされていない場合， OpenVMS Alpha では$ARG7 以降， OpenVMS I64 では$ARG9 以降の$ARGnシンボルは，スタック上のクォドワードとして渡されたパラメータのリストを常に指します。以降の引数は，最後に定義された$ARGnシンボルに続くクォドワードの中にあります。
- 引数リストがホーミングされている場合で，コンパイラによって検出された最大数 (自動的に検出されるか， MAX_ARGS によって指定します) に等しいかそれよりも前の引数を調べたい場合。引数リストがホーミングされている場合，$ARGnシンボルは常にホーミングされた引数リストを指します。以降の引数は，最後に定義された$ARGnシンボルに続くロングワードの中にあります。

たとえば，次のようにして， JSB ルーチン (引数リストは呼び出し元でホーミングされている必要があります) の 8 個の引数を超えて引数を調べることができます。

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

この例では，引数をホーミングするときに，呼び出し元が少なくとも 10 個の引数を検出したことが仮定されています。

引数をホーミングしていないルーチンの中で最後の$ARGnシンボルを超えて引数を見つけるには，上の例の EX .+4 を EX .+8 に置き換えます。

2.13.3.2 簡単に見つけることができない追加の引数

以下の場合は追加の引数を容易に見つけることができません。

- 引数リストがホーミングされておらず，$ARGnシンボルが OpenVMS Alpha では$ARG6， OpenVMS I64 では$ARG8 までしか定義されていない場合。この場合，存在している$ARGnシンボルは，レジスタまたはスタック・フレーム中のクォドワードのどちらかを指します。どちらの場合でも，定義されている$ARGnシンボルの後ろのクォドワードを参照しても，以降の引数を調べることはできません。
- 引数リストがホーミングされており，コンパイラが検出した数を超える引数を調べたい場合。$ARGnシンボルは，ホーミングされた引数リストに格納されているロングワードを指します。コンパイラは検出できた数の引数だけをこのリストに

移動します。ホーミングされた最後の引数の後ろのロングワードを調べると，さまざまな他のスタック・コンテキストを調べることになります。

これらの場合に追加の引数を探す唯一の方法は，コンパイルされた機械語コードを調べ，引数がある場所を確認することです。調べたい引数の最大数を MAX_ARGS に正しく指定すれば，これらの問題はどちらも解決できます。

2.13.4 OpenVMS I64 での VAX と Alpha のレジスタ名の使用

便宜上， OpenVMS I64 上の MACRO コンパイラではシンボル名 R0， R1，... R31 が定義されており， Alpha のこれらのレジスタの値が格納されている Itanium のレジスタを参照するようになっています。それでも，デバッガの名前%R0，%R1，... %R31 を使用してマシンのネイティブな番号付けでレジスタを参照することができます。

2.13.5 パック 10 進数データを使用したコードのデバッグ

パック 10 進数データを使用した，コンパイルされたVAX MACROコードを，OpenVMS Alpha システムまたは OpenVMS I64 システムでデバッグする際には，以下の情報を念頭に置いてください。

- EXAMINE コマンドを使用して，.PACKED 指示文で宣言された場所を調べるときには，デバッガは自動的に値をパック 10 進数データ型として表示します。
- パック 10 進数データを格納することができます。構文は VAX での構文と同じです。

2.13.6 浮動小数点データを使用したコードのデバッグ

浮動小数点データを使用した，コンパイルされたVAX MACROコードを， OpenVMS Alpha システムまたは OpenVMS I64 システムでデバッグする際には，以下の情報を念頭に置いてください。

- EXAMINE/FLOAT コマンドを使用して， Alpha または Itanium の整数レジスタの浮動小数点数値を調べることができます。

  OpenVMS Alpha システムと OpenVMS I64 システムでは浮動小数点演算用のレジスタ・セットがありますが，浮動小数点演算が含まれる，コンパイルされたVAX MACROコードでは，これらのレジスタは使用されません。整数レジスタだけが使用されます。

  コンパイルされたVAX MACROコードにおける浮動小数点演算は，コンパイラの外で動作するエミュレーション・ルーチンによって実行されます。そのため，VAX MACRO浮動小数点演算をたとえば R7 に対して実行しても，浮動小数点レジスタ 7 には影響がありません。

- EXAMINE コマンドを使用して.FLOAT 指示文またはその他の浮動小数点ストレージ指示文で宣言された場所を調べる場合は，デバッガは値を自動的に浮動小数点データとして表示します。
- EXAMINE コマンドを使用して G_FLOAT データ型を調べる場合は，デバッガは 2 つのレジスタの内容を使用して VAX データの値を構築しません。

  次の例を見てください。

```
EXAMINE/G_FLOAT   R4
```

  この例で， R4 と R5 の下位のロングワードは， VAX の場合と異なり値を構築するために使用されません。代わりに， R4 のクォドワードの内容が使用されます。

  D_FLOAT の演算と G_FLOAT の演算に対してコンパイラが生成するコードは，連続する 2 つのレジスタの下位のロングワードに VAX 形式のデータを保持します。そのため，これら 2 つのレジスタのどちらかに対してEXAMINE/G_FLOATを使用しても，正しい浮動小数点数値が得られません。また，これらどちらかのレジスタに対してDEPOSIT/G_FLOATを実行しても，目的の結果は得られません。ただし， 2 つの値の半分ずつを手動で組み合わせることはできます。たとえば，次の命令を実行したとします。

```
MOVG    DATA, R6
```

  次のようなシーケンスで， R6 と R7 にある G_FLOAT 値を読み取ることができます。

```
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
```

- DEPOSIT コマンドを使用して， Alpha または Itanium の整数レジスタに浮動小数点データを格納することができます。構文は VAX システムでの構文と同じです。
- H_FLOAT はサポートされていません。
- OpenVMS I64 システムでは，入力パラメータは R16 ～ R21 ではなく R32 ～ R39 です。出力パラメータは，コンパイラによって選択された，より大きな番号のレジスタに格納されます。

3
ソースに対する推奨される変更と必要な変更

この章では，VAX MACROコードを OpenVMS Alpha または OpenVMS I64 に移植する際に調べる必要があるコーディング構造について説明します。ここで説明するコーディング構造がモジュール中にあると，移植に時間がかかります。

コンパイラはすべてのVAX MACROコードを透過的に変換できるわけではありません。 OpenVMS Alpha または OpenVMS I64 のコードに直接コンパイルできないコードも多数あります。コンパイラはこのようなコードの多くを検出し，診断メッセージを出力します。正常にコンパイルするためには，これらのコードを削除するか変更する必要があります。

ほとんどの場合，ソース・コードの変更が必要です。例外がある場合はその旨明記してあります。

この章のトピックは以下のとおりです。

- 第 3.1 節, スタックの使用
- 第 3.2 節, 命令ストリーム
- 第 3.3 節, フロー制御メカニズム
- 第 3.4 節, イメージの動的な再配置
- 第 3.5 節, 静的なデータの上書き
- 第 3.6 節, 外部シンボルを使用した静的な初期化
- 第 3.7 節, 転送ベクタ
- 第 3.8 節, 算術例外
- 第 3.9 節, ページ・サイズ
- 第 3.10 節, ワーキング・セットへのページのロック
- 第 3.11 節, 同期

3.1 スタックの使用

OpenVMS の呼び出し規則では， OpenVMS Alpha システムおよび OpenVMS I64 システムのスタック・フレームの形式が定義されています。これは， OpenVMS VAX システムで定義されている形式とは大きく異なります。コードが OpenVMS VAX のスタック・フレームの形式に依存している場合は， OpenVMS Alpha システムまたは OpenVMS I64 システムに移植する際に変更する必要があります。

3.1.1 プロシージャ・スタック・フレームの参照

FP からの正のスタック・オフセットに対する参照はできず，参照するとコンパイラでエラーとなります。この規則に対する 1 つの例外は，VAX MACROコードでルーチン・アドレスを FP が指すスタック位置に移動することにより動的な条件ハンドラを設定する場合です。コンパイラはこれを検出し，このようなハンドラを設定するための適切な Alpha または Itanium のコードを生成します。しかし， 0(FP) への書き込みが JSB ルーチンの内部にある場合は，コンパイラはエラーとして出力します。コンパイラは負の FP オフセットを許し，これは，プロシージャの先頭で割り当てたスタック・ストレージを参照するために使用されます。

推奨される変更
スタック・フレームの参照は， OpenVMS Alpha または OpenVMS I64 の形式に変換するのではなく，可能であればすべて削除してください。たとえば，問題のコードが，保存されたレジスタの値を変更しようとしていた場合は，新しい値を一時的にスタックに格納し，ルーチンの出口でレジスタ値を設定します (エントリ・レジスタ・マスクからそのレジスタを削除します)。

3.1.2 現在のスタック・フレームの外側の参照

コンパイラは，VAX MACROモジュール全体でスタックの深さを監視することで，呼び出し元がプッシュしたデータをルーチン内で参照している箇所を検出し，エラーを出力します。

推奨される変更
呼び出し元がスタックにプッシュしたデータをルーチン内で参照している部分を削除する必要があります。代わりに，必要なデータをパラメータとして渡すか，データの読み込み先となるスタック・ベースへのポインタを渡します。

3.1.3 アラインされていないスタックの参照

ルーチンの呼び出し時に，スタックがオクタワード境界にアラインされていない場合は，コンパイラはスタックをオクタワード境界にアラインします。一部のコードでは，スタック上に構造を構築する際に，アラインされていないスタック参照を行ったり，スタック・ポインタがアラインされていない状態になります。コンパイラは，これに対して情報レベルのメッセージを出力します。

推奨される変更
スタックにプッシュするデータ要素や構造体に十分なパディングを追加するか，データ構造のサイズを変更します。アラインされていないスタック参照は， VAX の性能にも影響を与えるため，これらの修正を， VAX， Alpha， Itanium アーキテクチャ向けに設計されたコードに適用することをお勧めします。

3.1.4 スタック上のデータ構造の構築

一般的なコーディング・スタイルとしては，要素をプッシュすることでスタック上に構造を作成し，自動デクリメントによってスタック・ポインタを移動させ領域を割り当てます。この方法の問題を以下に挙げます。

- 構造体にパディングを追加してフィールドをアラインするために，構築しようとしている構造体の SDL ソースを変更すると，コードが破壊されます。コードに assume 文が含まれていない場合は，この問題はコンパイル時に検出されません。
- フィールドのサイズを拡張すると，現在の命令はすべてのデータにアクセスしなくなりますが，これも検出されません。

推奨される変更

最初の問題を修正し，2 番目の問題を検出するには，この例に示すコーディング・テクニックを使用してください。次のコード例を考えます。

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

このコードは，次のコードで置き換えます。

```
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 VAX の SP および PC へのクォドワードの移動

VAX と Alpha/Itanium コンピュータのアーキテクチャ上の違いにより，プログラマの介入なしでは， VAX のスタック・ポインタ (SP) およびプログラム・カウンタ (PC) へのクォドワードの移動を完全にエミュレートすることはできません。 VAX アーキテクチャでは， R14 が SP として定義され， R15 が PC として定義されています。 SP をターゲットとした MOVQ 命令では，次の例に示すように VAX の SP と PC へのロードが同時に実行されます。

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

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

SP を移動先とした MOVQ 命令をコンパイラが見つけると，コンパイラは指定された読み込み元からスタック・ポインタへの符号拡張されたロングワードのロードを生成し，次の情報メッセージを出力します。

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

推奨される変更

MOVQ 命令を使用している目的が，VAX の動作を実現することである場合は，次に示すように，MOVL 命令を使用し，その後で目的のアドレスへの分岐を使用することをお勧めします。

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

MOVQ 命令を使用している目的が，スタック・ポインタに 8 バイトの値をロードすることである場合は，次に示すように，代わりに EVAX_LDQ ビルトインを使用してください。

```
EVAX_LDQ        SP, REGDATA
```

3.2 命令ストリーム

この項で説明するVAX MACROのコーディング・スタイルと VAX 命令は，OpenVMS Alpha や OpenVMS I64 では動作しないか，予期しない結果になります。

3.2.1 命令ストリームに埋め込まれたデータ

コンパイラは，命令ストリームに埋め込まれたデータを検出し，エラーとして報告します。

命令ストリーム中のデータは，多くの場合 JSB 命令の後に.LONG が続いた形になります。この構造により，VAX MACROコードは，スタック上の戻りアドレスを使用してデータを見つける JSB ルーチンに暗黙的にパラメータを渡すことができます。コード・ストリーム中でデータを使用する方法としてもう 1 つ頻繁に使用されるのは，コードとともにメモリ中にデータを連続させ，1 つの単位として再配置できるようにすることです。

推奨される変更

暗黙的な JSB パラメータに対しては，パラメータ値をレジスタで渡します。ロングワードよりも大きな値の場合は，データを別のプログラム・セクション (psect) に格納し，そのアドレスを明示的に渡します。

静的なデータは独立したデータ psect にある必要があるため，コードとデータを一緒に再配置するようなコードは，書き直す必要があります。

3.2.2 実行時のコード生成

コンパイラはスタック領域や静的なデータ領域への分岐を検出し，エラーとして出力します。

推奨される変更
命令を構築して後で実行するコード，スタック領域に分岐するコード，静的なデータ領域に分岐するコードは削除するか変更する必要があります。このようなコードがどうしても必要な場合は，条件付けしてそのコードを OpenVMS VAX 用とし，それに相当する適切な OpenVMS Alpha または OpenVMS I64 用のコードを作成してください。

3.2.3 命令サイズへの依存

たとえば，命令の長さに基づいて分岐オフセットを計算するコードは，変更する必要があります。

推奨される変更
ラベルと標準的な分岐を使用するか，計算型 GOTO に対しては CASE 命令を使用してください。

3.2.4 不完全な命令

OpenVMS VAX コードの一部の CASE 命令は，その後にオフセット・テーブルがなく，代わりにリンカによる psect の配置に依存して命令を完成させています。コンパイラは不完全な命令をエラーとして出力します。

推奨される変更
モジュール内の命令を完成させるか，データ psect 内にアドレスのテーブルを作成し， CASE 命令を，テーブルから分岐先アドレスを選択して分岐するようなコードで置き換えます。

3.2.5 トランスレートされない VAX 命令

コンパイラは以下の VAX 命令をトランスレートできないため，エラーとして出力します。

- LDPCTX および SVPCTX
- XFC
- ESCD， ESCE，および ESCF
- BUGx

推奨される変更
通常これらの命令は， VAX アーキテクチャに大きく依存するコードに現れます。このようなコードを OpenVMS Alpha システムまたは OpenVMS I64 システムに移植するには，書き直す必要があります。

3.2.6 内部プロセッサ・レジスタの参照

以下の命令には特に注意してください。

- MFPR
- MTPR

推奨される変更
OpenVMS Alpha システムでは，これらの命令が正しい Alpha 内部プロセッサ・レジスタ (IPR) を参照していることを確認してください。正しい内部プロセッサ・レジスタを参照していないと，エラーになります。 Alpha の内部プロセッサ・レジスタについての詳細は， Alpha Architecture Reference Manual を参照してください。

OpenVMS I64 システムでは，コンパイラは VAX 命令 MFPR と MTPR を直接サポートしていません。ただし， OpenVMS で提供されている一連のマクロがあり，このマクロが同じ機能を実行するシステム・サービスを呼び出します。

3.2.7 BICPSW 命令での条件コード Z および N の使用

BICPSW 命令はサポートされていますが，条件コード Z と N は同時には設定できません。条件コード Z を設定すると条件コード N がクリアされ，逆に条件コード N を設定すると条件コード Z がクリアされます。

推奨される変更
コードで両方の条件コードを同時に設定している場合は，コードを変更してください。

3.2.8 インターロックされるメモリ命令

Alpha Architecture Reference Manual では，インターロックされるメモリ命令の使用に対する厳密な規則が記述されています。特に， LDxL/STxC シーケンスの内部での分岐またはこのシーケンス内への分岐は禁止されています。インターロックされるシーケンスから外に向かう分岐は正しいため，変更する必要はありません。 Alpha 21264 (EV6) プロセッサとそれ以降のすべての Alpha プロセッサでは，それ以前のプロセッサと比較して，より厳密にこの規則に従うことを義務付けています。Alpha 21264 プロセッサは， OpenVMS Alpha Version 7.1-2 で初めてサポートされました。

MACRO コンパイラが Macro-32 ソース・コードから生成したコードは，これらの規則に従っています。ただし，これらの命令をソース・コード中に直接記述できるように，EVAX_LQxLビルトインとEVAX_STxCビルトインが用意されています。

これらの命令が，インターロックされるメモリ命令を使用するための規則に従って使用されていることが保証できるように， MACRO コンパイラにチェックが追加されています。この機能は， OpenVMS Alpha Version 7.1-2 ではバージョン 3.1 のコンパ

イラから， OpenVMS Alpha Version 7.2 ではバージョン 4.1 のコンパイラから追加されています。

OpenVMS I64 システムでは，コンパイラは， EVAX_LDxL ビルトインを，ロードした値をハードウェアの AR.CCV レジスタに保存すると同時に，後で使用するためにローカルなコピーを保持する命令に置き換えます。 EVAX_STxC ビルトインは，EVAX_LDxL ビルトインによって以前保存された値と比較する compare-exchange (cmpxchg) 命令に置き換えられます。コンパイラは，インターロックされる命令シーケンスへの分岐に関する Alpha のすべての規則を適用します。

ここで述べた状況下では，コンパイラによって以下の警告メッセージが出力されます。

BRNDIRLOC，branch directive ignored in locked memory sequence

説明: LDx_L/STx_C シーケンス中に.BRANCH_LIKELY 指示文が見つかりました。

対処: ありません。.BRANCH_LIKELY 指示文は無視され，他のコーディング・ガイドラインの違反がなければ，コードは書かれているとおりに動作します。

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

説明: 分岐命令の分岐先が LDx_L/STx_C シーケンスの内部になっています。

対処: この警告が出ないようにするには，ソース・コードを書き直して， LDx_L/STx_C シーケンスの内部での分岐やこのシーケンス内への分岐をなくします。インターロックされるシーケンスから外部への分岐は有効であり，エラーになりません。

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

説明: LDx_L/STx_C シーケンス内にメモリの読み書きがあります。ソース・コード中の“MOVL data, R0”などの明示的な参照の場合と，メモリへの暗黙的な参照の場合があります。たとえば，データ・ラベルのアドレスのフェッチ (“MOVAB label, R0”など) は，リンケージ・セクションからの読み込みによって実現されます。リンケージ・セクションは，外部参照を解決するために使用されるデータ領域です。

対処: この警告が出ないようにするには，すべてのメモリ・アクセスを LDx_L/STx_C シーケンスの外に移動します。

RETFOLLOC，RET/RSB follows LDx_L instruction

説明: LDx_L 命令の後， STx_C 命令が出現する前に RET 命令または RSB 命令が見つかりました。これは，不適切なロック・シーケンスです。

対処: RET 命令や RSB 命令が LDx_L 命令と STx_C 命令の間に来ないようにコードを変更します。

RTNCALLOC, routine call within locked memory sequence in routine 'routine_name'
説明: LDx_L/STx_C シーケンス内でルーチンを呼び出しています。"JSB subroutine"のようにソース・コード中に明示的な CALL/JSB がある場合と，別の命令の結果として呼び出す場合があります。たとえば， MOVC や EDIV などのいくつかの命令では，実行時ライブラリの呼び出しが生成されます。
対処: この警告が出ないようにするには，ルーチン呼び出しや，ルーチン呼び出しを生成する命令を，コンパイラの指示どおりに LDx_L/STx_C シーケンスの外へ移動します。

STCMUSFOL, STx_C instruction must follow LDx_L instruction
説明: LDx_L 命令が見つかる前に STx_C 命令が見つかりました。これは，不適切なロック・シーケンスです。
対処: LDx_L 命令の後に STx_C 命令が来るようにコードを変更します。

推奨される変更
インターロックされるメモリ命令の不適切な使用が検出された場合は，警告メッセージで説明されている推奨される対処に従ってください。

3.2.9 MOVPSL 命令の使用

MOVPSL 命令は， OpenVMS Alpha の PS をフェッチします。この内容は， VAX の PSL (プロセッサ・ステータス・ロングワード) と異なります。たとえば， OpenVMS Alpha と OpenVMS I64 の PS には条件コード・ビットが含まれていません。OpenVMS Alpha の PS についての詳細は， Alpha Architecture Reference Manual を参照してください。

推奨される変更
MOVSPL 命令はめったに使用されません。コードに MOVPSL 命令が含まれている場合， OpenVMS Alpha システムまたは OpenVMS I64 システムに移植するためには，コードを書き換える必要があります。

3.2.10 マクロによって実装される命令

VAX Macro-32 の一部のオペレーション・コードは， Alpha Macro-32 ではビルトインになり (機能を実行するために PAL またはルーチン呼び出しにコンパイルされることがあります)， Itanium ではマクロとなります (これもルーチン呼び出しに展開されることがあります)。

マクロとなったこれらのオペレーション・コードは，命令とマクロの構文上の細かい違いが原因でエラーとなることがあります。通常，オペレーション・コードのパラメータは，引数内に式がある可能性があるため，コンマで区切る必要があります。マクロの引数は単純にテキストであるため，コンマだけでなくスペースやタブも引数のセパレータと見なされます。(一部のマクロ引数は式のように見えますが，評価される

コンテキストでのマクロ展開で使用されるまでは，Macro-32 で式としては扱われません。）

不要なスペースがなくなるように再フォーマットすることで，ソース・コードを修正することができます。

次のVAX MACROファイルはこの違いを示します。

```
.macro Prober_mac a,b,c
   prober  a,b,c
.endm

   prober one, two-too(r2), three(r3)          ; This one works.

   Prober_mac one, two-too(r2), three(r3)      ; This one works.

   prober one, two - too(r2), three(r3)        ; This one works.

   Prober_mac one, two - too(r2), three(r3)    ; This one fails
                                               ; because the spaces are
                                               ; argument separators.

   prober       one, -            ; This one works.
                two - -           ; Note the second argument is
                too(r2), -        ; separated onto two lines.
                three(r3)

   Prober_mac -                   ; This one fails.
                one, -
                two - -           ; Note the second argument is
                too(r2), -        ; separated onto two lines.
                three(r3)

   Prober_mac -                   ; This one works because there
                one, -            ; are no spaces or tabs before
                two--             ; or after the continuation
too(r2), -                        ; line break.
                three(r3)

   Prober_mac -                   ; This one works, too.
                r1, -
                r2, -
                r3

.end
```

3.3 フロー制御メカニズム

VAX MACROで使用される一部のフロー制御メカニズムは，OpenVMS Alpha システムや OpenVMS I64 システムでは，望ましい結果になりません。そのため，若干のコード変更が推奨または必要となります。

制御のフローを変更するために，JSB ルーチン内でスタック上の戻りアドレスを変更するコードがこのカテゴリに含まれます。これにはいくつかのバリエーションがあり，よく使用されます。すべて再コーディングが必要です。

3.3.1 条件コードによる通信

JSB 命令の直後に条件分岐があったり，ルーチンの最初の命令が条件分岐になっていると，コンパイラはこれを検出してエラー・メッセージを出力します。

推奨される変更
条件コードによる暗黙的な通信を行う代わりに，状態値またはフラグ・パラメータを返します。

例を示します。

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

これを，次のように書き替えます。

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

すでに R0 を使用している場合は，スタックにプッシュし，エラーを処理し終えてから復元する必要があります。

3.3.2 JSB ルーチンから CALL ルーチン内への分岐

JSB ルーチンから CALL ルーチン内への分岐があり，.JSB_ENTRY ルーチンがレジスタを保存する場合，情報レベル・メッセージが出力されます。このような分岐に対してメッセージが出力される理由は，保存されているレジスタを復元するための，プロシージャのエピローグ・コードが実行されないためです。レジスタを復元する必要がない場合は，変更は不要です。

推奨される変更
.JSB_ENTRY ルーチンは，おそらく呼び出し元に代わって RET を実行しようとしていると考えられます。 JSB_ENTRY によって保存されたレジスタを， RET を実行する前に復元する必要がある場合は，.CALL_ENTRY 内の共通コードを.JSB_ENTRY に変更し，両方のルーチンから呼び出すようにします。

たとえば，次のコード例があるとします。

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

このコードを OpenVMS Alpha システムまたは OpenVMS I64 システムに移植するには，次のように.CALL_ENTRY ルーチンを 2 つのルーチンに分割します。

```
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 スタックへの戻りアドレスのプッシュ

コンパイラは，たとえば (PUSHAB label) のように，アドレスをスタックにプッシュし，以降の RSB でその場所から実行を再開しようとする操作をすべて検出し，エラーとして出力します。(OpenVMS VAX システムでは，次の RSB でルーチンの呼び出し元に戻ります。)

推奨される変更

アドレスの PUSH を削除し，現在のルーチンの RSB の直前に分岐先ラベルへの明示的な JSB を追加します。これで同じ制御フローとなります。分岐先ラベルは.JSB_ENTRY ポイントとして宣言します。

たとえば，次のコードではソース・コードの変更が必要としてメッセージが出力されます。

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

明示的なJSB命令を追加することで，次のようにコードを変更します。JSBをRSBの直前に配置している点に注意してください。この前のコードでは，PUSHABがどこにあっても，continue_labelに制御を渡すのはRSB命令でした。新しいコードではPUSHABが削除されています。

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

3.3.4 スタック上の戻りアドレスの削除

コンパイラは，スタック上の戻りアドレスの削除(たとえばTSTL (SP)+)を検出して，エラーとして出力します。VAXコードでは，戻りアドレスを削除することで，呼び出し元のさらに呼び出し元に戻ることができます。

推奨される変更

呼び出し元に対して，その呼び出し元に戻る必要があることを示す状態値を返すように，ルーチンを書き換えてください。また，最初の呼び出し元から継続ルーチンのアドレスを渡し，最下位のルーチンがそこにJSBで分岐する方法もあります。継続ルーチンがRSBで最下位のルーチンに戻ったら，最下位のルーチンがRSBで戻ります。

たとえば，次のコードに対しては，ソースの変更が必要であるというメッセージがコンパイラから出力されます。

```
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
```

次のように，状態値を返すようにコードを書き換えることができます。

```
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 戻りアドレスの変更

コンパイラは，スタック上の戻りアドレスを変更する操作をすべて検出し，エラーとして出力します。

推奨される変更

スタック上の戻りアドレスを変更するコードは，呼び出し元に状態値を返すように書き換えてください。状態値を受け取った呼び出し元が特定の場所に分岐したり，状態値として特別な.JSB_ENTRY ルーチンのアドレスを返して，呼び出し元からそのルーチンを呼び出すようにすることができます。後者の場合，呼び出し元は，特別な.JSB_ENTRY ルーチンに対して JSB を実行した直後に， RSB を実行する必要があります。

たとえば，次のコードに対して，コンパイラはソースの変更が必要である旨出力します。

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

次のように，戻り値を返すようにコードを書き換えます。

```
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 コルーチン呼び出し

2 つのルーチン間でのコルーチン呼び出しは，一般にそれぞれのルーチン内で一連の JSB 命令として実装されます。それぞれの JSB は，スタック上の戻りアドレスに制御を渡し，その過程でその戻りアドレスを削除します (たとえば，JSB @(SP)+)。コンパイラはコルーチン呼び出しを検出し，エラーとして出力します。

推奨される変更

コルーチン・リンケージを開始するルーチンを，明示的なコールバック・ルーチン・アドレスを他のルーチンに渡すように書き換える必要があります。その後，コルーチンを開始するルーチンは， JSB 命令で他のルーチンを呼び出します。

例として，次のコルーチン・リンケージを考えます。

```
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
```

このようなコルーチン・リンケージに参加しているルーチンを，次のように，明示的なコールバック・ルーチン・アドレス (ここでは R6 と R7 内) を交換するように変更します。

```
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
```

レジスタの消費を避けるには，ルーチンの入口でコールバック・ルーチンのアドレスをスタックにプッシュします。次に， JSB @(SP)+ 命令を使用して，“直接”コールバック・ルーチンに JSB で分岐します。次の例では，コールバック・ルーチンのアドレスを R0 で渡していますが，ルーチンの入口ですぐにプッシュしています。

```
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 REI を使用したモードの変更

VAX での REI 命令の一般的な用途は，明示的な移行先 PC と PSL をスタックにプッシュすることで，モードを変更することです。以下の理由から，このコードは，ソース・コードをいくらか変更しないと， OpenVMS Alpha システムや OpenVMS I64 システムではコンパイルできません。

- OpenVMS Alpha のデスティネーション・コードでは，リンケージ・セクション・ポインタが確立されている必要があります。 OpenVMS I64 のデスティネーション・コードでは， GP が確立されている必要があります。 REI では，そのための方法が提供されていません。
- Integrity サーバの REI フレームは， Alpha システムや VAX システムよりも複雑で，保存された汎用レジスタ，その他のレジスタ，レジスタ・スタックについての情報が含まれています。同様に， Alpha システムの REI フレームは VAX システムよりも複雑で，保存されたレジスタが含まれています。

  Itanium でのプロローグ・コードでは，レジスタ・フレームを確立する必要があります。

  また， Itanium および Alpha では，すべてのサブルーチンには，保存された非スクラッチ・レジスタを復元するためのエピローグ・コードがあります。レジスタの受け渡しと復元を可能にするためには，新しい構文が必要です。

- モード変更は，移行先にある別のスタック上でプロセスを実行することを意味します。そのため，以前のスタックをクリアしなくてはならにという新しい要件が発生します。

推奨される変更

OpenVMS Alpha システムや OpenVMS I64 システムで同等の機能を実行するために，システム・ルーチン EXE$REI_INIT_STACK が作成されました。このルーチンは，新しいモードとコールバック・ルーチンのアドレスをパラメータとして受け取ります。このルーチンには，高級言語からも使用できるという利点があります。

このルーチン呼び出しを使用できるように，既存のコードを再構成する必要があります。実行を継続するコード・ラベルは，システム・ルーチンから呼び出されるため，エントリ・ポイント指示文で宣言する必要があります。

次の例は，VAX MACROコードで RET 命令を使用してモードを変更する 2 つの方法と， OpenVMS Alpha または OpenVMS I64 において，同じことを EXE$REI_INIT_STACK ルーチンを使用して実現する 1 つの方法を示します。

変更前 (1)

```
PUSHL  new_PSL
PUSHL new_PC
REI
```

変更前 (2)

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

変更後

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

プログラムが継続ルーチンに到達すると，新しいモードで新しい場所から実行され，古いモードのスタックは再初期化されます。メモリ・スタックと (OpenVMS I64 の場合) レジスタ・スタックのベースは，新しいモードのものに設定されます。

外部モードのルーチンに引数を渡すという問題がある点に注意してください。OpenVMS Alpha システムでは，ほとんどのレジスタは REI にまたがって保持されます。 OpenVMS I64 システムでは， R26， R27，および R10 (Alpha レジスタ R8, R9， R10 に相当) だけが保持されます。より多くのデータを渡す必要がある場合，内部モード・スタックにデータ構造を作成し，レジスタを通じてそのアドレスを外部モード・ルーチンに渡すことはできません。内部モード・スタックは， EXE$REI_INIT_STACK によってベースにリセットされるためです。代わりに， MFPR_xSP および MTPR_xSP を使用して外部モード・スタックに領域を割り当て，そこにデータを格納する必要があります。

3.3.8 ループのネスト制限

コンパイラは，正しくコードを生成するために，プログラム・コード中のループを検出する必要があります。ループの中に別のループがあったり，部分的に別のループと重なると，“ネストした”ループとなります。ループのネストが 32 レベルを超えると，コンパイラは回復不可能なエラーが発生したことを報告し，コンパイルが停止します。VAX MACROアセンブラではループを検出する必要がないため，このような制限はありませんでした。

推奨される変更

コンパイラによってこのエラーが報告された場合は，この制限を超えないようにコードを再構成してください。

3.4 イメージの動的な再配置

OpenVMS VAX システムでは，位置独立コード (PIC) をあるアドレス範囲から別のアドレス範囲にコピーすることができ，アセンブルと実行も正常に行われます。ソース・コード・ラベルを使用してコピーされるある範囲のコードが含まれているコードを， OpenVMS Alpha または OpenVMS I64 用にコンパイルすると， 2 つの理由で動作しません。まず，コンパイルされたコードは，ソース・コードと同じ順序でない可能性があるためです。 2 番目に， OpenVMS Alpha では，外部参照を行うためにはリンケージ・セクションが必要だという点です。リンケージ・セクションは別の psect にあるだけでなく，リンカやイメージ・アクティベータによって設定された絶対アドレスも含まれています。 OpenVMS I64 では，コードは，リンカおよびイメージ・アクティベータによって構築されたグローバル・リージョンの一部を必要とします。

推奨される変更

コードを複製するか，あるアドレス範囲から別のアドレス範囲へのコードのコピーを除去します。

3.5 静的なデータの上書き

VAX MACROアセンブラでは，初期化済みの静的な記憶域を自由に上書きすることができます。それには，通常は".="構文を使用して現在のロケーション・カウンタを変更します。

これに対し， MACRO コンパイラでは，.ASCII データ以外の既存のデータ項目が部分的に上書きされないように，上書きが制限されています。以下のものは上書きできません。

- スカラ項目を同じサイズの別のスカラ項目で上書きすること。
- 内容が設定されていないすべての領域 (.BLKxまたは".=.+n"で宣言)。
- .ASCII データの複数のセクションを，他の.ASCII 指示文や.BYTE 指示文で上書きすること。

推奨される変更

可能であれば，以下のいずれかの正しい形式にコードを変更してください。

- いずれかのスカラ領域指示文 (.BYTE，.WORD，.LONG など) を使用して宣言されたデータを，同じ種類の指示文，または同じバイト数を占める指示文を使用して上書きするコード。項目は，同じ場所にあり同じサイズである必要があります。部分的に重なることはできません。次に例を示します。

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

- .ASCII データの部分的な上書き

  以前書き込んだ.ASCII データの一部を，.ASCII または.BYTE を使用して上書きすることができます。(.ASCIC および.ASCIZ は，.ASCII/.BYTE 指示文のペアとして実装されているため，.ASCIC および.ASCIZ でも.ASCII を上書きすることができます。)

  次に例を示します。

```
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"
```

  新しいデータは，以前の.ASCII 文字列の範囲内に完全に収まっている必要があります。次の例は無効です。

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

他の種類の指示文(.LONG など)を使用した部分的な上書きはできません。

3.6 外部シンボルを使用した静的な初期化

外部シンボルを使用した静的な初期化のうち，いくつかの形式は使用できません。

推奨される変更

可能であれば，使用可能な形式のいずれかにコードを変更してください。そのためには，多くの場合，$xxxDEF マクロを使用して追加のシンボルを定義します。OpenVMS Alpha システムと OpenVMS I64 システムでは，コンパイラは次の形式の式をサポートしています。

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

ここで，OPRには以下のものが入ります。

+ 加算
- 減算
*乗算
/除算
@算術シフト
&論理積
! 論理和
\ 排他的論理和

symbol1とsymbol2 (どちらもオプション) は，外部の値または別の psect 内のラベルで，offset1とoffset2には式を指定できますが，コンパイル時に定数になる必要があります。

OpenVMS I64 システムでは， ELF オブジェクト・ファイル形式の制限により，symbol1またはsymbol2がリンカでルーチンのアドレスに解決される際，減算しか行うことができません。

コンパイラは，次の形式の式をサポートしていません。

```
<symbol1> OPR <symbol2>
```

静的な初期化の式をこの形式に変形できない場合は，コンパイラは不正な静的な初期化を報告します。演算子の優先順位が正しくなるように，括弧を使用してください。

3.7 転送ベクタ

VAX MACROコード中に.TRANSFER 指示文があると，/noflag=directivesを指定していないかぎり，コンパイラはエラーとして出力します。このオプションを指定するとメッセージは出力されませんが，.TRANSFER 指示文は無視されます。

OpenVMS VAX システムでは，VAX MACROソース・コード中で転送ベクタを明示的に定義することにより，共用可能イメージ中の再配置可能なコード用にユニバーサル・エントリ・ポイントを作成できます。 OpenVMS Alpha システムと OpenVMS I64 システムでは，**リンカ・オプション・ファイル**内でユニバーサル・エントリ・ポイントのシンボル値を宣言することで作成します。リンカは，共用可能イメージ内にシンボル・ベクタ・テーブルを作成します。これにより外部イメージは，再配置可能なユニバーサル・プロシージャ・エントリ・ポイントとストレージ・アドレスを見つけることができます。

推奨される変更

VAX MACROのソースから転送ベクタを削除する必要があります。コンパイラが生成したオブジェクト・ファイルをリンクする際に， SYMBOL_VECTOR 文が記述されているリンカ・オプション・ファイルを指定する必要があります。 SYMBOL_VECTOR 文の中で，どこからでも参照可能な，再配置可能なシンボル (プロシージャ・エントリ・ポイントまたはデータアドレス) を記述し，それぞれに対して DATA と PROCEDURE のどちらかを指定します。

リンカは，リンカ・オプション・ファイルにシンボルを記述した順序でシンボル・ベクタを作成します。このシンボルの順序を，後の共用可能イメージの作成まで保持する必要があります。すなわち，シンボル・リストの最後にエントリを追加したり，エントリを削除することはできますが，現行のエントリは，リスト中で同じ順序を保つ必要があります。転送ベクタについての詳細は， OpenVMS Linker Utility Manual を参照してください。

3.8 算術例外

OpenVMS Alpha システムでは，算術例外の処理は OpenVMS VAX システムと違っており，互換性がありません。 OpenVMS VAX システム用に設計された，算術例外を処理する例外ハンドラでは，期待しているシグナル名と OpenVMS Alpha システムで実際に発生するシグナル名を対応付けることができません。

OpenVMS VAX システムでは，算術例外が同期的に報告されることがアーキテクチャによって保証されますが， OpenVMS Alpha システムでは，算術例外は非同期に報告されます。 OpenVMS VAX システムでは， VAX 算術命令で例外 (オーバフローなど) が発生するとすぐに例外ハンドラに入り，以降の命令は実行されません。例外ハンドラに対して報告されるプログラム・カウンタ (PC) は，失敗した算術命令のプログラム・カウンタとなります。これによりアプリケーション・プログラムは，メイン・シーケンスを再開して失敗した演算をエミュレートしたり，同等の演算や代わりとなる演算と置き換えるといったことが可能になります。

OpenVMS Alpha システムでは，例外の原因となった命令の先にあるいくつかの命令 (分岐やジャンプを含む) が実行される可能性があります。これらの命令は，失敗した命令が使用していた元のオペランドを上書きすることがあるため，例外の解釈や修正に不可欠な情報が失われます。例外ハンドラに報告される PC は，失敗した命令の

PC ではなく，いくらか先にある命令の PC になります。アプリケーションの例外ハンドラに例外が報告されたときに，ハンドラが入力データを修正したり，命令を再実行することはできません。

算術例外の報告におけるこの基本的な違いにより， OpenVMS Alpha オペレーティング・システムでは，新たに単一の条件コード SS$_HPARITH が定義されており，すべての算術例外を示します。 SS$_HPARITH の例外シグナル配列については，Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications を参照してください。[1]

推奨される変更

アプリケーションの条件処理ルーチンが，発生した算術例外の数をカウントしているだけの場合や，算術例外が発生したら強制終了する場合は， Alpha で例外が非同期に報告されることによる影響はありません。これらの条件処理ルーチンに条件コード SS$_HPARITH のテストを追加するだけで済みます。 VAX の算術例外は OpenVMS Alpha では返されません (トランスレートされた VAX イメージからの例外を除く)。例外の原因となった演算を再実行しようとしている場合は，コードを書き換えるか，算術例外が正確に報告されるようにするためのコンパイラ修飾子または指示文を使用します。ただし，この機能を利用するためには，コンパイラはそれぞれの算術命令を使用した後に命令パイプラインから命令を取り除く必要があるため，性能に大きく影響します。

EVAX_TRAPB ビルトインを使用すると，それまでのすべてのトラップがシグナル通知されます。このビルトインは性能への影響があるため，分離する必要がある算術命令の後でのみ使用してください。

3.9 ページ・サイズ

ページ・サイズへの依存をコーディングする方法を標準化し，将来の変更を容易にするために，一連のマクロが開発されました。これらのマクロについては付録 D で説明しています。

3.10 ワーキング・セットへのページのロック

通常，ページは次のようにロックされます。

- イメージの初期化の際，イメージが動作する間， 1 セクションのコードのロック。この方法では，システム・サービス$LKWSET を呼び出してページをメモリにロックします。多くの場合，$LKWSET の呼び出しは，ロック対象のコードとは別のモジュールにあります。

[1] このマニュアルはアーカイブ扱いになっています。このマニュアルの保守は行われておらず， OpenVMS ドキュメント・セットにも含まれていません。ただし，http://www.hp.com/go/openvms/docからオンラインで参照することができます。

- オンザフライでの，特定の操作の小さなコード・セクションのロックダウン。多くの場合，**未熟なプログラマのロックダウン**方法を使用します。この方法では，IPL を上げ，その IPL をロック対象コードの終わりのデータ位置として指定します。

OpenVMS Alpha システムでは，コードのページをメモリにロックする必要があるだけでなく，コードのリンケージ・セクションもロックする必要があります。この制約とその他の制約により， OpenVMS Alpha システムには LOCK_SYSTEM_PAGES, UNLOCK_SYSTEM_PAGES， PMLREQ，および PMLEND マクロがありません。

OpenVMS I64 システムでは， OpenVMS Alpha システムで発生した問題を避けるため，パラメータが参照しているイメージ全体をロックするように$LKWSET が変更されました。これにより正しい動作が保証されますが，性能が低下するおそれがあります。詳細は， OpenVMS V8.2 リリース・ノート[翻訳版]を参照してください。

推奨される変更

コードの変更を最低限にするため，どちらのケースもいくつかの新しいマクロで対処します。

イメージの初期化時に行われるロックダウンに対しては， 3 つのマクロが提供されています。そのうちの 2 つのマクロは「開始」マーカと「終了」マーカとして機能し，ロックするコードを区切ります。残りの初期化マクロは，記述子を作成し，$LKWSET を呼び出します。

未熟なプログラマのロックダウンやその他のオンザフライのロックダウンが行われていたコードをロックまたはアンロックするために，さらに 2 つのマクロが提供されています。この項で説明したアーキテクチャ上の理由から，これらのマクロはシステム・サービス呼び出し$LKWSET および$ULWSET も使用してページのロックとアンロックを行います。

ワーキング・セットにページをロックする必要があるユーザは，すでに特権コードを実行しているはずなので，これらのマクロは LIB.MLB にあります。

注意

これら 2 つの方法 (イメージの初期化時に行われるロックダウンとオンザフライのロックダウン) を 1 つのイメージで組み合わせて使用することはできません。オンザフライのロックダウンで実行される$ULWSET **サービスは，初期化時にロックされたセクションをアンロックすることもできます。イメージ全体に対してどちらかの方法を選択する必要があります。異なるソース言語からコンパイルされたモジュールが含まれるイメージでは，特に注意してください。**

OpenVMS Alpha と OpenVMS I64 のアーキテクチャによるページのロックへの影響

OpenVMS Alpha システムと OpenVMS I64 システムでのワーキング・セットへのページのロックは，OpenVMS VAX システムよりもはるかに複雑です。これには以下の理由があります。

- コードをメモリにロックするだけでは十分ではありません。コードはそのリンケージ・セクションを参照してさまざまな値やルーチンのアドレスを取得するため，リンケージ・セクションもロックする必要があります。なお，OpenVMS Alpha のリンケージ・セクションは，OpenVMS I64 のグローバル・データ・セグメントまたは GP 領域に相当します。GP 領域は，プログラム全体に対するデータ領域であり，OpenVMS Alpha の場合のように 1 つのルーチン用ではありません。
- コンパイラが実行する最適化により，ロックするコードの前後にラベルを追加するだけでは不十分です。コンパイラが，ラベル間にあるコードの一部をその範囲の外側に移動する可能性があるためです。
- 未熟なプログラマの，IPL を上げ，ロック対象コードの最後の位置として IPL を指定する方法は，上記の理由に加え，以下の理由からもうまく機能しません。
  - このシーケンスは，複数の命令の実行が必要なため，IPL を参照して自動的に IPL を上げることはできません。
  - OpenVMS Alpha と OpenVMS I64 のコンパイラ技術では，コードとデータ (たとえば，IPL が格納されたロングワード) が，同じプログラム・セクションに存在することは禁止されています。

OpenVMS Alpha または OpenVMS I64 でページをワーキング・セットにロックする唯一の方法は，システム・サービス$LKWSET を呼び出すことです。

プログラム・セクション (psect) を使用したコードの線引き

ここで示すマクロでは，psect を使用して，ロックするコードのセクションを囲みます。マクロは 3 つの psect を作成し，順番に名前を付け，以下のように使用します。

- コードの開始ラベルと終了ラベルを，それぞれ最初の psect と最後の psect に定義します。
- コード自体は中間の psect に配置します。

すべての psect の属性が同じだとすると，リンカはイメージ内に順番に psect を格納します。同じことが，2 番目の$LKWSET 呼び出しを必要とするリンケージ・セクションにも行われる必要があります。

イメージ内のさまざまな場所にあるロック対象のすべてのコードは同じ psect に格納されるため，この psect を使用した方法では，いずれかの要求元がいずれかのセクションをロックすると，ロック可能なすべてのコードがロックされるという副作用があります。多くの場合，これは利点となります。OpenVMS Alpha や OpenVMS I64 のページは 8KB 以上であるため，ほとんどの要求元はページレットまたはそれより小

さな領域をロックします。そのため，ほとんどの時間，ロック対象のすべてのコードが単一のページに納まります。

注意

IPL を，ページ・フォルトが発生しない 2 よりも上げることでコードをロックする場合は，その範囲内のコードが実行時ライブラリ・ルーチンやその他のプロシージャを呼び出さないことを確認してください。VAX MACRO**コンパイラは，特定の** VAX **命令をエミュレートするためのルーチンの呼び出しを生成します。これらのマクロを使用するイメージは，これらのルーチンの参照がページング不可のエグゼクティブ・イメージ内のコードで解決されるように，システム・ベース・イメージとリンクする必要があります。**

イメージの初期化時のロックダウン

イメージ初期化時のロックダウンでは，以下の 3 つのマクロを使用します。

- $LOCKED_PAGE_START
- $LOCKED_PAGE_END
- $LOCK_PAGE_INIT

マクロ$LOCKED_PAGE_START と$LOCKED_PAGE_END は，ロックするコード・セグメントの先頭と終わりをマークします。これらのマクロで線引きされたコードには，ルーチン全体が含まれている必要があります。実行がいずれかのマクロを超えたり，ロックされたコードから外に分岐したり，外からロックされたコードの中に分岐することはできません。ロックされたコード・セクションの中に分岐した場合や，ロックされたコード・セクションから外に分岐しようとした場合，またはマクロを超えて実行しようとした場合は，コンパイラから次のエラー・メッセージが出力されます。

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

$LOCKED_PAGE_END にはオプションのパラメータ LINK_SECT があります。これは，ルーチンを実行した後で戻るリンケージ psect を指定するために使用します。これは，$LOCKED_PAGE_START マクロが実行されたときの実際のリンケージ psect が，デフォルトのリンケージ psect ($LINKAGE) でなかった場合にだけ使用します。

マクロ$LOCK_PAGE_INIT は，$LOCKED_PAGE_START と$LOCKED_PAGE_END を使用してロック対象の領域を線引きしているイメージの初期化ルーチンで実行する必要があります。このマクロは必要な psect を作成し，$LKWSET を呼び出してコードとリンケージ・セクションをワーキング・セット内にロックします。 R0 と R1 の内容は，このマクロによって壊れます。

$LOCK_PAGE_INIT には，オプションのパラメータ ERROR があります。これは，いずれかの$LKWSET の呼び出しが失敗した場合に分岐するエラー・アドレスです。このアドレスに到達した場合， R0 には失敗した呼び出しの状態が設定され， R1

は，コードをロックするための呼び出しに失敗した場合は 0，その呼び出しは成功したものの，リンケージ・セクションをロックするための呼び出しが失敗した場合は 1 となります。

psect を使用してロックするコードを識別しているため，$LOCK_PAGE_INIT マクロは，$LOCKED_PAGE_START マクロおよび$LOCKED_PAGE_END マクロで線引きされたコードと同じモジュールにある必要はありません。$LOCK_PAGE_INIT を実行すると，イメージ全体の線引きされたコードがすべてロックされます。

表 3–1 に，これらのマクロを使用するために必要なコードの変更を示します。線引きするラベルは，$LOCKED_PAGE_START マクロと$LOCKED_PAGE_END マクロに置き換えられています。記述子は削除され，初期化コード中の$LKWSET の呼び出しは$LOCK_PAGE_INIT に置き換えられています。

表 3–1 イメージ初期化時のロックダウン

コード・セクション	VAX システム	Alpha システム
データ宣言		
	LOCK_DESCRIPTOR: .ADDRESS LOCK_START .ADDRESS LOCK_END	なし。記述子はすべて削除します。
初期化		
	$LKWSET_S LOCK_DESCRIPTOR BLBC R0,ERROR	$LOCK_PAGE_INIT ERROR
メイン・コード		
	LOCK_START: Routine_A: . . . RSB LOCK_END:	$LOCKED_PAGE_START Routine_A: . . . RSB $LOCKED_PAGE_END

他の言語で記述されたコードのロック

VAX MACROモジュール中で$LOCK_PAGE_INIT マクロを使用することで，他のプログラミング言語で記述されたコードもロックダウンすることができます。生成されたコードで psect $LOCK_PAGE_2を使用しており，生成されたリンケージ・セクションで psect $LOCK_LINKAGE_2を使用していれば，このマクロによって，任意の言語で記述された，任意のモジュール中の任意のコードがロックされます。

オンザフライのロックダウン

オンザフライのロックダウンでは，$LOCK_PAGE と$UNLOCK_PAGE がそれぞれロック対象コード・セクションの先頭と終わりをマークします。マークされたコードは，ロック対象 psect 内の独立したルーチンとなり，イメージ中のあらゆる場所にあるすべてのロック対象コードがこの psect に配置されます。

$LOCK_PAGE は，ロックされるルーチンのページとリンケージ・セクションをワーキング・セットにロックし， JSR でそこにジャンプします。このマクロは，実行可能コード内にインラインで配置されます。このマクロと対応する$UNLOCK_PAGE マクロの間のすべてのコードが，ロックされるルーチンに入れられ，ロックダウンされます。

$UNLOCK_PAGE はロックされたルーチンから戻り，ページとリンケージ・セクションをワーキング・セットからアンロックします。このマクロは，実行可能コードの，$LOCK_PAGE マクロの後のいずれかの場所に，インラインで配置します。

$LOCK_PAGE と$UNLOCK_PAGE には，オプションのパラメータ ERROR があります。これは，$LKWSET または$ULWSET の呼び出しに失敗した場合に分岐するエラー・アドレスです。$UNLOCK_PAGE には 2 番目のオプション・パラメータ LINK_SECT があります。 LINK_SECT は，$LOCK_PAGE マクロが実行されたときの実際のリンケージ psect が，デフォルトのリンケージ psect ($LINKAGE) でなかった場合に戻るリンケージ psect です。

どちらのマクロでも，すべてのレジスタが保存されます。ただし，エラー・アドレス・パラメータが指定されており，いずれかの呼び出しが失敗すると， R0 には失敗した呼び出しの状態が設定されます。コードのロックまたはアンロックの呼び出しに失敗した場合は R1 に 0 が設定され，その呼び出しは成功したものの，リンケージ・セクションのロックまたはアンロックの呼び出しが失敗した場合には 1 が設定されます。

$LOCK_PAGE マクロで制御がコードに渡り，$UNLOCK_PAGE マクロで制御が戻ります。$LOCK_PAGE マクロを実行したときに有効だったローカル・シンボル・ブロックは，$UNLOCK_PAGE マクロを実行すると復元されますが，ロックされたコードは独立したルーチンとなるため，ロックされたコード自体は個別のローカル・シンボル・ブロックを持ちます。名前付きシンボルが使用されている場合でも，ロックされたコード・セクションへの外からの分岐や，ロックされたコード・セクションから外への分岐は禁止されており，コンパイラで次のエラー・メッセージが出力されます。

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

ロックされるコードは個別のルーチンになり，スタック・コンテキストは同じでなくなるため，ルーチン内でのローカル・スタック・ストレージへの参照は，すべて変更する必要があります。

注意

オンザフライのロックダウンは，4 つのシステム・サービスの呼び出しと，実行の都度追加のサブルーチンを呼び出すことによるオーバヘッドがあるため，性能が重要なコードでロックダウンを行う場合は，初期化時のロックダウンに変更することをお勧めします。イメージ内の他のルーチンが初期化時のロックダウンを使用している場合は，オンザフライのロックダウンを初期化時のロックダウンに変更する必要があります。

表 3–2 に，オンザフライのロックダウンでこれらのマクロを使用するために必要な変更を示します。$UNLOCK_PAGE マクロが，実行されるように，そのマクロが RSB の前にある点に注意してください。ルーチンによって R0 と R1 で渡される情報は，$UNLOCK_PAGE によってそれらのレジスタが保護されるため，内容はそのままとなります。

表 3–2 オンザフライのロックダウン

コード・セクション	VAX システム	Alpha システム
メイン・コード		
	Routine_A: . . . SETIPL 100$ . . . RSB 100$: .LONG IPL$SYNCH	Routine_A: .JSB_ENTRY . . . $LOCK_PAGE . . . $UNLOCK_PAGE RSB

表 3–3 に，同じオリジナルのコードと，初期化時のロックダウンで必要な変更を示します。

表 3–3 同じコードのイメージ初期化時のロックダウン

コード・セクション	VAX システム	Alpha システム
初期化		
	なし。	$LOCK_PAGE_INIT
メイン・コード		
	Routine_A: . . . SETIPL 100$ . . . RSB 100$: .LONG IPL$SYNCH	$LOCKED_PAGE_START Routine_A: .JSB_ENTRY . . . RSB $LOCKED_PAGE_END

3.11 同期

同期に関する以下の情報は，OpenVMS VAX システムから OpenVMS Alpha システムまたは OpenVMS I64 システムにコードを移植する場合に該当します。

- メモリ中のアラインされたロングワードに対してロングワード演算を行うコードは，同期を追加しなくても Alpha システムと Itanium システムで動作します。これはアーキテクチャ的に保証されています。

  Alpha と Itanium のアーキテクチャでは，この保証が拡張され，メモリ中のアラインされたクォドワードに対するクォドワード演算でも動作が保証されます。ただし，これは VAX システムとの下位互換性はありません。Alpha または Itanium のコードだけがこの機能に依存できます。

- インターロックされる命令 (BBSSI，BBCCI，および ADAWI) は引き続き機能します。ただし，これを使用する場合には以下の点に留意してください。
  - これらの命令をコンパイルする際には，MACRO コンパイラはメモリ・バリア機能を暗黙的に提供します。
  - これらの命令は，バイト細分性環境を前提としています。これらの命令が操作するデータ・セグメントが，さまざまなスレッドから並行して書き込まれる可能性がある場合は，MACRO コンパイラの PRESERVE 機能を使用して，データ・セグメントのさらなる同期を行う必要があります。
  - バイト細分性の問題に対処すると同時に，高い性能を達成するもう 1 つの方法は，データ・セグメントがアンパックされるように再構成する方法です。つまり，BBSSI または BBCCI で変更されるビット，または ADAWI で変更されるワードは，並行して動作する別の命令スレッドから他の部分が変更されないようなロングワードの中にある必要があります。

問題のデータをさらに分割し，そのデータが含まれている，アラインされた128 バイトのロック範囲の任意の場所への独立した並列アクセスが起きないようにすると， Alpha での Load-locked/Store-conditional 命令の実装の多くでさらに性能が向上します。

- VAX のインターロックされるキュー命令は， OpenVMS Alpha システムまたは OpenVMS I64 システムでも変更なく動作し，必要なインターロックとメモリ・バリア機能を持った同等の PALcode が呼び出されます。

  インターロックされないキュー命令も同等の PALcode にコンパイルされ，単一のプロセッサ上では引き続き不可分となります。

- VAX の同期ツールは， OpenVMS Alpha および OpenVMS I64 でもそのまま動作します。以下のすべてのメカニズムは，同期のために，インターロックされる命令を直接または間接的に使用します (使用されているインターロックされる命令は，透過的にメモリ・バリアを提供します)。
  - イベント・フラグ — 操作するすべてのシステム・サービス
  - スピンロック — 取得と解放の演算子すべて (LOCK と UNLOCK, FORKLOCK と FORKUNLOCK， DEVICELOCK と DEVICEUNLOCK)
  - ミューテクス — スピンロックにより保護
  - ロック・マネージャ — スピンロックにより保護

注意

この同期の保証は，マルチプロセシング・システムにのみ該当します。ユニプロセシング版のスピンロックは，インターロックされる命令を使用しません。その結果，ユニプロセッサのスピンロック，ミューテクス，ロック・マネージャの同期では，メモリ・バリアは提供されません。

- AST での並列スレッドと不可分性については，コードを再設計するか，コンパイラが提供する機能を使用して不可分性を適用する必要があります。 MACRO コンパイラには PRESERVE 機能があります。
- OpenVMS Alpha では，例外ハンドラを変更するコードは，処理待ちの算術トラップとマシン・アボートのいずれかまたは両方が非同期に発生する可能性がある場合は，変更が必要な場合があります。TRAPB 命令は，必要な同期メカニズムを提供します。ハンドラを変更しているときに同期を適用したい場合は，次の例に示すように，手動でプログラムに追加する必要があります。

```
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
```

- OpenVMS Alpha アセンブリ言語コードを記述するときには， Alpha アーキテクチャでの読み書き順序について理解してください。必要に応じて MB 命令をコードに追加してください。

4

移植したコードの性能改善

この章では，移植したコードの性能を改善する方法について説明します。

この章のトピックは以下のとおりです。

- 第 4.1 節, データのアラインメント
- 第 4.2 節, コード・フローと分岐予測
- 第 4.3 節, コードの最適化
- 第 4.4 節, 共通ベース参照

4.1 データのアラインメント

アラインされていないデータ参照は， OpenVMS Alpha や OpenVMS I64 でも動作しますが，低速です。システムは，アラインされていないアドレスのフォルトを取得して，アラインされていない参照を完了させる必要があるためです。データの参照がアラインされていないことが分かっている場合は，コンパイラは，手動でデータを抽出するための，アラインされていないクォドワードのロードとマスクを生成することができます。これはアラインされているロードより低速ですが，アラインメント・フォルトが発生する場合よりは高速です。ロングワードまたはクォドワードにアラインされていないグローバル・データ・ラベルに対しては，情報レベルのメッセージが表示されます。

また，/PRESERVE=ATOMICITY や.PRESERVE ATOMICITY を使用して，アラインされていないメモリ変更の参照を不可分にすることはできません。このような参照を行うと，回復不可能な予約オペランド・フォルトとなります。

4.1.1 アラインメントの想定

デフォルトでは，コンパイラは以下のことを想定します。

- ベース・ポインタとして使用されるレジスタ内のアドレスは，ルーチンの入口でロングワードにアラインされている。
- 外部参照はロングワードにアラインされている。
- DIVL などの特定の種類の命令から得られたアドレスはアラインされていない。

コンパイラは，レジスタが変更されるたびに，レジスタ内のベース・アドレスがまだアラインされているかどうかを確認します。レジスタと指定されたオフセットでアラインされたアドレスになる場合は，コンパイラはメモリ参照に対してアラインされたロードまたは格納を使用します。コンパイラは，レジスタの使用を追跡し，ベース・アドレスがアラインされているかどうかを確認します。たとえば， MOVL 4(R1),R0 などで，格納されているメモリ・アドレスをロードするときや， MOVL @4(R1),R0 などで間接的に使用される場合は，コンパイラはアドレスがアラインされていると想定します。

MOVQ 命令などのクォドワード・メモリ参照では，レジスタ追跡コードによってアドレスがロングワードにアラインされていない可能性があることが分かっていないかぎり，コンパイラはベース・アドレスがクォドワードにアラインされているものと想定します。言い換えれば，クォドワードのレジスタ・アラインメントは追跡されません。ロングワードのアラインメントだけが追跡されます。

次の例のような， OpenVMS Alpha または OpenVMS I64 のビルトインでのクォドワードの参照は，新しいコードなので，アラインメントは正しいはずです。そのため，次の例のすべてのメモリ参照では，アラインされたクォドワードのロードと格納が使用されます。

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

OpenVMS Alpha または OpenVMS I64 のビルトイン (EVAX_LDQU と EVAX_STQU 以外) を，クォドワードにアラインされていないアドレスに対して使用すると，実行時にアラインメント・フォルトが発生します。

4.1.2 アラインメントの想定を変更する指示文と修飾子

コンパイラには，コンパイラのアラインメントの想定を変更するための， 2 つの指示文と 1 つの修飾子があります。どちらの指示文を指定しても，コンパイラはより効率の良いコードを生成できます。.SET_REGISTERS 指示文では，レジスタがアラインされているかそうでないかを指定できます。この指示文は，演算の結果が，コンパイラが想定するものと逆である場合に使用します。また，指示文を使用しないとコンパイラが入力レジスタや出力レジスタとして検出しないレジスタを宣言することもできます。

.SYMBOL_ALIGNMENT 指示文では，シンボリック・オフセットを使用するメモリ参照のアラインメントを指定することができます。この指示文は，シンボリック・オフセットが使用されているすべての箇所でデータがアラインされていることが分かっている場合に使用します。

これらの指示文の説明は，付録 B にあります。それぞれの説明の例で使用方法を示しています。

MACRO/MIGRATION コマンドに対する/UNALIGN 修飾子は，コンパイラに対し，アラインメントを追跡せずに，すべてのレジスタ・ベースのメモリ参照がアラインされていないものと想定するように指示します。これは，コンパイラがアラインメントを知っているスタック・ベースの参照や静的な参照には影響を与えません。

この修飾子の説明は付録 A にあります。

4.1.3 アラインメント制御の優先順位

コンパイラのアラインメント制御の優先順位を，優先順位の最も高いもの (.SYMBOL_ALIGNMENT) から最も低いもの (組み込みの想定と追跡メカニズム) の順に並べると次のようになります。

1. .SYMBOL_ALIGNMENT 指示文
2. .SET_REGISTER 指示文
3. /UNALIGN 修飾子
4. 組み込みの想定と追跡メカニズム

4.1.4 データのアラインメントでの推奨事項

データのアラインメントでは次の推奨事項に従ってください。

- /PRESERVE=ATOMICITY または.PRESERVE ATOMICITY を使用してデータの参照を不可分にする必要がある場合は，データはアラインされている必要があります。
- パブリック・インタフェースでのアラインメントの問題は修正しないでください。既存のプログラムが正常に動作しなくなるおそれがあります。
- 内部インタフェースまたは特権インタフェースのデータでは，データのアラインメントを改善するための変更を自動的に行わないでください。データ構造にアクセスする頻度，構造をアラインし直すのに要する作業量，誤りが発生する危険性を考慮してください。必要な作業量を判断する際には，単に推測するのではなく，データに対するすべてのアクセスを確認してください。担当しているコードのすべてのアクセスを把握しており，いずれにしてもモジュールを変更する予定である場合は，アラインメントの問題を修正しても安全です。
- バイトまたはワードのデータを機械的にロングワードまたはクォドワードにアンパックしないでください。このような修正を行うのは，前述の注意および制約に従って，アラインメントの問題 (ワード境界にないワード) を修正する場合や，データの細分性に問題があることが分かっている場合です。
- データに対するすべてのアクセスを把握していない場合でも，アラインメントを修正するのが適切であるような状況があります。データに頻繁にアクセスする場合で，性能が問題となっており，データ構造を見直すことが不可避な場合は，同時に構造をアラインすることに意味があります。

作成しているコードに影響がある他のプログラマにも通知することが重要です。関連するすべてのモジュールが再コンパイルされるものと想定したり，データ・セルの分離に対する想定が誤っていることを，他の人がプログラムのドキュメントからを見つけてくれるとは思わないでください。このような変更によって，不適切なプログラミングが露呈したり，変更が順調に進まないことがあるということを常に想定してください。

4.2 コード・フローと分岐予測

Alpha アーキテクチャと Itanium アーキテクチャはパイプライン化されており，現在の命令が完了する前に，それよりも先のいくつかの命令を実行し始めます。パイプラインに命令が満たされた状態を保つようにコードを調整することで，はるかに高速にコードを動作させることができます。

Alpha アーキテクチャと Itanium アーキテクチャでは，次に実行する命令が命令パイプラインに正しく満たされるように，それぞれの条件分岐で分岐が発生するかどうかを予測しようとします。 Alpha アーキテクチャでは，前方条件分岐は成立せず，後方条件分岐が成立するものと予測されます。 Itanium アーキテクチャでは，分岐命令に分岐予測ヒントが埋め込まれています。分岐予測が外れると，パイプラインをフラッシュしなくてはならない上に，分岐先の命令が命令キャッシュにない可能性があるため，余分に時間がかかります。

コンパイラは，パイプライン化された Alpha アーキテクチャと Itanium アーキテクチャでコードを高い効率で実行できるように，VAX MACROコードのフローに従って，最もよく実行されるコード・パスが連続したブロックになるような Alpha および Itanium のコードを生成しようとします。しかし，場合によっては，コンパイラのデフォルトの規則では最も効率の良いコードが生成されないことがあります。高い性能が要求されるコード・セクションでは，実行される可能性がもっと高いコード・パスをコンパイラに教えることで，生成されるコードの効率を高めることができます。

4.2.1 デフォルトのコード・フローと分岐予測

特に指定しないかぎり，コンパイラは通常，VAX MACROの無条件分岐をたどり，VAX MACROの条件分岐は不成立とする経路で Alpha と Itanium のコードを生成します。たとえば，次のVAX MACROのコード・シーケンスを考えます。

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

このシーケンスに対して生成される Alpha コードは，次のようになります。

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

コンパイラは BLBS 命令を不成立とし， BLBS の直後の命令から続行している点に注意してください。コンパイラは， BRB 命令で，分岐命令は全く生成せず，コード・ブロック C から生成された Alpha と Itanium のコードの後に，分岐先のコード・ブロック E から生成された Alpha と Itanium のコードを続けています。ラベル 20$にあるコード・ブロック D のコードは，ルーチンの後のほうで生成されます。ラベル 20$への分岐がない場合は，コンパイラは次の情報メッセージを出力し，コード・ブロック D に対する Alpha または Itanium のコードを生成しません。

```
UNRCHCODE, unreachable code
```

ほとんどの場合，このアルゴリズムによって，アーキテクチャの想定に合った Alpha と Itanium のコードが生成されます。

- VAX MACROコードの条件分岐が後方分岐の場合は，分岐先はすでに Alpha および Itanium のコードが生成されている可能性が高いため，生成される分岐も後方となります。
- VAX MACROコードの条件分岐が前方分岐の場合は，分岐先は Alpha および Itanium のコードがまだ生成されていない可能性が高いため，生成される分岐も前方となります。

ただし，コンパイラは無条件分岐をたどるため，VAX MACROの後方分岐の分岐先はまだ生成されていない可能性があります。この場合，VAX MACROのソース・コードで後方分岐だった条件分岐が，生成される Alpha および Itanium のコードでは前方分岐になることがあります。この問題の詳細と対処方法については第 4.2.5 項を参照してください。

前方分岐が成立するとコンパイラが想定するケースがあります。たとえば，次の一般的なVAX MACROのコーディング・スタイルを考えます。

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

このように，分岐に続くインライン・コードが数行しかなく，分岐先でコード・フローに合流しない場合は，前方分岐を成立と判断します。これにより， OpenVMS Alpha システムと OpenVMS I64 システムでの分岐予測ミスによる遅延をなくすことができます。コンパイラは，分岐の見方を自動的に変更し，分岐とラベルの間のコードを，ルーチンの通常の出口の先に移動させます。この例に対しては，次のコードが生成されます。

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

4.2.2 コンパイラの分岐予測の変更

コンパイラでは 2 つの指示文.BRANCH_LIKELY と.BRANCH_UNLIKELY が提供され，分岐予測に関する想定を変更することができます。指示文.BRANCH_LIKELY は，分岐の可能性が高い (たとえば 75 パーセント以上) 場合に前方条件分岐に対して使用します。指示文.BRANCH_UNLIKELY は，分岐の可能性が低い (たとえば 25 パーセント未満) 場合に後方条件分岐に対して使用します。

これらの指示文は，高い性能が要求されるコードでのみ使用してください。また，.BRANCH_UNLIKELY を追加すると，分岐が実際に成立した場合のパスに分岐命令が追加されるため，この指示文を追加する場合には特に注意してください。つまり，後方分岐が分岐命令の前方分岐に変更され，そこからさらに元の分岐先に分岐します。

無条件分岐をたどらないようにコンパイラに指示する指示文はありません。ただし，分岐をたどらないコードをコンパイラに生成させたい場合は，無条件分岐を，常に成立することが分かっている条件分岐に変更することができます。たとえば，現在のコード・セクションで， R3 に常にデータ構造のアドレスが格納されていることが分かっている場合は， BRB 命令を TSTL R3 に変更し，その後に BNEQ 命令を続けます。この分岐は常に成立しますが，コンパイラはそのまま次に進み，次の命令のコード生成を続行します。これを実行すると，常に分岐予測ミスが発生しますが，場合によっては有効です。

4.2.3 .BRANCH_LIKELY の使用方法

成立する可能性が非常に高い前方条件分岐がコードにある場合は，分岐命令の直前に.BRANCH_LIKELY 指示文を挿入することで，その想定を使用してコードを生成するようにコンパイラに指示することができます。例を示します。

```
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$:
```

コンパイラは分岐をたどり，前の例で説明したように，足りない構造を扱うすべてのコードをモジュールの最後に移動させることで，コードのフローを変更します。

4.2.4 .BRANCH_UNLIKELY の使用方法

成立する可能性が非常に低い後方条件分岐がコードにある場合は，分岐命令の直前に.BRANCH_UNLIKELY 指示文を挿入することで，その想定を使用してコードを生成するようにコンパイラに指示することができます。例を示します。

```
        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
```

ここで.BRANCH_UNLIKELY 指示文を使用している理由は，コンパイラが 10$への後方分岐を成立する可能性が高いと予測するためです。プログラマは，それがまれなケースであることを知っているため，分岐命令を，成立と予測されない前方分岐に変更するために指示文を使用しています。

前方分岐の分岐先に無条件分岐命令があり，その分岐命令で元の分岐先に戻ります。この場合も，このコードは，ルーチンの通常の出口の後に移動されます。上のVAX MACROコードから生成されるコードは，次のようになります。

```
        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 ループへの前方ジャンプ

コンパイラがコード・フローをたどる方式であることが原因となり，ループの中に前方分岐した場合は，効率の良いコードにコンパイルされないケースあります。通常，この場合には，ループが 2 つの部分に大きく分割されたコードが生成されます。たとえば，次のマクロ・コーディング構造を考えます。

```
        (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)
```

MACRO コンパイラは，コード・フローを生成する際に BRB 命令をたどり，その後の 10$への条件分岐にたどり着きます。しかし，10$のコードは，BRB 命令によってスキップされるため，ルーチンの最後になるまで生成されません。そのため，条件分岐は後方分岐ではなく前方分岐に変換されます。生成されるコードの配置は次のようになります。

```
        (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$:    (Move block pointer to next section to be moved)
        BRB     20$
```

その結果，10$への分岐が常に不成立と予測され，コード・フローは2つの箇所の間で常に行ったり来たりするため，ループは非常に遅くなります。この状況は，10$に戻る条件分岐の前に.BRANCH_LIKELY 指示文を挿入することで解消できます。これにより，次のようなコード・フローとなります。

```
        (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 コードの最適化

MACRO コンパイラは，生成されるコードに対してさまざまな最適化を行います。デフォルトでは，VAXREGS を除き，すべての最適化を実行します (VAXREGS は OpenVMS Alpha システムでのみ使用できます)。これらのデフォルトの動作は，コマンド行の/OPTIMIZE スイッチで変更できます。有効なオプションは以下のとおりです。

- ADDRESSES

  コンパイラは，同じアドレスが複数回参照されるのを認識し，アドレスを一度だけロードして，複数の参照で使用します。

- REFERENCES

  コンパイラは，同じデータ値が複数回参照されるのを認識し，一度だけデータをロードして，複数の参照で使用します。ただし，使用するデータが古くなっていないことを確認するという制限に従います。

- PEEPHOLE

  コンパイラは，より小さな命令シーケンスで同じように実行できる命令シーケンスを識別し，長いシーケンスを短いシーケンスに置き換えます。

- SCHEDULING

  コンパイラは，Alpha アーキテクチャや Itanium アーキテクチャで複数の命令を実行できるという知識を使用し，最適な性能が得られるようにコードをスケジュールし直します。

- VAXREGS (Alpha システムのみ)

  デフォルトでは，レジスタ R13 ～ R28 は，ソース・コード中で使用されていなければ，コンパイラにより一時的なスクラッチ・レジスタとして使用されます。VAXREGS を指定すると，コンパイラは MACRO ソース・コードで明示的に使用されていない VAX レジスタ・セット (R0 ～ R12) も使用します。このようにして

使用されたVAXレジスタは，SCRATCHが宣言されていないかぎり，ルーチンの出口で元の値に復元されます。

注意

最適化にはコードの再配置や再スケジューリングが含まれているため，/NOOPTIMIZEを指定するとデバッグが容易になります。詳細は，第2.13.1項を参照してください。

4.3.1 VAXREGSを使用した最適化 (OpenVMS Alphaのみ)

VAXREGSを使用して最適化する場合は，すべてのルーチンで，使用しているレジスタがルーチン宣言.CALL_ENTRY，.JSB_ENTRY，.JSB32_ENTRYで正しく宣言されていることを確認する必要があります。また，呼び出されるルーチンが必要とするVAXレジスタや変更するVAXレジスタを明確にする必要があります。デフォルトでは，呼び出されているどのルーチンでも入力としてVAXレジスタを必要とせず，R0とR1以外のすべてのVAXレジスタが呼び出しの前後で保護されるものと想定されます。使用されていることを宣言するには，次の例のように，コンパイラ指示文.SET_REGISTERSの修飾子READおよびWRITTENを使用します。

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

この例では，コンパイラは，R3とR4がルーチンDO_SOMETHING_USEFULに対する入力で必要であり，R5がルーチンで上書きされることを認識します。レジスタの使用状況は，DO_SOMETHING_USEFULのinputマスクをREAD修飾子として使用し，outputマスクとscratchマスクを組み合わせたものをWRITE修飾子として使用することで判断できます。

注意

ルーチンのエントリ・ポイントとルーチンの呼び出しの両方でレジスタを正しく宣言せずにVAXREGS修飾子を使用すると，誤ったコードが生成されます。

4.4 共通ベース参照

OpenVMS Alphaシステムでは，一般にデータ・セルを参照する場合，リンケージ・セクションからデータ・セル・アドレスをロードするための参照と，データ・セル自体の参照の，合計2回メモリを参照する必要があります。いくつかのデータ・セルが互いに近くにあり，ADDRESSES最適化を使用した場合，コンパイラは共通のベース・アドレスをレジスタにロードし，個々のデータ・セルをベース・アドレスからのオフセットとして参照します。これにより，データ・セル・アドレスを個別にロードする必要がなくなります。これは，**共通ベース参照**と呼ばれます。

ADDRESSES 最適化が有効になっていると，コンパイラはローカルなデータ psect に対して自動的にこの最適化を実行します。コンパイラは，$PSECT_BASEnの形式のシンボルを生成して，ローカル psect のベースとして使用します。

外部データ psect に対して共通ベース参照を使用するためには，シンボルを共通ベースからのオフセットとして定義したプレフィックス・ファイルを作成する必要があります。VAX MACROアセンブラでは，シンボルを外部シンボルからのオフセットとして定義できないため，モジュールを OpenVMS VAX 用にアセンブルする際にはプレフィックス・ファイルは使用できません。

4.4.1 共通ベース参照のためのプレフィックス・ファイルの作成

次の例は，共通ベース参照を使用するためにプレフィックス・ファイルを作成することの利点を示しています。次の内容を示します。

- プレフィックス・ファイルを使用せずに生成されたコード
- プレフィックス・ファイルの作成方法
- プレフィックス・ファイルを使用して生成されたコード

次の単純なコード・セクション (CODE.MAR) について考えます。これは，別のモジュール (DATA.MAR) のデータ・セルを参照しています。

```
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
```

共通ベース参照を使用せずに CODE.MAR をコンパイルすると，次のコードが生成されます。

リンケージ・セクションに生成されるコード

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

コード・セクションに生成されるコード (プロローグ・コードとエピローグ・コードは除く)

```
        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
```

外部データ・セルを共通のベース・アドレスからのオフセットとして定義したプレフィックス・ファイルを作成することで，コンパイラが外部参照に対して共通ベース参照を使用するようにすることができます。この例のプレフィックス・ファイルでは，BASE からの相対として A， B， C， D を定義しています。

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

このプレフィックス・ファイルと最適化 ADDRESSES を使用して CODE.MAR をコンパイルすると，次のコードが生成されます。

リンケージ・セクションに生成されるコード

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

コード・セクションに生成されるコード (プロローグ・コードとエピローグ・コードは除く)

```
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
```

この例では，共通ベース参照によって，コード・セクションとリンケージ・セクションのサイズがどちらも小さくなり， 3 つのメモリ参照が除去されています。このように，プレフィックス・ファイルを作成し，外部データ・セルの共通ベース参照を有効にする方法は，多数のモジュールから使用されるデータ領域を定義した大きな単独のモジュールがある場合に有効です。

4.4.1.1 OpenVMS I64 システムでのコード・シーケンスの相違点

OpenVMS I64 システムでも，第 4.4.1 項に示した効果と同じ効果が得られますが，コード・シーケンスの詳細が異なります。同じコードに対して，次のような命令シーケンスが生成されます。

```
add     r19 = D, r1
add     r22 = C, r1
add     r23 = B, r1
add     r24 = A, r1
ld8     r19 = [r19]
ld8     r24 = [r24]
ld8     r22 = [r22]
ld8     r23 = [r23]
ld4     r4 = [r19]
ld4     r9 = [r24]
ld4     r3 = [r22]
ld4     r28 = [r23]
sxt4    r4 = r4
sxt4    r9 = r9
sxt4    r3 = r3
sxt4    r28 = r28
```

第 4.4.1 項に示したプレフィックス・ファイルの方法を使用することで，メモリ・アクセスのほぼ半分が除去された命令シーケンスが得られます。

```
add     r24 = BASE, r1
ld8     r24 = [r24]
mov     r23 = r24
ld4     r9 = [r24]
adds    r18 = 12, r24
adds    r19 = 8, r24
adds    r22 = 4, r24
adds    r24 = 12, r24
ld4     r4 = [r24], -4
sxt4    r9 = r9
ld4     r3 = [r24], -4
sxt4    r4 = r4
sxt4    r3 = r3
ld4     r28 = [r24], -4
sxt4    r28 = r28
```

5

MACRO の 64 ビット・アドレッシングのサポート

この章では， MACRO コンパイラとそれに関連する構成要素による 64 ビット・アドレッシングのサポートについて説明します。

この章のトピックは以下のとおりです。

- 第 5.1 節, 64 ビット・アドレッシングの構成要素
- 第 5.2 節, 64 ビット値の受け渡し
- 第 5.3 節, 64 ビット引数の宣言
- 第 5.4 節, 64 ビット・アドレス計算の指定
- 第 5.5 節, 符号拡張とチェック
- 第 5.6 節, Alpha 命令ビルトイン
- 第 5.7 節, ページ・サイズに依存する値の計算
- 第 5.8 節, 64 ビット・アドレス空間内でのバッファの作成と使用
- 第 5.9 節, 64 KB を超える移動のコーディング

5.1 64 ビット・アドレッシングの構成要素

MACRO コンパイラでは， 64 ビット・アドレッシングをサポートする構成要素が提供されています。

64 ビット・アドレッシングは， OpenVMS Alpha または OpenVMS I64 に移植したコードでのみ使用してください。 OpenVMS Alpha または OpenVMS I64 上で新たに開発する場合は，高級言語を使用することをお勧めします。

信頼性が高く保守が簡単なコードを生成できるように提供されている，修飾子，マクロ，指示文，ビルトインを使用して， 64 ビット・アドレッシングをコード中で明示的に行う必要があります。

表 5–1 に， MACRO プログラミングでの 64 ビット・アドレッシングのサポートを提供する構成要素を示します。

表 5–1 64 ビット・アドレッシング用の構成要素

構成要素	説明
$SETUP_CALL64	呼び出し手順を初期化するマクロ
$PUSH_ARG64	引数のプッシュ相当の処理を実行するマクロ
$CALL64	対象ルーチンを呼び出すマクロ
$IS_32BITS	64 ビット値の下位 32 ビットの符号拡張をチェックするマクロ
$IS_DESC64	記述子が 64 ビット形式の記述子かどうかを判断するマクロ
QUAD=NO/YES	64 ビット仮想アドレスをサポートするためのページ・マクロのパラメータ
/ENABLE=QUADWORD	64 ビット・アドレスの計算を含むように拡張された QUADWORD パラメータ
.CALL_ENTRY QUAD_ ARGS=TRUE | FALSE	QUAD_ARGS=TRUE | FALSE。引数リストに対するクォドワード参照の有無を示す (第 5.3 節を参照)
.ENABLE QUADWORD .DISABLE QUADWORD	64 ビット・アドレスの計算を含むように拡張された QUADWORD パラメータ
EVAX_SEXTL	64 ビット値の下位 32 ビットを符号拡張してデスティネーションに格納するビルトイン
EVAX_CALLG_64	可変サイズの引数リストを使用した 64 ビット呼び出しをサポートするためのビルトイン
$RAB64 および$RAB64_STORE	64 ビット・アドレス空間のバッファを使用するための RMS マクロ

5.2 64 ビット値の受け渡し

64 ビット値を渡すために使用する方法は，引数リストのサイズが固定か可変かによって変わります。ここでは，これらの方法について説明します。

5.2.1 固定サイズの引数リストを使用した呼び出し

固定サイズの引数リストを使用した呼び出しでは，表 5–2 に示すマクロを使用します。

表 5–2 固定サイズの引数リストを使用した 64 ビット値の受け渡し

使用する手順	使用するマクロ
1. 呼び出し手順の初期化	$SETUP_CALL64
2. 呼び出し引数の「プッシュ」	$PUSH_ARG64
3. 対象ルーチンの呼び出し	$CALL64

これらのマクロの使用例を以下に示します。引数は逆の順序でプッシュされる点に注意してください。これは， 32 ビットの PUSHL 命令を使う場合でも同様です。

```
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
```

$SETUP_CALL64 マクロは， 64 ビット呼び出しの状態を初期化します。$PUSH_ARG64 や$CALL64 を使用する前に必要です。引数の数が OpenVMS Alpha では 6, OpenVMS I64 では 8 を超える場合は，このマクロはローカルな JSB ルーチンを作成します(このルーチンは，呼び出しを実行するために呼び出されます)。そうでない場合は，引数のロードと呼び出しはインラインとなり，非常に効率が良くなります。$SETUP_CALL64 で指定する引数カウントには，シャープ(#)は含まれないことに注意してください。(標準的な呼び出し手順では，スタックはオクタワードでアラインされ，スタックの先頭に引数が置かれている必要があります。 JSB ルーチンはこのアラインメントを手助けします。)

インライン・オプションを使用して， 6 個または 8 個を超える引数を使用した呼び出しを，ローカルな JSB ルーチンを使用せずに行うことができます。ただし，その利用には制限があります(付録 E を参照)。

$PUSH_ARG64 マクロは，引数を正しい引数レジスタまたはスタック領域に直接移動します。これは実際にはスタックのプッシュではなく， 32 ビット呼び出しで使用する PUSHL 命令を模したものです。

$CALL64 マクロは，引数カウント・レジスタを設定し，対象ルーチンを呼び出します。 JSB ルーチンが作成されている場合は，そのルーチンを呼び出します。プッシュした引数の数が$SETUP_CALL64 で指定した数と一致しない場合，このルーチンはエラーを報告します。$CALL64 と$PUSH_ARG64 は，$SETUP_CALL64 がすでに呼び出されていることを確認します。

$SETUP_CALL64，$PUSH_ARG64，および$CALL64 を使用する際には，以下の点に注意してください。

- 引数は，アラインされたクォドワードとして読み込まれます。メモリにあるロングワードを渡すには，第 5.2.1 項の例に示したように，レジスタに移動してからそのレジスタを$PUSH_ARG64 で使用します。同様に，渡すクォドワードがアラインされていないことが分かっている場合は，まず値をレジスタに移動します。また， (R4)[R0]のようにインデックス指定されたオペランドは，$PUSH_ARG64 で使用されるときにクォドワード・インデックスを使用して評価される点に注意してください。
- 引数の数が OpenVMS Alpha では 6， OpenVMS I64 では 8 を超え，ローカルな JSB ルーチンが作成される場合は，$SETUP_CALL64 と$CALL64 の間での SP や AP の参照は行えません。$PUSH_ARG64 マクロと$CALL64 マクロは，オペランドでのこれらのレジスタの使用を報告します。しかし，この範囲にある他の命令では使用できませんが，使用してもエラーは報告されません。このような場

合，AP または SP ベースの引数を渡すには，$SETUP_CALL64 を呼び出す前にレジスタに移動します。

- OpenVMS Alpha システムのみ: 引数の数が 6 を超える場合は，$SETUP_CALL64 を実行した後で R15 以降のレジスタの値が保持されていることを期待しないでください。代わりに，スクラッチ・レジスタ以外のレジスタを一時レジスタとして使用してください。たとえば，スタック領域から取り出したデータを渡したいとし，呼び出しのパラメータが 7 個以上あるとします。この場合，値をレジスタに移動する必要があります。R28 などのスクラッチ・レジスタを使用せずに，R0 などの VAX レジスタを使用してください。VAX レジスタがすべて使用中の場合は，R13，R14，または R15 を使用してください。
- OpenVMS Alpha システムのみ: $SETUP_CALL64 と$CALL64 の間で R16 以降のスクラッチ・レジスタを使用できます。ただし，すでにロード済みの引数レジスタを使用しないように注意してください。引数レジスタは，R21 から R16 へ，小さいほうに向かってロードされます。そこで，呼び出しで 6 つの引数が渡されると仮定します。$PUSH_ARG64 によって R21 がロードされるため，最初の$PUSH_ARG64 の呼び出しの後で R21 を使用するのは危険です。$PUSH_ARG64 マクロは，オペランドがすでにロードされている引数レジスタを参照していないかをチェックします。そのようなオペランドがみつかると，コンパイラは警告を報告します。一時レジスタが必要な場合の最も安全なアプローチは，レジスタ R22 ～ R28 を使用することです。

注意

マクロ$SETUP_CALL64，$PUSH_ARG64，**および**$CALL64 **は，インライン・シーケンスで使用されることを意図しています。つまり**$SETUP_CALL64/$PUSH_ARG64/$CALL64 **シーケンスの途中に分岐したり，**$PUSH_ARG64 **マクロの前後で分岐したり，**$CALL64 **を呼び出さずにこのシーケンスから外に分岐することはできません。**

$SETUP_CALL64，$PUSH_ARG64，および$CALL64 についての詳細は，付録 E を参照してください。

5.2.2 可変サイズの引数リストを使用した呼び出し

可変サイズの引数リストを使用した呼び出しでは，以下の手順に示すように EVAX_CALLG_64 ビルトインを使用します。

1. インメモリ引数リストを作成します。
2. インメモリ引数リストを渡してルーチンを呼び出します。例を示します。

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

EVAX_CALLG_64 ビルトインの引数リストは，クォドワードの引数カウントから始まる一連のクォドワードとして読み込まれます。

5.3 64 ビット引数の宣言

.CALL_ENTRY のパラメータ QUAD_ARGS=TRUE を使用し，ルーチンの引数リスト中でのクォドワード引数の使用を宣言することができます。 QUAD_ARGS パラメータがあると，コンパイラは，引数リストに対するクォドワード参照があったときに異なる動作をします。まず，通常このような参照で必要となる引数リストのホーミングを適用しません。(ホーミングは，その定義上，引数をロングワード・スロットにパックするため，クォドワード値が格納された引数リストはホーミングできません。) また，引数リストに対するクォドワード参照では，**アラインされていないメモリ参照**は報告されません。

引数リスト参照自体に対して生成される実際のコードは， MOVQ のように VAX のクォドワード命令の中に参照がある場合を除いて， QUAD_ARGS 句があっても変わりません。たいていの場合， QUAD_ARGS は，クォドワード参照による引数リストのホーミングが行われないようにし，不要なアラインメント・メッセージを抑止するだけです。この抑止は， EVAX_ビルトインと， MOVQ などの VAX のクォドワード命令の両方に適用されます。

VAX のクォドワード命令では， QUAD_ARGS 句を指定すると， EVAX_ビルトインと同じように，クォドワード引数を実際のクォドワードとして読み込みます。次の例を考えます。

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

QUAD_ARGS 句を指定すると， MOVQ は引数 1 の 64 ビット全体を 8(R2) の位置のクォドワードに格納します。 QUAD_ARGS 句が指定されていないと， MOVQ は引数 1 と 2 の下位ロングワードを 8(R2) の位置のクォドワードに格納します。

QUAD_ARGS は， AP ベースの遅延モードのオペランドに対して生成されるコードにも影響を与えます。実効アドレスをメモリ内の引数からロードする必要がある場合は， QUAD_ARGS が有効になっていると，ロングワードとしてではなくクォドワードとして読み込まれます。

QUAD_ARGS を使用する場合には以下の点に注意してください。

- AP ベースのクォドワード引数リスト参照は，重なっているように見えるため，一見するとおかしいように思われます。この状況は，たとえば， FIRST_ARG, SECOND_ARG など，引数リスト・オフセットにシンボリック名を定義することで，改善することができます。ソース・コードの可読性を高めるために，引数の使用目的を示す意味のあるシンボリック名を定義することをお勧めします。また，引数レジスタの直接参照を使用し， OpenVMS Alpha システムでの最初の 6 個の引数を参照することができます。(OpenVMS I64 では，引数レジスタにアクセスすることはできません。) いずれにしても， QUAD_ARGS を宣言して，引数リストがホーミングされないようにすると便利です。

- コードを共用するルーチンでは，QUAD_ARGS の設定を同じにする必要があります。同じになっていないと，コンパイラから警告メッセージが報告されます。
- JSB ルーチンは，呼び出し元で QUAD_ARGS が宣言されていると，呼び出し元の引数リストを参照できません。JSB ルーチン内で AP を参照するためには，最後の CALL_ENTRY の引数リストがホーミングされている必要があります。HOME_ARGS と QUAD_ARGS は同時に使用できません。
- QUAD_ARGS を宣言すると，$ARGnシンボルがロングワードではなくクォドワードとして定義されます (このシンボルは，コンパイラによってデバッグ・シンボル・テーブルに格納されます)。これらのシンボルを使用することで，受け取った引数の値に容易にアクセスできるようになり，シンボリック・デバッガでデバッグする際に，レジスタ番号やスタック・オフセットの代わりに使用できます。

5.4 64 ビット・アドレス計算の指定

MACRO には，明示的なポインタ型宣言はありません。レジスタに 64 ビット・ポインタ値を作成する方法には，さまざまなものがあります。最も一般的な方法は，EVAX_LDQ ビルトインを使用して，メモリに格納されているアドレスをロードする方法と，MOVAxを使用して，特定のオペランドのアドレスを取得する方法です。

64 ビットのポインタ値をレジスタに格納したら，通常の命令で 64 ビット・アドレスにアクセスします。そのアドレスから読み込まれるデータの量は，使用する命令に依存します。次の例を考えます。

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

MOVL 命令は，R1 に 32 ビットのポインタが格納されている場合でも，64 ビットのポインタが格納されている場合でも，R1 からオフセット 4 の位置にあるロングワードを読み込みます。

しかし，特定のアドレッシング・モードでは，実効アドレスを計算するために，算術命令の生成が必要です。VAX との互換性のため，コンパイラはこれらをロングワード演算として計算します。たとえば，4 + <1@33>を計算すると，シフトした値が 32 ビットを超えるため，結果の値は 4 となります。クォドワード・モードが有効な場合は，上位ビットは失われません。

バージョン 7.0 よりも前の OpenVMS Alpha に添付されていたコンパイラでは，/ENABLE=QUADWORD 修飾子 (および対応する指示文.ENABLE QUADWORD と.DISABLE QUADWORD) は，定数式の評価が実行されるモードにだけ影響がありました。OpenVMS Alpha Version 7.0 および OpenVMS I64 では，アドレス計算にも影響を与えるように拡張されました。その結果，SxADDQ や ADDQ などのクォドワード命令でアドレスが計算されるようになりました。

モジュール全体でクォドワード演算を使用するには，コマンド行で /ENABLE=QUADWORD を指定します。特定のセクションだけにクォドワード演算を適用する場合は，指示文.ENABLE QUADWORD と.DISABLE QUADWORD を使用してこれらのセクションを囲みます。

/ENABLE=QUADWORD を使用しても性能が低下することはありません。

5.4.1 ロングワード演算のラップ動作への依存

既存のコードが 32 ビット演算のラップ動作に依存している可能性があるため，コンパイラは，すべてのアドレス計算でクォドワード演算を使うことはできません。つまり，上位ビットが破棄されることを承知で，実際に 32 ビットをオーバフローするアドレス演算を行っている可能性があります。クォドワード・モードで計算を行うと互換性が失われます。

/ENABLE を使用して，モジュール全体のクォドワード評価を設定する前に，既存のコードの中にロングワードのラップ動作に依存している部分がないか確認してください。この確認を行うための簡単な方法はありませんが，プログラミング・テクニックとしてはめったに使用されないためコード中にコメントが記載されていることがあります。

次の Alpha の例は，ラップ動作の問題を示します。

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

R1 に値 7FFFFFFF が格納されており， R0 に 1 が格納されているとします。MOVAL 命令は S4ADDL 命令を生成します。シフトと加算の結果 32 ビットを超えますが，格納される結果は符号拡張された下位 32 ビットです。

クォドワード演算を使用すると (S4ADDQ)，次の例に示すように真のクォドワード値となります。

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

ラップ動作の問題は，インデックス・モードのアドレッシングだけに限りません。次の例を考えます。

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

シンボル offset がコンパイル時に定数でない場合は，この命令によって値がリンケージ・セクションから読み込まれ， R1 の値に加算されます (ADDL 命令を使用)。これを ADDQ に変更すると，値が 32 ビットを超えている場合に結果が変わります。

Itanium アーキテクチャには S4ADDL 命令に相当する命令がありませんが，コンパイラは shladd 命令と sxt4 命令を生成し，この効果をシミュレートします。

5.5 符号拡張とチェック

ビルトイン EVAX_SEXTL (ロングワード符号拡張) を使用すると， 64 ビット値の下位 32 ビットを符号拡張してデスティネーションに格納することができます。このビルトインは，ソースの下位ロングワードの符号拡張を明確にして，デスティネーションに格納します。

EVAX_SEXTL は， 64 ビット値の下位 32 ビットを受け取り，上位 32 ビットに符号拡張 (値のビット 31 の内容) を設定して，結果の 64 ビットをデスティネーションに格納します。

以下の例はすべて正しい使用法です。

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

これらの例が示すように，オペランドは必ずしもレジスタである必要はありません。

マクロ$IS_32BITS を使用すると， 64 ビット値の下位 32 ビットの符号拡張をチェックすることができます。説明については付録 E を参照してください。

5.6 Alpha 命令ビルトイン

コンパイラは，多くの Alpha 命令をビルトインとしてサポートしています。これらのビルトインの多くは (コンパイラの最初のリリースから利用可能です)， 64 ビットのデータを処理するために使用できます。各ビルトインの機能と有効なオペランドについては，付録 C を参照してください。

これらビルトインの多くは OpenVMS I64 でも使用でき， 1 つ以上の Itanium 命令を使用して同じ動作がエミュレートされます。

5.7 ページ・サイズに依存する値の計算

以下に示す各ページ・マクロでは， 64 ビット仮想アドレスをサポートするためのパラメータ QUAD=NO/YES が使用できます。

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

これらのマクロは，ページ・サイズに依存した値を計算するための，標準的でアーキテクチャに依存しない方法を提供します。これらのマクロについての詳細は，付録 D を参照してください。

5.8 64 ビット・アドレス空間内でのバッファの作成と使用

制御ブロック・マクロ$RAB と$RAB_STORE は 64 ビット・アドレス空間内にデータ・バッファを作成して使用できるように拡張されています。 64 ビット版の名前は，それぞれ$RAB64 と$RAB64_STORE です。残りの RMS インタフェースは，現時点では 32 ビットに制限されています。$RAB64 および$RAB64_STORE についての詳細は， HP OpenVMS Programming Concepts Manual を参照してください。

5.9 64 KB を超える移動のコーディング

MACRO 命令 MOVC3 および MOVC5 は， 64 ビット・アドレスを適切に処理しますが，移動する長さは 64 KB までに制限されています。この制限は， MOVC3 と MOVC5 がワード・サイズの長さを受け取ることが原因です。

64 KB を超えて移動を行うには， OTS$MOVE3 と OTS$MOVE5 を使用します。 OTS$MOVE3 と OTS$MOVE5 は，ロングワード・サイズの長さを受け取ります。 (LIB$MOVC3 および LIB$MOVC5 には， MOVC3 および MOVC5 と同じ 64 KB の長さ制限があります。)

以下では， MOVC3 を OTS$MOVE3 で置き換える例を示します。

MOVC3 を使用したコード:

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

長さがロングワードの，対応する 64 ビット・コード

```
$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
```

MOVC3 では R0， R2， R4，および R5 がクリアされるため，これらの副作用が必要でないことを確認してください。

OTS$MOVE3 と OTS$MOVE5 は，他の LIBOTS ルーチンとともに OpenVMS RTL General Purpose (OTS$) Manual で説明されています。

第2部
リファレンス

A

MACRO コンパイラの修飾子

この付録では， MACRO Compiler for OpenVMS systems の起動形式と，そのそれぞれの修飾子について説明します。

MACRO/MIGRATION

MACRO Compiler for OpenVMS systems を起動して， 1 つ以上のVAX MACROアセンブリ言語ソース・ファイルをコンパイルし， OpenVMS Alpha または OpenVMS I64 のネイティブ・オブジェクト・コードを生成します。

フォーマット

MACRO/MIGRATION *filespec[+...]*

注意

OpenVMS Alpha および OpenVMS I64 では， MACRO コマンドのデフォルトは/MIGRATION です。

パラメータ

filespec[+...]
コンパイル対象のVAX MACROアセンブリ言語ソース・ファイルを指定します。複数のファイルを指定する場合は，ファイル指定をプラス記号(+)で区切ります。プラス記号で区切ったファイル指定は， 1 つの入力ファイルとして連結され，単一のオブジェクト・ファイルとリスト・ファイル(指定されている場合)が生成されます。

注意

VAX アセンブラと異なり， MACRO コンパイラでは，ソース・ファイルをコンマ(,)で区切ることによる個別のオブジェクト・ファイルの作成はサポートされていません。

ファイル指定ではワイルドカード文字は指定できません。それぞれのファイル指定に対して，デフォルトのファイル・タイプは MAR となります。

コンパイラは，出力先ディレクトリにある既存のファイル (バージョン番号が最も大きなもの) よりもバージョン番号が 1 つ大きな出力ファイルを作成します。

説明

MACRO/MIGRATION コマンドに対する修飾子は，コマンド (グローバル) 修飾子と定位置修飾子のどちらかとなります。**コマンド修飾子**は，コマンドで指定されたすべてのファイルに影響します。**定位置修飾子**は，それが修飾するファイルにだけ影響します。/LIBRARY を除くすべての MACRO/MIGRATION 修飾子がコマンド修飾子または定位置修飾子として使用できます。/LIBRARY 修飾子は，定位置修飾子としてのみ使用できます。

多くの修飾子では 1 つ以上の引数を指定できます。引数を 1 つだけ指定する場合は，括弧を省略できます。

コンパイラは，標準の MACRO 修飾子のほとんどをサポートしています。これらの修飾子のいくつかにはコンパイラ固有のオプションがあり，VAX MACROのオプションがいくつかサポートされていない修飾子もあります。コンパイラは，コンパイラ固有の修飾子もいくつかサポートしています。これらの修飾子のすべてを表 A–1 に示しています。

表 A–1 コンパイラ修飾子

標準の MACRO 修飾子	固有の修飾子
/DEBUG	/FLAG
/DIAGNOSTICS	/MACHINE
/DISABLE[1]	/OPTIMIZE
/ENABLE[1]	/PRESERVE
/LIBRARY	/RETRY_COUNT
/LIST	/SYMBOLS
/OBJECT	/TIE
/SHOW	/UNALIGNED
	/WARN

[1]コンパイラ固有のオプションが追加されており，一部のVAX MACROオプションがサポートされていません。

修飾子

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

/NODEBUG
オブジェクト・モジュールのシンボル・テーブルまたはトレースバック情報に，ローカル・シンボルを含めるかどうかを指定します。以下のオプションを 1 つ以上指定できます。

オプション	説明
ALL	オブジェクト・モジュール中のローカル・シンボルとトレースバック情報をデバッガで使用できるようにする。この修飾子は，/ENABLE=(DEBUG,TRACEBACK) に相当する。
NONE	オブジェクト・モジュール中のローカル・シンボルとトレースバック情報をデバッガで使用できないようにする。この修飾子は，/DISABLE=(DEBUG,TRACEBACK) に相当する。
SYMBOLS	オブジェクト・モジュール中のすべてのローカル・シンボルをデバッガで使用できるようにし，すべてのトレースバック情報を使用できないようにする。この修飾子は/ENABLE=SYMBOLS に相当する。
TRACEBACK	オブジェクト・モジュール中のトレースバック情報をデバッガで使用できるようにし，ローカル・シンボル情報を使用できないようにする。この修飾子は，/ENABLE=TRACEBACK に相当する。

/DEBUG のデフォルト値は ALL です。/DEBUG 修飾子は，コマンド行での順序にかかわらず，/ENABLE=(DEBUG,TRACEBACK) や /DISABLE=(DEBUG,TRACEBACK) より優先されます。

注意

/NOOPTIMIZE を指定することでデバッグが容易になります。この修飾子は，生成されるコードが，ソース行の境界を越えて移動されないようにします。

デバッグについての詳細は， OpenVMS Debugger Manual を参照してください。

/DIAGNOSTICS[=filespec]
/NODIAGNOSTICS (**デフォルト**)
アセンブラ・メッセージと診断情報が格納されたファイルを作成します。ファイル指定を省略した場合のデフォルトのファイル名はソース・プログラムと同じ名前で，デフォルトのファイル・タイプは DIA です。

ファイル指定ではワイルドカード文字は使用できません。

診断ファイルは， VAX Language-Sensitive Editor (LSE) などのレイヤード・プロダクトで使用するために予約されています。

/DISABLE=(option[,...])
/NODISABLE
MACRO の.DISABLE 指示文と.ENABLE 指示文で制御可能なコンパイラ機能を初期設定します。

以下の機能を 1 つ以上指定することができます。

オプション	説明
DEBUG	デバッガが使用するローカル・シンボル・テーブル情報を，オブジェクト・ファイルに含めない。同時に/DEBUG 修飾子も指定すると，コマンド行での順序にかかわらず，/DISABLE=(DEBUG,TRACEBACK) や/ENABLE=(DEBUG,TRACEBACK) より優先される。
FLAGGING	コンパイラのメッセージ出力を無効にする。
GLOBAL	未定義のシンボルは外部シンボルであるという想定を無効にする。
OVERFLOW	次のオペレーション・コードに対するオーバフロー・トラップ・コードの生成を無効にする: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (xはyよりも大きい。例: CVTLB), AOBxx, ACBL, SOBxx
QUADWORD	クォドワードのリテラル式とアドレス式のサポートを無効にする。
SUPPRESSION	参照されていないシンボルをシンボル・テーブルに登録しない。
TRACEBACK	トレースバック情報をデバッガに渡さない。同時に /DEBUG 修飾子も指定すると，コマンド行での順序にかかわらず，/DISABLE=(DEBUG,TRACEBACK) や /ENABLE=(DEBUG,TRACEBACK) より優先される。

デフォルトでは，コンパイラの起動時に FLAGGING， GLOBAL，および SUPPRESSION が有効になり， DEBUG， OVERFLOW， QUADWORD，および TRACEBACK が無効になります。

/NODISABLE 修飾子は，/DISABLE 修飾子を省略したのと同じ効果があります。また，コマンド行で先に指定した/DISABLE 修飾子の効果を打ち消すためにも使用できます。

注意

コマンド行で/DISABLE **を 2 回以上使用すると，最後に指定した**/DISABLE **が優先されます。最後の**/DISABLE **で指定しなかったオプションは，デフォルト値に戻ります。**

さらに，/ENABLE **と**/DISABLE **を同じコマンド行で同じオプションに対して使用すると，コマンド行での位置にかかわらず，常に**/DISABLE **が優先されます。**

対処方法: **2 つ以上のオプションを無効にしたい場合は，次のように指定します。**

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

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

MACRO の.DISABLE 指示文と.ENABLE 指示文で制御可能なコンパイラ機能を初期設定します。

以下の機能を 1 つ以上指定することができます。

オプション	説明
DEBUG	デバッガが使用するローカル・シンボル・テーブル情報を，オブジェクト・ファイルに含める。同時に/DEBUG 修飾子も指定すると，コマンド行での順序にかかわらず，/ENABLE=(DEBUG,TRACEBACK) や/DISABLE=(DEBUG,TRACEBACK) より優先される。
FLAGGING	コンパイラのメッセージ出力を有効にする。
GLOBAL	未定義のシンボルは外部シンボルであると想定する。
OVERFLOW	次のオペレーション・コードに対するオーバフロー・トラップ・コードの生成を有効にする: ADDx， ADWC， INCx， ADAWI， SUBx， SBWC， DECx， MNEGx， MULx， CVTxy (xはyよりも大きい。例: CVTLB)， AOBxx， ACBL， SOBxx
QUADWORD	クォドワードのリテラル式とアドレス式のサポートを有効にする。
SUPPRESSION	参照されていないシンボルをシンボル・テーブルに登録する。
TRACEBACK	トレースバック情報をデバッガに渡す。同時に /DEBUG 修飾子も指定すると，コマンド行での順序にかかわらず，/ENABLE=(DEBUG,TRACEBACK) や /DISABLE=(DEBUG,TRACEBACK) より優先される。

デフォルトでは，コンパイラの起動時に FLAGGING， GLOBAL， TRACEBACK，および SUPPRESSION が有効になり， DEBUG， OVERFLOW，および QUADWORD が無効になります。

/NOENABLE 修飾子は，/ENABLE 修飾子を指定しなかったのと同じ効果があります。また，コマンド行で先に指定した/ENABLE 修飾子の効果を打ち消すためにも使用できます。

注意

/ENABLE 修飾子のすべてのオプションについて，/ENABLE と/DISABLE を同じコマンド行で同じオプションに対して使用すると，コマンド行での位置にかかわらず，常に/DISABLE が優先されます。

シンボルを使用して以前無効にしたオプションを有効にしたい場合があります。たとえば，次のように，頻繁に使用される以下のオプションを DCL シンボル MAC に対応付けているとします。

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

シンボル MAC を使用して FLAGGING を有効にするには，次のコマンドを実行します。

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

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

コンパイラが報告する情報メッセージのクラスを指定します。以下のオプションがあります。

オプション	説明
ALIGNMENT	アラインされていないスタックおよびメモリの参照を報告する。
ALL	すべてのオプションを有効にする。
ARGLIST	引数リストがホーミングされていることを報告する (第 2.4.1 項を参照)。
BAD_FIELD_USAGE (I64 のみ)	BBC/BBS 命令によるビット 32 以降の使用を報告する。
CODEGEN	自己変更コードなど，実行時のコード生成を報告する (第 3.2.2 項を参照)。
COMPILER_VERSION (I64 のみ)	コンパイラのバージョンを SYS$ERROR に出力する。
DIRECTIVES	サポートされていない指示文を報告する。
HINTS	レジスタ・ヒント input/output/auto-preserved を報告する。
INDIRECT_CALLS (I64 のみ)	間接的な呼び出しで，その前に.USE_LINKAGE 指示文のない CALLS/CALLG 命令を報告する。
INDIRECT_JSB (I64 のみ)	間接的な呼び出しで，その前に.USE_LINKAGE 指示文のない JSB 命令を報告する。
INSTRUCTIONS	コンパイルは成功するものの調査が必要な，絶対アドレスを使用した命令を報告する (絶対アドレスはシステムごとに異なる可能性があるため)。
JUMPS	ルーチン間での分岐を報告する。
LINKAGE (I64 のみ)	OpenVMS リンカに渡されるリンケージ情報を報告する。
NONE	すべてのオプションを無効にする。
STACK	ユーザのスタック操作によって生成されるすべてのメッセージを報告する。

コンパイラの起動時のデフォルトは，/FLAG=(ALIGNMENT, ARGLIST, CODEGEN, DIRECTIVES, INSTRUCTIONS, JUMPS, STACK) です。

注意

クロス・コンパイラ・メッセージのサブセットを有効にするために，/NOFLAG 修飾子と/FLAG 修飾子を同時に使用しても，期待どおりに機能しません。キーワードが示すメッセージだけを有効にするのではなく，/NOFLAG /FLAG=(keyword,keyword) **のように同時に使用すると，すべてのクロス・コンパイラ・メッセージが有効になります。しかし，**/FLAG=(none,keyword) **を使用すると，** keyword **で指定したメッセージだけが有効になります。**

/NOFLAG や/FLAG=NONE を指定しても，正常なコンパイルを妨げるコーディング構造の報告は無効になりません。上位レベル・スタックの参照など，変更が必要なコードは引き続き報告されます。

/LIBRARY
/NOLIBRARY

定位置修飾子です。

/LIBRARY 修飾子に関連付けられた入力ファイルは，マクロ・ライブラリでなければなりません。デフォルトのファイル・タイプは MLB です。/NOLIBRARY 修飾子は，/LIBRARY 修飾子を指定しないのと同じ効果があります。また，コマンド行でそれよりも前に指定した/LIBRARY 修飾子の効果を打ち消します。

コンパイラは，最大 16 個のライブラリを検索できますが，その 1 つは常に STARLET.MLB です。この数は，特定のコンパイルに適用され，必ずしも特定の MACRO コマンドに適用されるわけではありません。複数のソース・ファイルをコンパイルするように MACRO コマンドを入力する場合に，ソース・ファイルを別々にコンパイルする場合は，それぞれのコンパイルで最大 16 個のマクロ・ライブラリを指定できます。一度のコンパイルで複数のマクロ・ライブラリを指定すると，指定した順序とは逆の順序でライブラリが検索されます。

ソース・プログラム中にマクロ呼び出しがあると，マクロが未定義の場合，次の検索シーケンスが開始されます。

1. まず，.LIBRARY 指示文で指定したライブラリが，指定した順序とは逆の順序で検索されます。
2. .LIBRARY 指示文で指定したライブラリにマクロ定義が見つからない場合は，MACRO コマンド行で指定したライブラリが検索されます (指定した順序と逆の順序で検索されます)。
3. コマンド行で指定したライブラリにマクロ定義が見つからなかった場合は，STARLET.MLB が検索されます。

/LIST[=filespec]
/NOLIST
出力リストを作成するかどうかを指定し，オプションで出力ファイルを指定します。リスト・ファイルのデフォルトのファイル・タイプは LIS です。ファイル指定ではワイルドカード文字は使用できません。

対話型の MACRO コマンドは，デフォルトではリスト・ファイルを生成しません。/NOLIST 修飾子を明示的に (またはデフォルトで) 指定すると，エラーは現在の出力デバイスに報告されます。

/LIST 修飾子は，バッチ・ジョブにおける MACRO コマンドではデフォルトとなります。/LIST 修飾子では，コマンド行の中の修飾子の位置によって，出力ファイル指定に適用するデフォルトを制御することができます。

/MACHINE
/NOMACHINE (**デフォルト**)
この修飾子と/LIST 修飾子を同時に指定することで，機械語コードのリスト出力が有効になります。

/OBJECT[=filespec]
/NOOBJECT
オブジェクト・モジュールを作成するかどうかを指定し，またファイル指定を定義します。デフォルトでは，コンパイラは最初の入力ファイルと同じ名前でオブジェク

ト・モジュールを作成します。オブジェクト・ファイルのデフォルトのファイル・タイプは OBJ です。ファイル指定ではワイルドカード文字は使用できません。

/OBJECT 修飾子では，コマンド行での修飾子の位置によって，出力ファイル指定に適用するデフォルトを制御することができます。

/OPTIMIZE[=(option[,...])]
/NOOPTIMIZE
最適化オプションの有効と無効を切り替えます。 VAXREGS を除くすべてのオプションは，デフォルトで有効になっています (第 4.3.1 項を参照)。

オプションは以下のとおりです。

オプション	説明
[NO]PEEPHOLE	のぞき穴式最適化
[NO]SCHEDULE	コードのスケジューリング
[NO]ADDRESSES	共通ベース・アドレスによるロード
[NO]REFERENCES	共通データ参照
[NO]VAXREGS	OpenVMS Alpha システムのみ: VAX レジスタ (R0 ～ R12) が使用されていなければ，それを一時レジスタとして使用することを許可します。
ALL	すべての最適化
NONE	最適化なし

OpenVMS Alpha システムでは，/OPTIMIZE=ALL を指定すると VAXREGS が有効になり，モジュール中のすべてのルーチンのすべてのレジスタの使用が正しく宣言されていないと，正しくないコードが生成されます。

/PRESERVE[=(option[,...])]
/NOPRESERVE (**デフォルト**)
モジュール全体のすべてのVAX MACRO命令に対して， VAX が持つ操作の不可分性と細分性に依存した特別な OpenVMS Alpha または OpenVMS I64 のコードを生成するように，コンパイラに指示します (第 2.11 節を参照)。

オプションは以下のとおりです。

オプション	説明
GRANULARITY	VAX の書き込みの細分性の規則を維持する。/PRESERVE=GRANULARITY を指定すると，コンパイラは，バイト，ワード，またはアラインされていないロングワードの書き込みを実行する VAX 命令に対して生成するコードの中で， Alpha の Load-locked 命令シーケンスと Store-conditional 命令シーケンス，または Itanium の compare-exchange (cmpxchg) 命令を使用する。

オプション	説明
ATOMICITY	VAX の変更操作の不可分性を維持する。/PRESERVE=ATOMICITY を指定すると，コンパイラは，変更オペランドを伴う VAX 命令に対して生成するコードの中で，Alpha の Load-locked 命令シーケンスおよび Store-conditional 命令シーケンス，または Itanium の compare-exchange (cmpxchg) 命令を使用する。

/PRESERVE と/PRESERVE=(GRANULARITY,ATOMICITY) は同じです。細分性と不可分性の維持がどちらも有効になり，細分性と不可分性の両方の保証が必要な VAX コーディング構造が見つかると，細分性よりも不可分性が優先されます。

VAX の細分性と不可分性の保証が必要なVAX MACROコードのセクションがある場合は，モジュール全体に対してこれらの保証を適用するようにコンパイラに指示する必要はありません。代わりに，.PRESERVE 指示文と.NOPRESERVE 指示文を使用し (付録 B を参照)，これらのセクションだけに保証を適用することができます。コンパイラは，モジュール全体に対して膨らんだコードを生成する必要がないため，これらの指示文によってコードが最適化されます。

/PRESERVE=ATOMICITY を指定すると，ユニプロセシング・システムだけでなく，マルチプロセシング・システムでも不可分性が保証されます。

/PRESERVE 修飾子があるときには，/RETRY_COUNT 修飾子を指定することで，細分性または不可分性が保証された更新を，コンパイラが生成するコードで何度リトライするかを制御できます。

警告

/PRESERVE=ATOMICITY を有効にすると，アラインされていないデータ参照はすべて回復不可能な予約オペランド・フォルトになります。第 2.11.5 項を参照してください。また，/PRESERVE=GRANULARITY を有効にすると，アラインされていると想定されたアドレスに対するアラインされていないワード参照は，回復不可能な予約オペランド・フォルトになります。

/RETRY_COUNT=count

コンパイラに対し，生成されるコード中で以下の操作を実行する回数を指定します。

- VAX のインターロックされる命令を使って，ソース内で実行される操作のリトライ回数。
- /PRESERVE 修飾子または.PRESERVE 指示文がある場合の，不可分性または細分性を保証するための更新のリトライ回数。

/RETRY_COUNT 修飾子がない場合，コンパイラは無限にこれらの操作をリトライするコードを生成します。

/SHOW[=(function[,...])]
/NOSHOW[=(function[,...])]
コンパイラ指示文.SHOW および.NOSHOW によって制御される機能を初期設定します。

以下の機能を 1 つ以上指定することができます。

オプション	説明
CONDITIONALS	.IF 指示文と.ENDC MACRO 指示文に関連付けられた満たされない条件付きコードをリスト出力する。
CALLS	マクロ呼び出しと繰り返し範囲の展開をリスト出力する。
DEFINITIONS	マクロ定義をリスト出力する。
EXPANSIONS	マクロ展開をリスト出力する。
BINARY	マクロ呼び出しの展開によって生成されるバイナリ・コードをリスト出力する。

/SYMBOLS
/NOSYMBOLS (**デフォルト**)
コマンド行でこの修飾子と/LIST 修飾子がともに指定されている場合，リスト・ファイルに対し，シンボル・テーブルと psect 一覧テーブルを生成します。

/TIE (OpenVMS Alpha **でのデフォルト**)
/NOTIE (OpenVMS I64 **でのデフォルト**)
トランスレートされたイメージに対して，適切な外部コールアウトを生成します。トランスレートされたイメージとは， DECmigrate (VEST とも呼ばれます) ファシリティによって変換されたイメージです。 TIE (Translated Image Environment) を使用すると， OpenVMS VAX システム上で動作しているかのように，トランスレートされたイメージを実行できます。トランスレートされたイメージを呼び出さない場合は，性能を向上させるために/NOTIE を使用してください。

/UNALIGNED
/NOUNALIGNED (**デフォルト**)
すべてのレジスタ・ベースのメモリ参照に対して，アラインされていないロードと格納を使用するようにコンパイラに指示します (FP ベースの参照や SP ベースの参照，またはローカルにアラインされた静的データへの参照を除く)。

デフォルトでは，コンパイラはベース・ポインタとして使用されるレジスタ中のアドレス (FP ベースのポインタや SP ベースのポインタを除く) は，ルーチンの入口でロングワードにアラインされているものと想定し， BYTE， WORD，および LONG のデータをそれに応じてロードするコードを生成します。その結果，想定が正しくないと，実行時にアラインメント・フォルトが発生し，性能に大きな影響が出ます。/UNALIGNED を指定すると，コンパイラは，ポインタがアラインされていないものと想定してコードを生成します。このコードはかなり大きくなりますが，アラインメント・フォルトを処理するよりも効率が良くなります。

注意

コンパイラはクォドワード・レジスタのアラインメントを追跡しません。クォドワードのメモリ参照 (VAX の MOVQ 命令など) では，レジスタ追跡コードによってアドレスがロングワードにアラインされていない可能性があることが分かっていないかぎり，コンパイラはベース・アドレスがクォドワードにアラインされていることを想定します。 OpenVMS Alpha および OpenVMS I64 のビルトインでのクォドワードの参照は，常にクォドワードにアラインされているものと想定されます。これらは必ず新しいコードにあるため，データは正しくアラインされているはずです。

/UNALIGNED 修飾子は，通常，データがアラインされていないことが多いが，ソース中のデータ・アラインメントを訂正することでメリットが得られるほど高い性能を必要としないモジュールにのみ適しています。

/WARN=[[option]...]
/NOWARN

情報レベルのメッセージまたは警告レベルのメッセージをすべて無効にします。どちらもデフォルトで有効になっています。以下のオプションがあります。

オプション	説明
INFO	情報レベルのメッセージをすべて有効にする。
NOINFO	情報レベルのメッセージをすべて無効にする。
WARN	情報レベルのメッセージと警告レベルのメッセージをすべて有効にする。
NOWARN	情報レベルのメッセージと警告レベルのメッセージをすべて無効にする。

B

専用の指示文

この付録では， MACRO Compiler for OpenVMS systems 専用の指示文について説明します。

B.1 VAX MACROアセンブラの指示文

MACRO Compiler for OpenVMS systems は，VAX MACRO and Instruction Set Reference Manual で説明している，標準的なVAX MACROアセンブラの指示文のほとんどをサポートしています。ただし，VAX MACROアセンブラでサポートされているいくつかの指示文は，コードがコンパイルされる場合には意味を持ちません。そのため，そのような指示文があると，コンパイラはメッセージを出力して実行を継続します。/NOFLAG=DIRECTIVES を指定することで，これらの指示文に対するメッセージ出力を無効にできます。

無効にできる指示文は以下のとおりです。

- .ENABLE および.DISABLE ABSOLUTE— 絶対アドレス・モードの適用
- .ENABLE および.DISABLE TRUNCATION— 浮動小数点切り捨ての有効化
- .LINK— リンカ・オプション・ファイルによるリンカ・オプションの指定
- .DEFAULT— 変位長の設定
- .OPDEF および.REFn— オペレーション・コードの定義
- コード psect 中のアラインメント指示文 (.ALIGN，.EVEN，および.ODD)
- .TRANSFER (第 3.7 節を参照)
- .MASK

注意

MACRO Compiler for OpenVMS systems を使用するときには，.ASCID 指示文に対する引数の長さは， 996 文字に制限されています。VAX MACROアセンブラにはこのような制限はありません。

B.2 MACRO コンパイラ専用の指示文

MACRO Compiler for OpenVMS systems には，以下に示す専用の指示文があります。

- .BRANCH_LIKELY
- .BRANCH_UNLIKELY
- .CALL_ENTRY
- .CALL_LINKAGE (OpenVMS I64 のみ)
- .DEFINE_LINKAGE (OpenVMS I64 のみ)
- .DEFINE_PAL (OpenVMS Alpha のみ)
- .DISABLE
- .ENABLE
- .EXCEPTION_ENTRY (OpenVMS Alpha のみ)
- .GLOBAL_LABEL
- .JSB_ENTRY
- .JSB32_ENTRY
- .LINKAGE_PSECT (OpenVMS Alpha のみ)
- .PRESERVE
- .SET_REGISTERS
- .SYMBOL_ALIGNMENT
- .USE_LINKAGE (OpenVMS I64 のみ)

これらの指示文に対して特定の引数を使用し，レジスタ・セットを指示することができます。レジスタ・セットを指定するには，次の例のように，複数のレジスタをコンマで区切り，山括弧で囲みます。

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

レジスタ・セット中に 1 つしかレジスタがない場合は，次の例のように，山括弧は必要ありません。

```
R1
```

.BRANCH_LIKELY

次の分岐が成立する可能性が高いことをコンパイラに指示します。

フォーマット

.BRANCH_LIKELY

この指示文にはパラメータはありません。

説明

Alpha ハードウェアでは，前方条件分岐は成立せず，後方条件分岐は成立すると予測されます。 Alpha アーキテクチャに基づき，これらの想定がコンパイラに組み込まれており，条件分岐のコード生成に影響を与えます。

.BRANCH_LIKELY が前方条件分岐の前にあると，コンパイラは条件分岐を変更して，可能性が低いパスが，可能性が高い分岐の代わりに前方分岐となるようにコードの順序を入れ替えます。

Itanium アーキテクチャには，各分岐命令に対して明示的な分岐予測機能があります。しかし，ここでもコンパイラは，前方分岐は成立せず，後方分岐は成立するという仮定に従ってコードの順序を変更します。コンパイラは，分岐予測フラグを適切に設定します。

例

1.
```
MOVL (R0),R1
.BRANCH_LIKELY
BNEQ    10$
  .
  .
  .
10$
```

コンパイラは， BNEQ 命令とラベル 10$の間にあるコードを，モジュールの最後に移動し， BNEQ 10$を，移動したコードへの BEQL に変更します。次に，BEQL 命令のすぐ後から，ラベル 10$から始まるコードを生成します。

.BRANCH_UNLIKELY

コンパイラに対して，その下にある分岐は成立する可能性が低いことを指示します。その結果，コンパイラはこの想定を取り入れたコードを生成します。

フォーマット

.BRANCH_UNLIKELY

この指示文にはパラメータはありません。

説明

分岐予測を行う際にコンパイラが使用する想定については，.BRANCH_LIKELY 指示文の説明を参照してください。

OpenVMS Alpha システムでは，.BRANCH_UNLIKELY が後方条件分岐の前にあると，コンパイラは条件分岐とコードを変更して，無条件後方分岐命令への前方分岐になるようにします。.BRANCH_UNLIKELY は，単にそのまま次に進むよりも可能性が低いというだけでなく，分岐の可能性が非常に低い場合にだけ使用します。

OpenVMS I64 システムでは，.BRANCH_UNLIKELY が後方条件分岐の前にあると，コンパイラは，生成された Itanium 命令に対して，適切な分岐予測フラグを使用します。

.BRANCH_UNLIKELY は，前方条件分岐の前に指定しても効果がありません。

例

1.
```
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
```

.BRANCH_UNLIKELY 指示文をここで使用している理由は， Alpha ハードウェアによって， 10$への後方分岐が成立する可能性が高いと予測されるためです。

プログラマは，それがまれなケースであることを知っているため，この指示文を使用してその分岐を，成立しないと予測される前方分岐に変更しています。

.CALL_ENTRY

呼び出されるルーチンのエントリ・ポイントをコンパイラに対して宣言します。このエントリ・ポイント宣言によって，ルーチンで変更され，scratchまたはoutputとして宣言されていないすべてのレジスタ (R0 と R1 を除く) に対して， 64 ビット全体の保存と復元が行われます。

フォーマット

.CALL_ENTRY *[max_args*=number*] [,home_args=TRUE | FALSE] [,quad_args=TRUE | FALSE] [,input] [,output] [,scratch] [,preserve] [,label]*

パラメータ

max_args=number
呼び出されるプロシージャが期待する引数の最大数です。コンパイラは，引数リストをホーミングする必要がある場合に，スタック・フレームの固定の一時領域に割り当てるロングワードの数としてこの引数を使用します。ホーミングが必要でない場合は，max_argsの個数は必要ありません。引数リストのホーミングが必要なプロシージャのエントリ・ポイントでmax_argsが指定されていないと，コンパイラはメッセージを出力します。

引数リストをホーミングする際に，max_argsが 14 を超える.CALL_ENTRY ルーチンに対しては，受け取った引数カウントとmax_argsのうち，小さい方が使用されます。

home_args=TRUE | FALSE
呼び出されるプロシージャの引数リストをホーミングする必要があるかどうかをコンパイラに指示します。home_args引数は，引数リストをホーミングしなくてはならない状況にあるかどうかを判断する，コンパイラのデフォルトのロジック (第 2.4.1 項を参照) を変更します。

quad_args=TRUE | FALSE
呼び出されるプロシージャの引数リストにクォドワード参照があるかどうかをコンパイラに指示します。

input=<>
ルーチンが入力値を受け取るレジスタを示すレジスタ・セットです。

このレジスタ・セットは，指定したレジスタに，ルーチンの入口で意味のある値が格納されており，コンパイラが最初のレジスタの使用を検出する前であっても，一時レジスタとして使用できないことをコンパイラに指示します。このレジスタ・セットにレジスタを指定すると，次の 2 つの場合にコンパイラの一時レジスタの使用に影響を与えます。

- 最適化オプション VAXREGS (OpenVMS Alpha のみ) を使用している場合。この最適化を有効にすると，VAX MACROコードで明示的に使用されていないすべての VAX レジスタを，コンパイラが一時レジスタとして使用できるようになります。
- Alpha または Itanium のいずれかのレジスタ (R13 以降) を，明示的に使用している場合。

上記いずれかに該当する場合は，入力として使用するレジスタをinput引数に指定しないと，コンパイラがそのレジスタを一時レジスタとして使用し，入力値が壊れるおそれがあります。

このレジスタ・セットは，コンパイラのデフォルトのレジスタ保護動作には影響を与えません。 VAXREGS 最適化スイッチを使用しない場合や， Alpha レジスタを使用しない場合は，入力マスクはルーチンの文書化のためにのみ使用されます。

output=<>
ルーチンがその呼び出し元に返す値を代入するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれるレジスタに対しては，ルーチンによって変更される場合でも，コンパイラによる保存と復元が行われません。

このレジスタ・セットは，指定されたレジスタにはルーチンの出口で意味のある値が格納されており，コンパイラが最後のレジスタの使用を検出した後でも，一時レジスタとして使用できないことをコンパイラに対して通知します。このレジスタ・セットにレジスタを指定すると，以下の 2 つの場合にコンパイラの一時レジスタの使用に影響を与えます。

- 最適化スイッチ VAXREGS (OpenVMS Alpha のみ) を使用している場合。この最適化を有効にすると，VAX MACROコードで明示的に使用されていないすべての VAX レジスタを，コンパイラが一時レジスタとして使用できるようになります。
- Alpha または Itanium のいずれかのレジスタ (R13 以降) を，明示的に使用している場合。

上記いずれかに該当する場合は，出力として使用するレジスタをoutput引数で指定しないと，コンパイラがそのレジスタを一時レジスタとして使用し，出力値が壊れるおそれがあります。

scratch=<>
ルーチン内で使用されているものの，ルーチンの入口と出口で保存と復元を行うべきでないレジスタを示すレジスタ・セットです。ルーチンの呼び出し元は，出力値を受け取ることを期待しておらず，レジスタが保護されることも期待していません。この

レジスタ・セットに含まれるレジスタに対しては，ルーチンによって変更される場合でも，コンパイラによる保存と復元が行われません。

これは，コンパイラの一時レジスタの使用にも関係します。 OpenVMS Alpha システムでは，レジスタ R13 以降がルーチンのソース・コードで使用されていなければ，コンパイラはこれらのレジスタを一時レジスタとして使用します。 OpenVMS Alpha システムでは， R13 ～ R15 が変更される場合は保護する必要があるため，コンパイラはこれらのレジスタを使用する場合には保護します。

しかし，これらのレジスタがscratchレジスタ・セット宣言に指定されている場合は，コンパイラは，一時レジスタとしてそれを使用する場合に保護しません。その結果，これらのレジスタは，ルーチンのソースで使用されていなくても，scratchセットに指定されていれば，ルーチンの出口では壊れている可能性があります。 VAXREGS (OpenVMS Alpha のみ) による最適化を使用した場合は，これはレジスタ R2 ～ R12 にも適用されます。

OpenVMS I64 システムでは，コンパイラはこれらのレジスタを一時レジスタとして使用しません。

preserve=<>
ルーチン呼び出しの前後で保護する必要があるレジスタを指示するレジスタ・セットです。これには，変更され， 64 ビットの内容全体を保存および復元する必要があるレジスタだけを含める必要があります。

このレジスタ・セットを指定すると，コンパイラによってレジスタが自動的に保護されているかどうかにかかわらずレジスタは保護されます。 R0 と R1 はスクラッチ・レジスタであるため，このレジスタ・セットで指定しないかぎり，標準の定義を呼び出しただけでは，コンパイラはこれらのレジスタの保存と復元を実行しません。レジスタ R16 以降は指定できません。

このレジスタ・セットは，outputレジスタ・セットとscratchレジスタ・セットより優先されます。preserveレジスタ・セットと，outputレジスタ・セットまたはscratchレジスタ・セットの両方でレジスタを指定すると，コンパイラは次の警告を報告します。

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

label=name
VAX MACROの.ENTRY 指示文と同じように，オプションでラベルを指定することができます。これは，モジュールを OpenVMS VAX と OpenVMS Alpha や OpenVMS I64 で共通にする場合や， OpenVMS VAX バージョンで.MASK 指示文のあるエントリを参照する必要がある場合， OpenVMS Alpha または OpenVMS I64 のバージョンで特別な.CALL_ENTRY パラメータを使用する必要がある場合に使用できます。labelパラメータが指定され，シンボル VAX が定義されていると，.ENTRY 指示文が使用されます (第 1.7.3 項を参照)。シンボル VAX が定義されていない場合は，ラベルを作成して通常の.CALL_ENTRY の動作をします。labelは最初のパラメータ

ではありません。そのため，単純に.ENTRY を.CALL_ENTRY に置き換えることはできません。labelパラメータ宣言を使用する必要があります。

.CALL_LINKAGE (OpenVMS I64 のみ)

名前付きリンケージまたは匿名リンケージをルーチン名に関連付けます。コンパイラが，ルーチン名をターゲットとする CALLS，CALLG，JSB，BSBB，BSBW のいずれかの命令を見つけると，関連付けられているリンケージを使用して，呼び出しの前後で保存と復元が必要なレジスタを決定します。

フォーマット

.CALL_LINKAGE *routine_name [,linkage_name] [,input] [,output] [,scratch] [,preserve]*

パラメータ

routine_name
リンケージに関連付けるルーチンの名前です。

linkage_name =
.DEFINE_LINKAGE 指示文で定義したリンケージの名前です。linkage_nameを指定する場合，パラメータinput，output，scratch，preserveは指定できません。

input=<>
routine_nameが入力値を受け取るレジスタを示すレジスタ・セットです。このパラメータは文書化のためにのみあります。

inputレジスタ・セットを指定する場合，linkage_nameは指定できません。

output=<>
routine_nameがルーチンの呼び出し元に返す値を代入するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれているレジスタは，呼び出しの前後で保存と復元が行われません。

outputレジスタ・セットを指定する場合は，linkage_nameは指定できません。

scratch=<>
ルーチン内で使用するレジスタを示すレジスタ・セットです。

scratchレジスタ・セットを指定する場合は，linkage_nameは指定できません。

preserve=<>
routine_nameが保護するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれているレジスタは，ルーチンの呼び出しの前後で保存と復元が行われません。保存と復元は，呼び出されるルーチンが行うためです。

preserveレジスタ・セットを指定する場合は，linkage_nameは指定できません。

.DEFINE_LINKAGE (OpenVMS I64 のみ)

以降の.CALL_LINKAGE 指示文または.USE_LINKAGE 指示文で使用可能な名前付きリンケージを定義します。

フォーマット

.DEFINE_LINKAGE　*linkage_name [,input] [,output] [,scratch] [,preserve]*

パラメータ

linkage_name
定義するリンケージの名前です。

input=<>
このリンケージを持つルーチンが入力値を受け取るレジスタを示すレジスタ・セットです。このパラメータは文書化のためにのみあります。

output=<>
このリンケージを持つルーチンが，その呼び出し元に返す値を代入するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれているレジスタは，呼び出しの前後で保存と復元が行われません。

scratch=<>
このリンケージを持つルーチンで使用されるレジスタを示すレジスタ・セットです。このパラメータは文書化のためにのみあります。

preserve=<>
このリンケージを持つルーチンが内容を保護するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれているレジスタは，ルーチンの呼び出しの前後で保存と復元が行われません。この作業は，呼び出されるルーチンで行うためです。

.DEFINE_PAL (OpenVMS Alpha のみ)

後で MACRO から呼び出すことができるように，任意の PALcode 機能を定義します。

フォーマット

.DEFINE_PAL *name, pal_opcode, [,operand_descriptor_list]*

パラメータ

name
PALcode 機能の名前です。コンパイラは，指定された名前に接頭辞 EVAX_を付加します (EVAX_MTPR_USP など)。

pal_opcode
PALcode 機能のオペレーション・コード値です。 PALcode のオペレーション・コードの一覧は Alpha Architecture Reference Manual に記載されています。

機能コードを 16 進形式 (^X) で指定する場合は，必ず山括弧で囲んでください。機能コードを 10 進形式で指定する場合は，山括弧は必要ありません。

operand_descriptor_list
オペランドの数とそれぞれの型を指定するオペランド記述子のリストです。リストには最大 6 個のオペランド記述子を指定できます。コンパイラがレジスタとスタックの使用を正しく追跡できるように，オペランドは正しく指定してください。表 B–1 に，オペランド記述子の一覧を示します。

表 B–1 オペランド記述子

アクセス・タイプ	データ型			
	バイト	ワード	ロングワード	オクタワード
アドレス	AB	AW	AL	AQ
読み込み専用	RB	RW	RL	RQ
変更	MB	MW	ML	MQ
書き込み専用	WB	WW	WL	WQ

説明

デフォルトで，コンパイラは多数の Alpha PALcode 命令をビルトインとして定義しています。ビルトインの一覧は付録 C を参照してください。コンパイラ・ビルトインが提供されていない Alpha PALcode 命令を使用する必要がある場合は，.DEFINE_PAL 指示文を使用して独自にビルトインを定義する必要があります。

例

1. `.DEFINE_PAL MTPR_USP, <^X23>, RQ`

注意

これは例です — コンパイラは，MTPR 命令を直接 PAL 呼び出しにコンパイルします。

.DISABLE

ある範囲のソース・コードに対してコンパイラの機能を無効にします。

フォーマット

.DISABLE *argument-list*

パラメータ

argument-list
次の表に示すシンボリック引数を 1 つ以上使用できます。

オプション	説明
DEBUG	デバッガが使用するローカル・シンボル・テーブル情報をオブジェクト・ファイルに含めない。
FLAGGING	コンパイラのメッセージ出力を無効にする。
GLOBAL	未定義のシンボルは外部シンボルであるという想定を無効にする。

オプション	説明
OVERFLOW	次のオペレーション・コードに対するオーバフロー・トラップ・コードの生成を無効にする: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (xはyよりも大きい。例: CVTLB), AOBxx, ACBL, SOBxx
QUADWORD	クォドワードのリテラル式とアドレス式のサポートを無効にする。
SUPPRESSION	参照されていないシンボルをシンボル・テーブルに登録しない。
TRACEBACK	トレースバック情報をデバッガに渡さない。

.ENABLE

ある範囲のソース・コードに対してコンパイラ機能を有効にします。

フォーマット

.ENABLE *argument-list*

パラメータ

argument-list

次の表に示すシンボリック引数を 1 つ以上使用できます。

オプション	説明
DEBUG[1]	デバッガが使用するローカル・シンボル・テーブル情報をオブジェクト・ファイルに含める。
FLAGGING	コンパイラのメッセージ出力を有効にする。
GLOBAL	未定義のシンボルは外部シンボルであると想定する。
OVERFLOW	次のオペレーション・コードに対するオーバフロー・トラップ・コードの生成を有効にする: ADDx, ADWC, INCx, ADAWI, SUBx, SBWC, DECx, MNEGx, MULx, CVTxy (xはyよりも大きい。例: CVTLB), AOBxx, ACBL, SOBxx
QUADWORD	クォドワードのリテラル式とアドレス式をサポートする。
SUPPRESSION	参照されていないシンボルをシンボル・テーブルに登録する。
TRACEBACK[2]	トレースバック情報をデバッガに渡す。

[1]有効にするためには，/DEBUG または/ENABLE=DEBUG を指定してコンパイルする必要があります。

[2]有効にするためには，/DEBUG または/ENABLE=TRACEBACK を指定してコンパイルする必要があります。

.EXCEPTION_ENTRY (OpenVMS Alpha のみ)

例外サービス・ルーチンのエントリ・ポイントをコンパイラに宣言します。

フォーマット

.EXCEPTION_ENTRY *[preserve] [,stack_base]*

パラメータ

preserve=<>
ルーチン呼び出しの前後で内容の保存と復元を行うレジスタを示すレジスタ・セットです。デフォルトでは，コンパイラはルーチンが変更するすべてのレジスタの 64 ビット全体の内容を，ルーチンの入口で保存し，ルーチンの出口で復元します。

.EXCEPTION_ENTRY ルーチンの場合，例外のディスパッチにより R2 ～ R7 が PC および PSL とともにスタックに保存され，ルーチン自身が実行する REI 命令によってこれらのレジスタの値が復元されます。その他のレジスタが使用されている場合は，コンパイラが生成するコードによって保存され，ルーチンが CALL 命令または JSB 命令を実行する場合は，その他すべてのレジスタが保存されます。

stack_base
ルーチンの入口でスタック・ポインタ (SP) の値を移動するレジスタです。例外エントリ・ポイントで，例外ディスパッチ処理により，レジスタ R2 ～ R7， PC， PSL がスタックにプッシュされます。 VAX のレジスタ PSL に対応する Alpha のレジスタは，プロセッサ状態 (PS) レジスタです。stack_baseで指定したレジスタに返される値は，例外サービス・ルーチンがこれらのレジスタの値を見つけるのに役立ちます。

SYS$LIBRARY:LIB.MLB のマクロ$INTSTKDEF を使用して， R2 ～ R7， PC， PSL が格納されるスタック上の領域に対するシンボルを定義することができます。シンボルは以下のとおりです。

- INTSTK$Q_R2
- INTSTK$Q_R3
- INTSTK$Q_R4
- INTSTK$Q_R5
- INTSTK$Q_R6
- INTSTK$Q_R7
- INTSTK$Q_PC

- INTSTK$Q_PS

これらのシンボルを，例外ルーチンの中でstack_base 値に対するオフセットとして使用します。stack_base 値とともに適切なシンボル・オフセットを使用することで，例外ルーチンはこれらのレジスタの保存された内容にアクセスできます。たとえば，例外ルーチンは PSL を調べて，例外が発生したときに有効だったアクセス・モードを知ることができます。

説明

.EXCEPTION_ENTRY 指示文は，例外サービス・ルーチンのエントリ・ポイントを示します。ルーチンの入口で， R3 にはプロシージャ記述子のアドレスが格納されている必要があります。ルーチンは REI 命令で終了する必要があります。

.EXCEPTION_ENTRY 指示文を使用して，以下の割り込みサービス・ルーチンをすべて宣言する必要があります。

- インターバル・クロック
- プロセッサ間割り込み
- システムまたはプロセッサの訂正可能なエラー
- 電源障害
- システムとプロセッサのマシン・アボート
- ソフトウェア割り込み

.GLOBAL_LABEL

ルーチンに対するエントリ・ポイントではないグローバル・ラベルを，ルーチン内で宣言します。

フォーマット

Label: .GLOBAL_LABEL

この指示文にはパラメータはありません。

説明

.GLOBAL_LABEL 指示文は，ルーチン内に，ルーチンのエントリ・ポイントではないグローバル・ラベルを定義します。.GLOBAL_LABEL で宣言しないかぎり，コード中のグローバル・ラベル (“::”で指定) は，エントリ・ポイント・ラベルであると見なされ，宣言が必要となります。宣言されていないと，エラーとして出力されます。

グローバル・ラベルのアドレスは保存することができます (たとえば， PUSHAL 命令を使用します)。グローバル・ラベルまたはエントリ・ポイントとして宣言されていないラベルを保存しようとすると，コンパイラはエラーを出力します。

.GLOBAL_LABEL 指示文を使用することで保存されたコード・アドレスが CALL 命令や JSB 命令のターゲットではないことを知らせることになります。グローバル・ラベルは，ルーチンの境界内に現れる必要があります。

.GLOBAL_LABEL 指示文で宣言したラベルは，$UNWIND (Unwind Call Stack) システム・サービスの呼び出しでnewpc引数として使用できます。このシステム・サービスでは，ラベルのアドレスを保存することができるためです。

しかし，コンパイラには，スタック・ポインタをこのようなラベルの位置に自動的に調整して，スタック上で渡された引数を削除したり，スタックのアラインメントを補償するための仕組みはありません。呼び出しスタックが，呼び出し元ルーチンの代替 PC までアンワインドされても，スタックには引数とアラインメント・バイトが格納されたままとなっており，呼び出し元の，元のスタックの深さに戻すこの調整 (VAX では自動的に行われます) を期待するスタック・ベースの参照は，正しくなくなります。

$UNWIND の代替 PC ターゲットとして使用する目的で，この指示文で宣言したラベルを含むコードは，正しく動作するように，特にスタック・ポインタ・ベースの参照について慎重に調べる必要があります。

.JSB_ENTRY

JSB ルーチンのエントリ・ポイントをコンパイラに宣言します。このエントリ・ポイント宣言によって，ルーチンで変更され，scratchまたはoutputとして宣言されていないすべてのレジスタ (R0 と R1 を除く) に対して， 64 ビット全体の保存と復元が行われます。.JSB32_ENTRY も参照してください。

フォーマット

.JSB_ENTRY *[input] [,output] [,scratch] [,preserve]*

パラメータ

input=<>
ルーチンが入力値を受け取るレジスタを示すレジスタ・セットです。

このレジスタ・セットは，指定したレジスタに，ルーチンの入口で意味のある値が格納されており，コンパイラが最初のレジスタの使用を検出する前であっても，一時レジスタとして使用できないことをコンパイラに指示します。このレジスタ・セットにレジスタを指定すると，次の 2 つの場合にコンパイラの一時レジスタの使用に影響を与えます。

- 最適化オプション VAXREGS (OpenVMS Alpha のみ) を使用している場合。この最適化を有効にすると，VAX MACROコードで明示的に使用されていないすべての VAX レジスタを，コンパイラが一時レジスタとして使用できるようになります。
- Alpha または Itanium のいずれかのレジスタ (R13 以降) を，明示的に使用している場合。

上記いずれかに該当する場合は，入力として使用するレジスタをinput引数に指定しないと，コンパイラがそのレジスタを一時レジスタとして使用し，入力値が壊れるおそれがあります。

このレジスタ・セットは，コンパイラのデフォルトのレジスタ保護動作には影響を与えません。 VAXREGS 最適化スイッチを使用しない場合や， Alpha レジスタを使用しない場合は，入力マスクはルーチンの文書化のためにのみ使用されます。

output=<>
ルーチンがその呼び出し元に返す値を代入するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれるレジスタに対しては，ルーチンによって変更される場合でも，コンパイラによる保存と復元が行われません。

このレジスタ・セットは，指定されたレジスタにはルーチンの出口で意味のある値が格納されており，コンパイラが最後のレジスタの使用を検出した後でも，一時レジスタとして使用できないことをコンパイラに対して通知します。このレジスタ・セットにレジスタを指定すると，以下の 2 つの場合にコンパイラの一時レジスタの使用に影響を与えます。

- 最適化オプション VAXREGS (OpenVMS Alpha のみ) を使用している場合。この最適化を有効にすると，VAX MACROコードで明示的に使用されていないすべての VAX レジスタを，コンパイラが一時レジスタとして使用できるようになります。
- Alpha または Itanium のいずれかのレジスタ (R13 以降) を，明示的に使用している場合。

上記いずれかに該当する場合は，出力として使用するレジスタをoutput引数で指定しないと，コンパイラがそのレジスタを一時レジスタとして使用し，出力値が壊れるおそれがあります。

scratch=<>
ルーチン内で使用されているものの，ルーチンの入口と出口で保存と復元を行うべきでないレジスタを示すレジスタ・セットです。ルーチンの呼び出し元は，出力値を受け取ることを期待しておらず，レジスタが保護されることも期待していません。このレジスタ・セットに含まれるレジスタに対しては，ルーチンによって変更される場合でも，コンパイラによる保存と復元が行われません。

OpenVMS Alpha システムでは，レジスタ R13 以降がルーチンのソース・コードで使用されていなければ，コンパイラはこれらのレジスタを一時レジスタとして使用します。 OpenVMS Alpha システムでは， R13 ～ R15 が変更される場合は保護する必要があるため，コンパイラはこれらのレジスタを使用する場合には保護します。

しかし，これらのレジスタがscratchレジスタ・セット宣言に指定されている場合は，コンパイラは，一時レジスタとしてそれを使用する場合に保護しません。その結果，これらのレジスタは，ルーチンのソースで使用されていなくても，scratchセットに指定されていれば，ルーチンの出口では壊れている可能性があります。 VAXREGS (OpenVMS Alpha のみ) による最適化を使用した場合は，これはレジスタ R2 ～ R12 にも適用されます。

OpenVMS I64 システムでは，コンパイラはこれらのレジスタを一時レジスタとして使用しません。

preserve=<>
ルーチン呼び出しの前後で保護する必要があるレジスタを指示するレジスタ・セットです。これには，変更され， 64 ビットの内容全体を保存および復元する必要があるレジスタだけを含める必要があります。

このレジスタ・セットを指定すると，コンパイラによってレジスタが自動的に保護されているかどうかにかかわらずレジスタは保護されます。 R0 と R1 はスクラッチ・レジスタであるため，このレジスタ・セットで指定しないかぎり，標準の定義を呼び出しただけでは，コンパイラはこれらのレジスタの保存と復元を実行しません。

このレジスタ・セットは，outputレジスタ・セットとscratchレジスタ・セットより優先されます。preserveレジスタ・セットと，outputレジスタ・セットまたはscratchレジスタ・セットの両方にレジスタを指定すると，コンパイラは次の警告を報告します。

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

注意

OpenVMS Alpha では，.JSB_ENTRY 指示文で宣言したプロシージャに対しては， MACRO コンパイラがヌル・フレーム・プロシージャ記述子を自動的に生成します。

ヌル・フレーム・プロシージャでは新しいコンテキストが設定されないため， SHOW CALLS **コマンドや** SHOW STACK **に対する応答として，このようなプロシージャに関して，完全に正しいデバッガ情報が表示される保証がないという副作用があります。たとえば，呼び出されたヌル・プロシージャ** (JSB **の実行対象) 内の行番号として，** JSB **を発行した呼び出し元プロシージャの行番号が報告される可能性があります。**

.JSB32_ENTRY

JSB ルーチンのエントリ・ポイントをコンパイラに宣言します。この指示文は，PRESERVE パラメータを指定しないかぎり， VAX レジスタの値 (R2 ～ R12) を保護しません。ルーチン自体が，レジスタをスタックにプッシュすることで，その内容を保存および復元してもかまいませんが，この方法ではレジスタの上位 32 ビットは保護されません。.JSB_ENTRY も参照してください。

注意

.JSB32_ENTRY 指示文が使用できることが分かっていると，大幅に時間が節約できる場合があります。レジスタの上位 32 ビットが使用されている状況で.JSB32_ENTRY を使用すると，非常に分かりにくく追跡が難しいバグの原因となります。問題が発生しているルーチンの数段階上の呼び出しで 64 ビットの値が破壊されるためです。

.JSB32_ENTRY は， AST ルーチン，条件ハンドラなど，非同期に実行されるコードでは使用しないでください。

フォーマット

.JSB32_ENTRY *[input] [,output] [,scratch] [,preserve]*

パラメータ

input=<>
ルーチンが入力値を受け取るレジスタを示すレジスタ・セットです。

.JSB32_ENTRY 指示文では，このレジスタ・セットはコードの文書化のためにのみ使用されます。

output=<>
ルーチンがその呼び出し元に返す値を代入するレジスタを示すレジスタ・セットです。

.JSB32_ENTRY 指示文では，このレジスタ・セットはコードの文書化のためにのみ使用されます。

scratch=<>
ルーチン内で使用されているものの，ルーチンの入口と出口で保存と復元を行うべきでないレジスタを示すレジスタ・セットです。ルーチンの呼び出し元は，出力値を受け取ることを期待しておらず，レジスタが保護されることも期待していません。

scratch引数は，コンパイラの一時レジスタの使用にも関係します。 OpenVMS Alpha システムでは，レジスタ R13 以降がルーチンのソース・コードで使用されていなければ，コンパイラはこれらのレジスタを一時レジスタとして使用します。 Alpha システムでは， R13 ～ R15 が変更される場合は保護する必要があるため，コンパイラはこれらのレジスタを使用する場合に保護します。

しかし，これらのレジスタがscratchレジスタ・セット宣言に指定されている場合は，コンパイラは，一時レジスタとしてそれを使用する場合に保護しません。その結果，これらのレジスタは，ルーチンのソースで使用されていなくても，scratchセットに指定されていれば，ルーチンの出口では壊れている可能性があります。

OpenVMS I64 システムでは，コンパイラはこれらのレジスタを一時レジスタとして使用しません。

R2 ～ R12 はデフォルトでは保護されないため，これらのレジスタをscratchに含めるのは，文書化だけが目的です。

preserve=<>
ルーチン呼び出しの前後で保護する必要があるレジスタを指示するレジスタ・セットです。これには，変更され， 64 ビットの内容全体を保存および復元する必要があるレジスタだけを含める必要があります。

このレジスタ・セットを指定すると，コンパイラによってレジスタが保護されます。デフォルトでは，.JSB32_ENTRY 指示文ではレジスタは保護されません。

このレジスタ・セットは，outputレジスタ・セットとscratchレジスタ・セットより優先されます。preserveレジスタ・セットと，outputレジスタ・セットまたはscratchレジスタ・セットの両方にレジスタを指定すると，コンパイラは次の警告を報告します。

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

説明

.JSB32_ENTRY 指示文は， JSB エントリ・ポイントを宣言するもう 1 つの方法です。これは，単独のアプリケーションや自己完結型のサブシステムなど，明確に定義された，境界のあるアプリケーション環境で動作するVAX MACROルーチンの宣言を

円滑にすることを目的としています。.JSB32_ENTRY 指示文で宣言されたルーチンに対しては，コンパイラは VAX レジスタ (R2 ～ R12) の保存と復元を自動的に行わないため，現在の 32 ビット演算はそのままとなります。.JSB32_ENTRY 指示文を使用して JSB エントリ・ポイントを宣言する場合は，保護が必要なレジスタの宣言と保存は自身で行う必要があります。

サブシステムの外部から参照可能なエントリ・ポイントが 64 ビット環境から呼び出される可能性がある場合は，それらのエントリ・ポイントを.JSB32_ENTRY としては宣言しないでください。代わりに，必要に応じて.JSB_ENTRY (または.CALL_ENTRY) を使用し，レジスタ値の 64 ビット全体が保存されるようにしてください。

.LINKAGE_PSECT (OpenVMS Alpha のみ)

リンケージ・セクションの psect の名前を変更することができます。

フォーマット

.LINKAGE_PSECT　*program-section-name*

パラメータ

program_section_name
プログラム・セクションの名前です。名前は最大 31 文字で，英数字と，下線 (_)，ドル記号 ($)，ピリオド (.) が使用できます。

説明

.LINKAGE_PSECT 指示文を使用すると，ルーチン内の他の psect を参照することで，ルーチンのリンケージ・セクションを検索することができます。これにより，メモリ内のコードのロック (第 3.10 節を参照) やコード位置の指定などの操作が容易になります。コード位置を指定する例としては，リンカ・オプションを使用して，リンカによって作成されるイメージ中に明示的に psect を配置することが挙げられます。これによって隣接する psect を使って境界を見つけることができるようになります。

.LINKAGE_PSECT 指示文を単一のソース・モジュール内で複数回使用し，さまざまなルーチンに異なるリンケージ・セクションを設定することができます。ただし，ルーチンのリンケージ・セクションはルーチン全体で同じです。ルーチンのリンケージ・セクションの名前は，ルーチンのエントリ・ポイント指示文より前の最後の.LINKAGE_PSECT指示文で指定した名前となります。

コード・パスを共用する複数のルーチンに対して異なるリンケージ・セクションが指定されていると，コンパイラは回復不可能なエラーを報告します。

.LINKAGE_PSECT 指示文は，デフォルトのリンケージ psect $LINKAGE と同じ psect 属性を設定します。リンケージ psect に NOEXE NOWRT が設定されていないかぎり，属性は通常の psect のデフォルト属性と同じになります。

リンケージ・セクションの psect 属性を変更するには，.LINKAGE_PSECTで psect を宣言した後で.PSECT 指示文を使用します。

例

1.
```
        .LINKAGE_PSECT LINK_001
        .PSECT LINK_000
LS_START:
        .PSECT LINK_002
LS_END:
```
このコードを使用すると，プログラムで LS_START と LS_END を使用して計算することで，ルーチンのリンケージ・セクション (LINK_001) の位置とサイズを判断することができます。

.PRESERVE

モジュール全体で，すべてのVAX MACRO命令に対して， VAX が持つ操作の不可分性と細分性の保証に依存した特別な OpenVMS Alpha または OpenVMS I64 のコードを生成することをコンパイラに指示します。

フォーマット

.[NO]PRESERVE　*argument-list*

パラメータ

argument-list
次の表に示すシンボリック引数を 1 つ以上指定します。

オプション	説明
オプション	**説明**
GRANULARITY	VAX の書き込みの細分性の規則を維持する。.PRESERVE=GRANULARITY を指定すると，コンパイラは，バイト，ワード，またはアラインされていないロングワードの書き込みを実行する VAX 命令に対して生成するコードの中で，Alpha の Load-locked 命令と Store-conditional 命令のシーケンス，または Itanium の compare-exchange (cmpxchg) 命令を使用する。
ATOMICITY	VAX の変更操作の不可分性を維持する。.PRESERVE=ATOMICITY を指定すると，コンパイラは，変更オペランドを伴う VAX 命令に対して生成するコードの中で，Alpha の Load-locked 命令と Store-conditional 命令のシーケンス，または Itanium の compare-exchange (cmpxchg) 命令を使用する。

説明

.PRESERVE 指示文と.NOPRESERVE 指示文を使用すると，コンパイラは，ソース・モジュールの一部のVAX MACRO命令に対し，VAX の操作の不可分性または細分性 (第 2.11 節を参照) の保証に依存した特別な Alpha アセンブリ・コードを生成します。

.PRESERVE または.NOPRESERVE を，GRANULARITYやATOMICITYを指定せずに使用すると，両方のオプションに影響を与えます。細分性と不可分性の両方の維持を有効にし，細分性と不可分性の両方の保証が必要な VAX コーディング構造が見つかると，コンパイラは細分性よりも不可分性を優先します。

また，コンパイラ修飾子/PRESERVE と/NOPRESERVE を使用して，MACRO ソース・モジュール全体で生成されるコードの不可分性と細分性に影響を与えることもできます。ただし，必要のない箇所で余分なコードによるオーバヘッドが発生し，プログラムが大幅に遅くなる可能性があるため，この方法はお勧めしません。

.PRESERVE ATOMICITY を指定すると，ユニプロセシング・システムだけでなく，マルチプロセシング・システムでも不可分性が保証されます。

.PRESERVE 指示文があると，コマンド行で/RETRY_COUNT 修飾子を指定することで，コンパイラが生成するコードで細分性または不可分性が保証された更新を何度リトライするかを制御できます。

警告

.PRESERVE ATOMICITY を有効にすると，アラインされていないデータ参照はすべて回復不可能な予約オペランド・フォルトになります。第 2.11.5 項を参照してください。

また，.PRESERVE GRANULARITY を有効にすると，アラインされていると想定されるアドレスに対するアラインされていないワード参照は，回復不可能な予約オペランド・フォルトになります。

例

1. `INCW 1(R0)`

   この命令を.PRESERVE GRANULARITY 付きでコンパイルすると，割り込みが発生した場合は新しいワード値の挿入がリトライされます。しかし，.PRESERVE ATOMICITY 付きでコンパイルしたときには，割り込みが発生した場合は初期値をフェッチし直してインクリメントします。両方のオプションを指定すると，後者になります。

.SET_REGISTERS

この指示文を使用すると，コンパイラのアラインメントの想定を変更するとともに，レジスタの暗黙的な読み書きを宣言することができます。

フォーマット

.SET_REGISTERS *argument-list*

パラメータ

argument-list
次の表に示す引数を 1 つ以上指定します。それぞれの引数に対し， 1 つ以上のレジスタを指定できます。

オプション	説明
aligned=<>	1 つ以上のレジスタをロングワード境界にアラインされているものとして宣言する。
unaligned=<>	1 つ以上のレジスタをアラインされていないものとして宣言する。この宣言は明示的であるため，このアラインされていない条件によって実行時にフォルトが発生することはない。
read=<>	ここで宣言しないとコンパイラによって入力レジスタとして検出されない 1 つ以上のレジスタを読み込みとして宣言する。
written=<>	ここで宣言しないとコンパイラによって出力レジスタとして検出されない 1 つ以上のレジスタを書き込みとして宣言する。

説明

この指示文に対してaligned修飾子とunaligned修飾子を指定することで，コンパイラのアラインメントの想定を変更することができます。この目的でこの指示文を使用すると，特定のケースでより効率の良いコードが生成されます (第 4.1 節を参照)。

この指示文に対してread修飾子とwritten修飾子を指定することで，レジスタの暗黙的な読み書きを宣言することができます。これは，通常，呼び出されるルーチンでレジスタを使用していることを宣言し，プログラムを文書化するのに有効です。

.SET_REGISTERS 指示文は，レジスタの値を変更しないかぎり，ルーチンが終了するまで有効な (正しいアラインメント処理を保証する) ままになります。ただし，フロー・パスが.SET_REGISTERS 指示文に続くコードに合流するような条件下では，このようにならない場合があります。

次の例はこの例外を示します。 R2 はalignedとして宣言されており，レジスタに対する次の書き込みアクセスの前にあるラベル 10$で，フロー・パスがコードに合流します。 R2 は，他のパスではアラインされていないため，ラベルの後ではunalignedとして扱われます。

```
        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
```

コマンド行修飾子とオプション/OPTIMIZE=VAXREGS (OpenVMS Alpha のみ) を指定する場合は，.SET_REGISTERS 指示文とそのread修飾子およびwritten修飾子は，レジスタ R2 ～ R12 でデータを受け渡しするすべてのルーチン呼び出しで必要です。これは，/OPTIMIZE=VAXREGS を指定すると，使用されていない VAX レジスタを一時レジスタとして使用できるようになるためです。

例

1.

```
DIVL R0,R1

.SET_REGISTERS ALIGNED=R1
MOVL     8(R1), R2          ; Compiler will use aligned load.
```

この例では，通常，コンパイラは除算の後で R1 がアラインされていないと見なします。 R1 をベース・レジスタとして使用したすべてのメモリ参照 (再度変更されるまで) では，アラインされていないロードと格納が使用されます。実際の値が常にアラインされていることが分かっている場合は，上記のように.SET_REGISTERS 指示文を追加することで性能が向上します。

2.
```
MOV1     4(R0), R1          ;Stored memory addresses assumed
.SET_REGISTERS UNALIGNED=R1 ;aligned so explicitly set it un-
MOVL     4(R1), R2          ;aligned to avoid run-time fault.
```

この例では， MOVL の後で R1 はロングワードにアラインされていると見なされます。実際にはアラインされていない場合は，以降のメモリ参照で実行時にアラインメント・フォルトが発生します。これを防ぐには，上記のように.SET_REGISTERS 指示文を使用します。

3.
```
.SET_REGISTERS READ=<R3,R4>, WRITTEN=R5
JSB      DO_SOMETHING_USEFUL
```

この例で，読み書き属性が使用されているのは，コンパイラが検出できないレジスタの使用を明示的に宣言するためです。 R3 と R4 は JSB ターゲット・ルーチンに対する入力レジスタで， R5 は出力レジスタです。これは，この JSB を含むルーチン自身でこれらのレジスタを使用していない場合や，.SET_REGISTERS 指示文と JSB がマクロに埋め込まれている場合に特に有効です。/FLAG=HINTS 付きでコンパイルすると，このマクロを使用するルーチンでは， R3 と R4 をルーチンで使用していない場合でも，これらのレジスタが入力レジスタである可能性があるものとして表示されます。

.SYMBOL_ALIGNMENT

この指示文は，アラインメント属性とレジスタ・オフセットのシンボル定義を関連付けます。この指示文は，ベース・レジスタのアラインメントを知っている場合に使用できます。この属性は，ベース・レジスタのアラインメントが同じであることをコンパイラに対して保証します。これにより，コンパイラは最適なコードを生成することができます。

フォーマット

.SYMBOL_ALIGNMENT *argument-list*

パラメータ

argument-list
次の表に示すいずれかの引数を指定します。

オプション	説明
long	この指示文の後で宣言するすべてのシンボルに対し，ロングワード・アラインメントを宣言する。
quad	この指示文の後で宣言するすべてのシンボルに対し，クォドワード・アラインメントを宣言する。
none	それ以前に.SYMBOL_ALIGNMENT 指示文で指定したアラインメントを無効にする。

説明

.SYMBOL_ALIGNMENT 指示文は，ベース・アラインメントが分かっている場合に，アラインメント属性を構造体のフィールドに対応付けるために使用します。この指示文はペアで使用します。最初の.SYMBOL_ALIGNMENT 指示文は，ロングワード (long) アラインメントまたはクォドワード (quad) アラインメントをその後のシンボルに対応付けます。 2 番目の指示文.SYMBOL_ALIGNMENT none は，対応付けを無効にします。

アラインメント属性を持つシンボルが参照されるたび，その参照の実際のベース・レジスタがシンボルのアラインメントを継承します。また，その後のアラインメント追跡のために，コンパイラはベース・レジスタのアラインメントをロングワードにリセットします。このアラインメントの保証により，コンパイラはより効率の良いコード・シーケンスを生成します。

例

1.

```
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)
```

OFFSET1 および OFFSET5 に対しては，コンパイラはその追跡情報だけを使用して OFFSET1(Rn) の中の Rnがアラインされているかどうかを判断します。他の参照に対しては，ベース・レジスタはロングワード (OFFSET2 および OFFSET3) またはクォドワード (OFFSET4) にアラインされているものとして扱われます。

OFFSET2 または OFFSET4 を使用した後は，参照でのベース・レジスタはロングワード・アラインメントにリセットされます。この例では， OFFSET4 に対する参照ではより強力なクォドワード・アラインメントを使用していますが， R8 と R6 のアラインメントはロングワードにリセットされます。

.USE_LINKAGE (OpenVMS I64 のみ)

一時的な名前付きリンケージまたは匿名リンケージを確立します。これは，コンパイラによって，字句解析の順で次に処理される CALLS， CALLG， JSB， BSBB， BSBW 命令に対して使用されます。この指示文は，次の CALLS， CALLG， JSB， BSBB， BSBW のいずれかの命令のターゲットが名前でなく，実行時の値 (たとえば CALLS #0, (R6)) である場合に使用します。コンパイラが次の CALLS， CALLG， JSB， BSBB， BSBW のいずれかの命令を見つけると，関連付けられているリンケージを使用して，呼び出しの前後で保存と復元が必要なレジスタを決定します。命令の処理を終えると，一時リンケージはヌルにリセットされます。

フォーマット

.USE_LINKAGE　*[linkage_name] [,input] [,output] [,scratch] [,preserve]*

パラメータ

linkage_name
.DEFINE_LINKAGE 指示文で定義済みのリンケージの名前です。linkage_nameを指定すると，input句，output句，scratch句，preserve句は指定できません。

input=<>
次の CALLS，CALLG，JSB，BSBB，BSBW のいずれかの命令で呼び出されるルーチンが入力値を受け取るレジスタを示すレジスタ・セットです。

output=<>
次の CALLS，CALLG，JSB，BSBB，BSBW のいずれかの命令で呼び出されるルーチンが，その呼び出し元に返す値を代入するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれるレジスタは，呼び出しの前後で保存されません。

scratch=<>
次の CALLS，CALLG，JSB，BSBB，BSBW のいずれかの命令で呼び出されるルーチンの中で使用しているレジスタを示すレジスタ・セットです。このパラメータは，文書化のためにのみあります。

preserve=<>
次の CALLS，CALLG，JSB，BSBB，BSBW のいずれかの命令で呼び出されるルーチンが保護するレジスタを示すレジスタ・セットです。このレジスタ・セットに含まれるレジスタは，ルーチンの呼び出しの前後で保存と復元が行われません。この作業は，呼び出されるルーチンで行うためです。

C

MACRO コンパイラ・ビルトイン

この付録では， MACRO Compiler for OpenVMS Systems で提供されているビルトインについて説明します。

OpenVMS Alpha システムでは，以下の 2 種類のビルトインが提供されています。

- Alpha 命令ビルトイン。 VAX には対応する命令がない Alpha 命令にアクセスするために使用します (第 C.1 節を参照)。
- Alpha PALcode ビルトイン。 Alpha には対応する命令がない VAX 命令をエミュレートし，クォドワード・キュー操作などのその他の機能を実行するために使用します (第 C.2 節を参照)。

OpenVMS I64 システムでは，以下の 2 種類のビルトインが提供されています。

- Alpha 命令ビルトイン。 VAX には対応する命令がない Itanium 命令を生成するために使用します (第 C.1 節を参照)。
- Itanium 命令ビルトイン。 VAX には対応する命令がない Itanium 命令を生成するために使用します (第 C.3 節を参照)。

OpenVMS I64 システムでは，すべての Alpha PALcode ビルトインはシステムによって提供されるマクロでエミュレートされます。

両方の種類のビルトインを表に示します。それぞれの表の 2 番目の欄は，ビルトインに渡すオペランドを示します。略語の意味は以下のとおりです。

WL =書き込みロングワード (write longword)
ML =変更ロングワード (modify longword)
AL =ロングワードのアドレス (address of longword)
WQ =書き込みクォドワード (write quadword)
RQ =読み込みクォドワード (read quadword)
MQ =変更クォドワード (modify quadword)
AQ =クォドワードのアドレス (address of quadword)
AB =バイトのアドレス (address of byte)
AW =ワードのアドレス (address of word)
WB =書き込みバイト (write byte)
WW =書き込みワード (write word)

注意

同じレジスタに対してビルトインとVAX MACRO命令を混在させる場合は注意してください。コンパイラから生成されるコードは，レジスタに 32 ビット

の符号拡張された値が格納されていることを期待しますが，この形式でない 64 ビット値を作成することも可能です。そのようなレジスタに対してロングワード操作を行うと，正しい結果が得られません。

そのため，レジスタをVAX MACRO**命令のソース・オペランドとして使用する場合は，その前にレジスタを 32 ビットの符号拡張形式に戻してください。**VAX MACRO**命令 (MOVL など) を使用してレジスタに新しい値をロードすると，この形式に戻ります。**

C.1 Alpha 命令ビルトイン (OpenVMS Alpha システムおよび OpenVMS I64 システム向け)

移植したVAX MACROコードでは，64 ビットのサイズを直接扱ったり，VAX には相当する命令がない Alpha 命令を使用するために，Alpha のネイティブな命令にアクセスしなくてはならない場合があります。コンパイラでは，このような命令にアクセスできるようにするためのビルトインが提供されています。OpenVMS I64 システムでは，コンパイラは同等の Itanium 命令を生成します。

以下のバイトおよびワードのビルトインが MACRO コンパイラで提供されています。

- EVAX_LDBU
- EVAX_LDWU
- EVAX_STB
- EVAX_STW
- EVAX_SEXTB
- EVAX_SEXTW

これらのビルトインを使用する際には，ネイティブ VAX 命令と同じように，VAX オペランド・モードを使用します。たとえば，EVAX_ADDQ 8(R0),(SP)+,R1は有効です。唯一の例外は，Alpha のロード/格納ビルトイン (EVAX_LD*，EVAX_ST*) の最初のオペランドがレジスタであることです。

OpenVMS Alpha で，バイトとワードのビルトインが含まれているコードを実行する最適な環境は，これらの命令をハードウェアで実装しているシステムです。このようなコードを，ソフトウェアで命令をエミュレートする Alpha システムで実行すると，以下の制限があります。

- 大幅な性能低下

  例外を処理してソフトウェア・エミュレーションを起動するオーバヘッドがあるため，性能が大幅に低下します。ソフトウェア・エミュレーションが実施される場合，次のメッセージが出力されます。

```
%SYSTEM-I-EMULATED,
 an instruction not implemented on this processor was emulated
```

- ソフトウェア・エミュレーションに実装されていないいくつかの機能

  ソフトウェア・エミュレーションでは，その命令をハードウェアで実装しているシステム上にあるすべての機能を提供することはできません。内部アクセス・モードや高い IPL で動作するコードは，これらの命令を使用することができます。たとえば，IPL 2 よりも高いレベルのソフトウェア・エミュレーションを有効にすると，バグ・チェックは行われません。しかし，ハードウェア制御レジスタへの直接書き込みなど，これらの命令が有効なアプリケーションは実行が不可能になります。これは，そのようなアプリケーションでは，機能がエミュレートできないようなアドレス・ラインの存在を必要とするためです。

さらに，これらのビルトインを使用したコードを，バイトおよびワードのソフトウェア・エミュレータや，バイト命令とワード命令がハードウェアで実装されたプロセッサがないシステムで実行すると，次のように回復不可能な例外が発生します。

```
%SYSTEM-F-OPCDEC, opcode reserved to Digital fault at
PC=0000000000020068,PS=0000001B
```

注意

MACRO コンパイラ・ビルトイン内のメモリ参照は，常にクォドワードでアラインされていると想定されます。ただし，EVAX_SEXTB**，**EVAX_SEXTW**，**EVAX_LDBU**，**EVAX_LDWU**，**EVAX_STB**，**EVAX_STW**，**EVAX_LDQU**，および** EVAX_STQU **を除きます。**

表 C–1 に，コンパイラでサポートされている Alpha ビルトインの要約を示します。Alpha 専用の (Itanium 命令を生成するためや Itanium 命令にアクセスするために使用できない) ビルトインは，表にその旨明記しています。

表 C–1 Alpha 命令ビルトイン (OpenVMS Alpha システムおよび OpenVMS I64 システム向け)

ビルトイン	オペランド	説明	OpenVMS I64 で動作するか
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	○

(次ページに続く)

表 C–1 (続き) Alpha 命令ビルトイン (OpenVMS Alpha システムおよび OpenVMS I64 システム向け)

ビルトイン	オペランド	説明	OpenVMS I64 で動作するか
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	○
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	○

(次ページに続く)

表 C–1 (続き) Alpha 命令ビルトイン (OpenVMS Alpha システムおよび OpenVMS I64 システム向け)

ビルトイン	オペランド	説明	OpenVMS I64 で動作するか
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	○
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 ビルトイン

Alpha PALcode ビルトインは，主に特権コード用であり， Alpha 命令ビルトインと同じように使用しますが，例外が 2 つあります。

- キュー PAL 関数では，その他すべての関数と異なり，コンパイラは PAL 呼び出しを行う前に入力引数を Alpha レジスタに移動しません。そのため，この処理を実行するコードを記述する必要があります。
- 値を返す PAL 呼び出しのビルトインを使用する場合は，ソース・コードで明示的に R0 から戻り値を読み込む必要があります。

Alpha PALcode ビルトイン EVAX_INSQHIQR， EVAX_INSQTIQR，EVAX_REMQHIQR，および EVAX_REMQHITR は，クォドワード・キューの操作をサポートしていますが，この機能はVAX MACROではサポートされていません。これらのビルトインを使用する場合は，次の例に示すように，入力引数を R16 (およびEVAX_INSQxxxxでは R17) に移動するコードを記述する必要があります。

```
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
```

Alpha PALcode ビルトインを表 C–2 に示します。

注意

コンパイラ指示文.DEFINE_PAL **を使用することで，この表に載っていない** Alpha PALcode **操作のビルトインを定義することができます。付録** B, **専用の指示文を参照してください。**

OpenVMS I64 **システムでは，ビルトインの多くはシステム提供のマクロでエミュレートされています。**

表 C–2 Alpha PALcode **ビルトイン**

ビルトイン	オペランド	説明
EVAX_CFLUSH	<RQ>	Cache flush
EVAX_DRAINA	<>	Drain aborts
EVAX_LDQP	<AQ>	Load quadword physical
EVAX_STQP	<AQ,RQ>	Store quadword physical
EVAX_SWPCTX	<AQ>	Swap privileged context
EVAX_BUGCHK	<RQ>	Bugcheck
EVAX_CHMS	<>	Change mode supervisor

(次ページに続く)

表 C–2 (続き) Alpha PALcode ビルトイン

ビルトイン	オペランド	説明
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
EVAX_MFPR_ESP	<>	Move from processor register ESP
EVAX_MFPR_SSP	<>	Move from processor register SSP

(次ページに続く)

表 C–2 (続き) Alpha PALcode ビルトイン

ビルトイン	オペランド	説明
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

C.3 Itanium 命令ビルトイン (OpenVMS I64 システム向け)

表 C–3 Itanium 命令ビルトイン (OpenVMS I64 システム向け)

ビルトイン	オペランド	説明
IA64_BREAK	<RQ>	指定したイミディエイト・オペランドを使用してブレーク命令フォルトを生成する。
IA64_GETINDREG	<WQ,RQ,RQ>	move-from-indirect-register 命令を生成する。第 1 オペランドがデスティネーション，第 2 オペランドがアクセス対象の間接レジスタ・ファイルを指定するリテラル[1]，第 3 オペランドがレジスタ・ファイルに対するインデックスとなる。
IA64_GETREG	<WQ,RQ>	move-from-application-register 命令または move-from-control-register 命令を生成する。第 1 オペランドがデスティネーション，第 2 オペランドが読み込み対象のアプリケーション・レジスタまたは制御レジスタを指定するリテラル[2]となる。
IA64_LFETCH IA64_LFETCH_EXCL	<RQ,RQ>	行プリフェッチ ('LFETCH') 命令または排他的行プリフェッチ ('LFETCH.EXCL') 命令を生成する。第 1 オペランドはプリフェッチするアドレス，第 2 オペランドは reg-base-update-form または imm-base-update-form のどちらかとなる。オペランドがリテラルのゼロの場合は， no-base-update-form が使用される。
IA64_PROBER	<WQ,RQ,RQ>	probe.r 命令を生成する。第 1 オペランドはデスティネーション，第 2 オペランドはプローブ対象の仮想アドレス，第 3 オペランドは特権レベルとなる。
IA64_PROBEW	<WQ,RQ,RQ>	probe.w 命令を生成する。第 1 オペランドはデスティネーション，第 2 オペランドはプローブ対象の仮想アドレス，第 3 オペランドは特権レベルとなる。
IA64_RSM	<RQ>	指定されたマスクを使用して，リセット・システム・マスク ('RSM') 命令を生成する。
IA64_RUM	<RQ>	指定されたマスクを使用して，リセット・ユーザ・マスク ('RUM') 命令を生成する。
IA64_SETREG	<RQ,RQ>	move-to-application-register 命令または move-to-control-register 命令を生成する。第 1 オペランドは書き込み先のアプリケーション・レジスタまたは制御レジスタを指定するリテラル[2]，第 2 オペランドはレジスタに書き込む値となる。
IA64_SRLZD	<>	serialize data ('SRLZD') 命令を生成する。
IA64_SRLZI	<>	serialize instruction ('SRLZI') 命令を生成する。
IA64_SSM	<RQ>	指定されたマスクを使用して， set system mask ('SSM') 命令を生成する。
IA64_SUM	<RQ>	指定されたマスクを使用して set user mask ('SUM') 命令を生成する。
IA64_TAK	<WK,RQ>	read translation access key ('TAK') 命令を生成する。

[1]有効な間接レジスタ・ファイルの一覧は，ファイル SYS$LIBRARY:STARLET.MLB のモジュール$IA64REGDEF に接頭辞 IA64_REG$_INDR で格納されています。

[2]有効なアプリケーション・レジスタと制御レジスタの一覧は，ファイル SYS$LIBRARY:STARLET.MLB のモジュール$IA64REGDEF に接頭辞 IA64_REG$_AR および IA64_REG$_CR で格納されています。

D

VAX から Alpha または I64 への移植用のマクロ

この付録では，VAX MACROコードを OpenVMS Alpha システムまたは OpenVMS I64 システムに移植する際に役立つマクロについて説明します。マクロは，機能ごとに以下のグループに分類されています。

- 第 D.1 節, ページ・サイズ値の計算
- 第 D.2 節, 64 ビット・レジスタの保存と復元
- 第 D.3 節, ワーキング・セットへのページのロック

この付録で説明するマクロでは，レジスタ・セットを示す引数を使用できます。レジスタ・セットを表現するには，次の例のようにレジスタをコンマで区切り，山括弧で囲みます。

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

レジスタ・セットにレジスタが 1 つしか含まれない場合は，次のように山括弧を省略します。

```
R1
```

D.1 ページ・サイズ値の計算

以下のマクロは，ページ・サイズに依存する値を計算するための，標準的でアーキテクチャに依存しない方法を提供します。

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

これらのマクロはディレクトリ SYS$LIBRARY:STARLET.MLB にあり，アプリケーション・コードとシステム・コードの両方で使用できます。アプリケーション・コードは SYSTEM_DATA_CELLS にアクセスできないため，適切なマスク，シフト値などを指定する必要があります。

シフト値はプロセッサのページ・サイズと関係があります。表 D–1 に示すように，rightshift値は負で，leftshift値は正です。

表 D–1 シフト値

ページ・サイズ	rightshift	leftshift
512 バイト (VAX)	-9	9
8K (OpenVMS Alpha または OpenVMS I64)	-13	13
16K[1]	-14	14
32K[1]	-15	15
64K[1]	-16	16

[1]将来の OpenVMS Alpha システムまたは OpenVMS I64 システムでこのアーキテクチャ上許される大きなページ・サイズが実装された場合。

通常，アプリケーションは$GETSYI (アイテム記述子 SYI$_PAGESIZE を指定) を呼び出して CPU 固有のページ・サイズを取得し，それを元にその他の値を計算します。

以下の規則がこの項で説明するマクロに適用されます。

- デスティネーション・オペランドに何も指定しないと，ソース・オペランドがデスティネーションとして使用されます。
- すべてのマクロは，シンボル VAXPAGE および BIGPAGE に対する条件付きでコードが生成されます。
- いくつかのマクロでは，independent=YES引数を指定することで， VAX システム上でページ・サイズに依存しないコードとなります。これらのマクロが生成するコードでは， I ストリーム・フェッチがメモリ・アクセスに変更されます。これは，その性質上 VAX システムでは低速であるため，independent引数のデフォルト値はNOになっています。

$BYTES_TO_PAGES

バイト数をページ数に変換します。

フォーマット

$BYTES_TO_PAGES *source_bytcnt, dest_pagcnt, rightshift, roundup=YES, quad=YES*

パラメータ

source_bytcnt
ソースバイト数です。

dest_pagcnt
ページ数の格納先です。

rightshift
アプリケーション指定のシフト値のアドレスです (かけ算の代わり)。この値は，表 D–1 に示すように，ページ・サイズの関数となっています。

roundup=YES
YESの場合，シフトの前に「ページ・サイズ –1」がバイト数に加算されます。NOの場合，ページ数は切り捨てられます。その他の値は，「ページ・サイズ –1」の値を指すユーザ指定のアドレスとして扱われます。roundup=YESは，rightshift引数と一緒に指定できない点に注意してください。両方の引数を指定してマクロを実行すると，コンパイル時に警告が出力されます。

quad=YES
YESの場合，変換で 64 ビット・アドレッシングがサポートされます。NOの場合，変換で 64 ビット・アドレッシングがサポートされません。

$NEXT_PAGE

次のページの先頭バイトの仮想アドレスを計算します。

フォーマット

$NEXT_PAGE *source_va, dest_va, clearbwp=NO, user_pagesize_addr, user_mask_addr, quad=YES*

パラメータ

source_va
ソース仮想アドレスです。

dest_va
次のページの仮想アドレスの格納先です。

clearbwp=NO
YESの場合，ソース仮想アドレスのページ内のバイトの部分をマスクします。clearbwp=NOオプションを指定すると，ページ境界を指定していることが分

かっている場合や，ページ・サイズで除算しようとしている場合に不要な命令が実行されなくなり，性能が向上します。

user_pagesize_addr
アプリケーション・データ領域にあるページ・サイズ値のアドレスです(アイテム記述子 SYI$_PAGESIZE を指定して$GETSYI システム・サービスを呼び出すことで返されます)。この引数を省略した場合，マクロは MMG$GL_PAGESIZE (bigpage) または MMG$C_VAX_PAGE_SIZE (vaxpage) を使用します。

user_mask_addr
アプリケーションが用意するページ内バイトのマスクのアドレスです。この引数を省略した場合，user_pagesize_addrも省略した場合は MMG$GL_BWP_MASK が使用され，そうでない場合はuser_pagesize_addrの内容から 1 を引き，その値が使用されます。

quad=YES
YESの場合，変換で 64 ビット・アドレッシングがサポートされます。NOの場合，変換で 64 ビット・アドレッシングがサポートされません。

$PAGES_TO_BYTES

ページ数をバイト数に変換します。

フォーマット

$PAGES_TO_BYTES *source_pagcnt, dest_bytcnt, leftshift, quad=YES*

パラメータ

source_pagcnt
ソース・ページ数です。

dest_bytcnt
バイト数の格納先です。

leftshift
アプリケーション指定のシフト値のアドレスです(かけ算の代わり)。この値は，表 D–1 に示すように，ページ・サイズの関数となっています。

quad=YES
YESの場合，変換で 64 ビット・アドレッシングがサポートされます。NOの場合，変換で 64 ビット・アドレッシングがサポートされません。

$PREVIOUS_PAGE

前のページの先頭バイトの仮想アドレスを計算します。

フォーマット

$PREVIOUS_PAGE *source_va, dest_va, clearbwp=NO, user_pagesize_addr, user_mask_addr, quad=YES*

パラメータ

source_va
ソース仮想アドレスです。

dest_va
前のページの仮想アドレスの格納先です。

clearbwp=NO
YESの場合，ソース仮想アドレスのページ内のバイトの部分をマスクします。clearbwp=NOオプションを指定すると，ページ境界を指定していることが分かっている場合や，ページ・サイズで除算しようとしている場合に不要な命令が実行されなくなり，性能が向上します。

user_pagesize_addr
アプリケーション・データ領域にあるページ・サイズ値のアドレスです(アイテム記述子 SYI$_PAGESIZE を指定して$GETSYI システム・サービスを呼び出すことで返されます)。この引数を省略した場合，マクロは MMG$GL_PAGESIZE (bigpage) または MMG$C_VAX_PAGE_SIZE (vaxpage) を使用します。

user_mask_addr
アプリケーションが用意するページ内バイトのマスクのアドレスです。この引数を省略した場合，user_pagesize_addrも省略した場合は MMG$GL_BWP_MASK が使用され，そうでない場合はuser_pagesize_addrの内容から 1 を引き，その値が使用されます。

quad=YES
YESの場合，変換で 64 ビット・アドレッシングがサポートされます。NOの場合，変換で 64 ビット・アドレッシングがサポートされません。

$ROUND_RETADR

メモリ管理サービスから返されたretadr配列内の仮想アドレスが示す範囲を，CPU 固有のページに基づく範囲に丸めます。戻り値は，それ以降の別のメモリ管理システム・サービスの呼び出しで，inadr配列として指定できます。

フォーマット

$ROUND_RETADR *retadr, full_range, user_mask_addr, direction=ASCENDING*

パラメータ

retadr
2 つの 32 ビット・アドレスからなる配列のアドレスです。通常，$CRMPSC またはそれに似たサービスから返されます。この値の形式は，"label"または "(Rx)"です。

full_range
2 つのロングワードからなる出力配列。FULL_RANGE[0]は，retadr[0]を CPU 固有のページ境界に丸めたもの (切り捨て) で，FULL_RANGE[1]は，retadr[1]を CPU 固有のページ境界に切り上げ，1 を引いたもの (つまり，ページ内の最後のバイト) です。

user_mask_addr
アプリケーションが用意するページ内バイトのマスクのアドレスです。この引数を省略した場合，OpenVMS Alpha システムまたは OpenVMS I64 システムでは MMG$GL_BWP_MASK が使用され，OpenVMS VAX システムでは VA$M_BYTE が使用されます。

direction=ASCENDING
丸めの方向です。キーワードを次の表に示します。

ASCENDING	retadr[0] < retadr[1]
DESCENDING	retadr[1] < retadr[0]
UNKNOWN	値は実行時に比較され，正しく丸められる。

$START_OF_PAGE

仮想アドレスを，ページ内の先頭バイトのアドレスに変換します。

フォーマット

$START_OF_PAGE　*source_va, dest_va, user_mask_addr, quad=YES*

パラメータ

source_va
ソース仮想アドレスです。

dest_va
ページ内の先頭バイトの仮想アドレスの格納先です。

user_mask_addr
アプリケーションが用意するページ内バイトのマスクのアドレスです。この引数を省略した場合， OpenVMS Alpha システムまたは OpenVMS I64 システムでは MMG$GL_BWP_MASK が使用され， OpenVMS VAX システムでは MMG$C_VAX_PAGE_SIZE - 1 ($pagedef で定義) が使用されます。

quad=YES
YESの場合，変換で 64 ビット・アドレッシングがサポートされます。NOの場合，変換で 64 ビット・アドレッシングがサポートされません。

D.2　64 ビット・レジスタの保存と復元

VAX MACROソース・コードでレジスタ値を保存して復元しなければならないことがよくあります。その理由としては，この処理がインタフェースの一部として定義されていることや，コードが作業レジスタを必要とすることが考えられます。

OpenVMS VAX では，コード上で任意の数のマクロを呼び出してこの処理を行えます。 OpenVMS Alpha と OpenVMS I64 では，これらのマクロを，スタックに対する 64 ビット版のプッシュとポップに単純に置き換えることはできません。マクロの呼び出し元が，クォドワード境界にアラインされたスタックを持っているとは限らないためです。代わりに，このようなマクロを$PUSH64 マクロと$POP64 マクロで置き換えます。これらのマクロは STARLET.MLB にあり，レジスタの 64 ビット値全体を保存/復元しますが，ロングワード参照を使用してこの処理を実行します。

$POP64

スタックの先頭の 64 ビット値をレジスタにポップします。

フォーマット

$POP64 *reg*

パラメータ

reg
スタックの先頭から取り出した 64 ビット値を格納するレジスタです。

説明

$POP64 は，スタックの先頭にある 64 ビット値を取り出し，ロングワード命令を使用してレジスタに格納します。これは，アラインメント・フォルトを避けなければならない場合で，しかも 64 ビット全体を復元する必要がある場合に，クォドワード命令の使用を避けるために使用します。

$PUSH64

64 ビット・レジスタの内容をスタックにプッシュします。

フォーマット

$PUSH64 *reg*

パラメータ

reg
スタックにプッシュするレジスタです。

説明

$PUSH64 は 64 ビット・レジスタを受け取り，ロングワード命令を使用してスタックに格納します。これは，アラインメント・フォルトを避けなければならない場合で，しかも 64 ビット全体を格納する必要がある場合に，クォドワード命令の使用を避けるために使用します。

D.3 ワーキング・セットへのページのロック

ワーキング・セットにページをロックするために，5 つのマクロが提供されています。これらのマクロは SYS$LIBRARY:LIB.MLB にあります。これらのマクロの使用方法については，第 3.10 節を参照してください。

3 つのマクロは，イメージ初期化時のロックダウンで使用し，2 つのマクロは，オンザフライのロックダウンで使用します。

注意

IPL を，ページ・フォルトが発生しない 2 よりも上げることでコードをロックする場合は，その範囲内のコードが実行時ライブラリやその他のプロシージャを呼び出さないことを確認してください。VAX MACRO**コンパイラは，特定の** VAX **命令をエミュレートするためのルーチンの呼び出しを生成します。これらのマクロを使用するイメージは，これらのルーチンの参照がページング不可のエグゼクティブ・イメージ内のコードで解決されるように，システム・ベース・イメージとリンクする必要があります。**

OpenVMS I64 **システムでは，これらのマクロはまだ開発中であり，ワーキング・セットをロックするための追加の** OpenVMS **ルーチンが提供されます。詳細は，**OpenVMS V8.2 **リリース・ノート[翻訳版]を参照してください。**

D.3.1 イメージ初期化時のロックダウン

以下のマクロは，イメージ初期化時のロックダウンで使用します。

- $LOCK_PAGE_INIT
- $LOCKED_PAGE_END
- $LOCKED_PAGE_START

$LOCK_PAGE_INIT

$LOCKED_PAGE_START と$LOCKED_PAGE_END を使用して初期化時にロックする領域を線引きするイメージの初期化ルーチンで使用する必要があります。

フォーマット

$LOCK_PAGE_INIT *[error]*

パラメータ

[error]
いずれかの$LKWSET 呼び出しが失敗した場合に分岐する分岐先アドレスです。このアドレスに到達した場合， R0 には失敗した呼び出しの状態が設定されます。 R1 には，コードをロックするための呼び出しに失敗した場合は 0，その呼び出しは成功したものの，リンケージ・セクションをロックするための呼び出しが失敗した場合は 1 が設定されます。

説明

$LOCK_PAGE_INIT は，必要な psect を作成し，$LWKSET を呼び出して，$LOCKED_PAGE_START と$LOCKED_PAGE_END で定義されたコード・セクションとリンケージ・セクションを，ワーキング・セット内にロックします。 R0 と R1 の内容は，このマクロによって壊れます。

このマクロでロックされる psect は$LOCK_PAGE_2 と$LOCK_LINKAGE_2 です。他の言語で記述された他のモジュールのコード・セクションでこれらの psect を使用している場合は，VAX MACROモジュール内でこのマクロを呼び出すことでロックされます。

$LOCKED_PAGE_END

イメージの初期化時に$LOCK_PAGE_INIT マクロによってロックされるコード・セクションの終わりをマークします。

フォーマット

$LOCKED_PAGE_END *[link_sect]*

パラメータ

[link_sect]
$LOCKED_PAGE_START マクロを実行したときの実際のリンケージ psect がデフォルトのリンケージ psect $LINKAGE でなかった場合に戻る psect です。

説明

$LOCKED_PAGE_END は$LOCKED_PAGE_START とともに使用され，イメージの初期化時に$LOCK_PAGE_INIT マクロによって初期化されるコードを線引きします。これらのマクロで線引きされたコードには，ルーチン全体が含まれている必要があります。実行がいずれかのマクロを超えたり，ロックされたコードから外に分岐したり，外からロックされたコードの中に分岐することはできません。ロックされたコード・セクションの中に分岐した場合や，ロックされたコード・セクションから外に分岐しようとした場合，またはマクロを超えて実行しようとした場合は，コンパイラによってエラーが出力されます。

$LOCKED_PAGE_START

イメージの初期化時に$LOCK_PAGE_INIT マクロによってロックされるコード・セクションの先頭をマークします。

フォーマット

$LOCKED_PAGE_START

このマクロには，パラメータはありません。

説明

$LOCKED_PAGE_START は$LOCKED_PAGE_END とともに使用され，イメージの初期化時に$LOCK_PAGE_INIT マクロによって初期化されるコードを線引きします。これらのマクロで線引きされたコードには，ルーチン全体が含まれている必要があります。実行がいずれかのマクロを超えたり，ロックされたコードから外に分岐したり，外からロックされたコードの中に分岐することはできません。ロックされたコード・セクションの中に分岐した場合や，ロックされたコード・セクションから外に分岐しようとした場合，またはマクロを超えて実行しようとした場合は，コンパイラによってエラーが出力されます。

D.3.2 オンザフライのロックダウン

以下のマクロは，オンザフライのロックダウンで使用されます。

- $LOCK_PAGE
- $UNLOCK_PAGE

$LOCK_PAGE

オンザフライでロックするコード・セクションの先頭をマークします。

フォーマット

$LOCK_PAGE *[error]*

パラメータ

[error]
いずれかの$LKWSET 呼び出しが失敗した場合に分岐するアドレスです。

説明

このマクロは，実行可能コードにインラインで配置され，この後で$UNLOCK_PAGE マクロを呼び出す必要があります。$LOCK_PAGE/$UNLOCK_PAGE マクロのペアは，個別の psect に個別のルーチンを作成します。$LOCK_PAGE は，この個別のルーチンのページとリンケージ・セクションをワーキング・セットにロックして， JSR でそこにジャンプします。このマクロとそれに対応する$UNLOCK_PAGE マクロの間のすべてのコードが，ロックされるルーチンに入れられ，ロックダウンされます。

このマクロでは，すべてのレジスタが保護されます。ただし，エラー・アドレス・パラメータが指定されており，いずれかの呼び出しが失敗すると， R0 には失敗した呼び出しの状態が設定されます。 R1 には，コードをロックするための呼び出しに失敗した場合は 0，その呼び出しは成功したものの，リンケージ・セクションをロックするための呼び出しが失敗した場合には 1 が設定されます。

error パラメータを使用する場合は， error ラベルは$LOCK_PAGE と$UNLOCK_PAGE のペアの外側に配置する必要があります。これは，$LOCK_PAGE

と$UNLOCK_PAGE で作成されたサブルーチンを呼び出す前にエラー・ルーチンに分岐するためです。

ロックされるコードは個別のルーチンになり，スタック・コンテキストは同じでなくなるため，ルーチン内でのローカル・スタック・ストレージを参照しているときには，すべて変更する必要があります。また，ルーチンの他の部分から，ロックされるコードに分岐したり，逆にロックされるコードからルーチンの他の部分に分岐することはできません。

$UNLOCK_PAGE

オンザフライでロックするコード・セクションの終わりをマークします。

フォーマット

$UNLOCK_PAGE *[error][,LINK_SECT]*

パラメータ

[error]
いずれかの$ULKWSET 呼び出しが失敗した場合に分岐するエラー・アドレスです。

[link_sect]
$LOCK_PAGE マクロを実行したときの実際のリンケージ psect がデフォルトのリンケージ psect $LINKAGE でなかった場合に戻るリンケージ psect です。

説明

$UNLOCK_PAGE は，$LOCK_PAGE マクロと$UNLOCK_PAGE マクロのペアで作成された，ロックされるルーチンから戻り，ページとリンケージ・セクションをワーキング・セットからアンロックします。このマクロは，実行可能コード内で，$LOCK_PAGE マクロの後にインラインで配置します。

このマクロでは，すべてのレジスタが保護されます。ただし，エラー・アドレス・パラメータが指定されており，いずれかの呼び出しが失敗すると， R0 には失敗した呼び出しの状態が設定されます。 R1 には，コードをアンロックするための呼び出しに失敗した場合は 0，その呼び出しは成功したものの，リンケージ・セクションをアンロックするための呼び出しが失敗した場合には 1 が設定されます。

errorパラメータを使用する場合は，errorラベルは$LOCK_PAGE と$UNLOCK_PAGE のペアの外側に配置する必要があります。これは，$LOCK_PAGE と$UNLOCK_PAGE で作成されたサブルーチンから戻った後でエラー・ルーチンに分岐するためです。

E

64 ビット・アドレッシング用のマクロ

この付録では，以下の内容について説明します。

- 第 E.1 節, 64 ビット・アドレスを操作するためのマクロ
- 第 E.2 節, 符号拡張と記述子の形式をチェックするためのマクロ

これらのマクロは，ディレクトリ SYS$LIBRARY:STARLET.MLB にあり，アプリケーション・コードとシステム・コードの両方で使用することができます。

ページ・マクロは， 64 ビット・アドレスをサポートしています。このサポートは，QUAD=NO/YES パラメータによって提供されます。

これらのマクロで特定の引数を使用することで，レジスタ・セットを指定できます。レジスタ・セットを表現するには，次の例のようにレジスタをコンマで区切り，山括弧で囲みます。

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

レジスタ・セットにレジスタが 1 つしか含まれない場合は，山括弧は不要です。

E.1 64 ビット・アドレスを操作するためのマクロ

以下のマクロは， 64 ビット・アドレスを操作するために使用します。

- $SETUP_CALL64
- $PUSH_ARG64
- $CALL64

$SETUP_CALL64

呼び出し手順を初期化します。

フォーマット

$SETUP_CALL64 *arg_count, inline=true* | *false*

パラメータ

arg_count
呼び出しでの引数の数です。

inline
TRUE の場合，JSB ルーチンを作成するのではなく，インライン展開します。引数の数が，OpenVMS Alpha で 6 以下，OpenVMS I64 で 8 以下の場合は，デフォルトはinline=trueです。

説明

このマクロは，64 ビット呼び出しの状態を初期化します。$PUSH_ARG64 や$CALL64 を使用する前に使用する必要があります。

引数の数が，OpenVMS Alpha で 6 以下，OpenVMS I64 で 8 以下の場合は，コードは常にインラインになります。

デフォルトでは，引数の数が OpenVMS Alpha で 7 以上，OpenVMS I64 で 9 以上の場合，このマクロは実際の呼び出しを行うために呼び出される JSB ルーチンを作成します。ただし，インライン・オプションinline=trueが指定されていると，コードはインラインで生成されます。

このオプションは，このマクロを使用するコードのスタックの深さが固定の場合にのみ有効にしてください。RUNTIMSTK メッセージまたは VARSIZSTK メッセージがこれまで報告されたことがない場合は，スタックの深さが固定であると考えることができます。そうでない場合は，スタック・アラインメントが少なくともクォドワードでなければ，呼び出されたルーチンと，それが呼び出すすべてのもので多数のアラインメント・フォルトが発生します。デフォルトの動作 (inline=false) では，この問題はありません。

引数の数が OpenVMS Alpha で 7 以上，OpenVMS I64 で 9 以上の場合，$SETUP_CALL64 とそれに対応する$CALL64 の間で AP や SP を参照することはできません。これは，$CALL64 コードが独立した JSB ルーチンにある可能性があるためです。また，一時レジスタ (R16 以降) は，$SETUP_CALL64 を呼び出すと内容が変わってしまう可能性があります。ただし，R16 ～ R12 がすでに設定済みの引数レジスタと干渉する場合を除き (Alpha のみ)，一時レジスタをこの範囲で使用することができます。干渉する場合は，より番号の大きな一時レジスタを使用してください。

注意

$SETUP_CALL64，$PUSH_ARG64，および$CALL64 マクロは，インライン・シーケンスで使用するように設計されています。つまり，$SETUP_CALL64/$PUSH_ARG64/$CALL64 シーケンスの途中に分岐したり，$PUSH_

ARG64 マクロの前後で分岐したり，$CALL64 を呼び出さずにこのシーケンスから外に分岐することはできません。

$PUSH_ARG64

呼び出しのための引数プッシュに相当する動作を行います。

フォーマット

$PUSH_ARG64 *argument*

パラメータ

argument
プッシュする引数です。

説明

このマクロは， 64 ビット呼び出し用に 64 ビット引数をプッシュします。$PUSH_ARG64 を使用する前に，マクロ$SETUP_CALL64 を使用する必要があります。

引数はアラインされたクォドワードとして読み込まれます。つまり，$PUSH_ARG64 4(R0) は 4(R0) の位置にあるクォドワードを読み込み，そのクォドワードをプッシュします。インデックス付きの操作はクォドワード・モードで実行されます。

メモリから読み込んだロングワード値をクォドワードとしてプッシュするには，まずロングワード命令を使用してレジスタに移動し，次にレジスタに対して$PUSH_ARG64 を使用します。同様に，アラインされていない事が分かっているクォドワード値をプッシュするには，まず一時レジスタに移動し，その後$PUSH_ARG64 を使用します。

呼び出しに含まれている引数の数が， OpenVMS Alpha で 7 以上， OpenVMS I64 で 9 以上の場合は，このマクロは引数の中で SP や AP が参照されていないかチェックします。

呼び出しに含まれている引数の数が， OpenVMS Alpha で 7 以上， OpenVMS I64 で 9 以上の場合は， SP 参照は行うことができず， AP 参照はインライン・オプションを使用している場合にのみ行えます。

OpenVMS Alpha システムのみ: マクロは，現在の$CALL64 に対してすでに設定されている引数レジスタの参照もチェックします。このような参照が見つかると，警告が報告され，$PUSH_ARG64 の中でソースとして使用する前に引数レジスタを上書きしないように注意が促されます。

OpenVMS Alpha システムのみ: 引数の数が 6 以下の場合，同じチェックが AP 参照に対して行われます。参照自体は可能ですが，それを使用する前にプログラムが上書きしてしまうのを防ぐことは，コンパイラにはできません。そのため，このような参照が見つかると，情報メッセージが出力されます。

OpenVMS Alpha システムのみ: オペランドで，名前に R16 ～ R21 のいずれかの文字列が含まれたシンボル(レジスタ参照ではない)が使用されていると，このマクロで誤ってエラーが出力されることがあります。たとえば， R21 が設定された後で$PUSH_ARG64 SAVED_R21 を呼び出すと，このマクロは，引数レジスタの上書きに関する情報メッセージを誤って出力します。

また，$PUSH_ARG64 は条件付きコード内では使用できません。$PUSH_ARG64 は，残りの引数の個数など，いくつかのシンボルを更新します。$SETUP_CALL64 /$CALL64 シーケンスの途中に，$PUSH_ARG64 の前後で分岐するコードを記述すると，正常に機能しません。

$CALL64

ターゲット・ルーチンを呼び出します。

フォーマット

$CALL64　*call_target*

パラメータ

call_target
呼び出すルーチンです。

説明

このマクロは，$SETUP_CALL64 を使用して引数の個数が指定され，$PUSH_ARG64 を使用してクォドワード引数がプッシュされたものと見なして，指定されたルーチンを呼び出します。このマクロは，プッシュの回数が，セットアップの呼び出しで指定された個数と一致するかどうかを確認します。

call_targetオペランドは，APベースやSPベースであってはなりません。

E.2 符号拡張と記述子の形式をチェックするためのマクロ

以下のマクロは，特定の値をチェックし，チェック結果に基づいてプログラム・フローを変更するために使用します。

- $IS_32BITS
- $IS_DESC64

$IS_32BITS

64 ビット値の下位 32 ビットの符号拡張をチェックし，チェック結果に基づいてプログラム・フローを変更します。

フォーマット

$IS_32BITS *quad_arg, leq_32bits, gtr_32bits, temp_reg=22*

パラメータ

quad_arg
レジスタまたはアラインされたクォドワード・メモリ領域にある 64 ビット・データ。

leq_32bits
quad_arg が 32 ビットの符号拡張値の場合に分岐する分岐先ラベル。

gtr_32bits
quad_arg が 32 ビットよりも大きな場合に分岐する分岐先ラベル。

temp_reg=22
ソース値の下位ロングワードを保持するための一時レジスタとして使用するレジスタ。デフォルトは R22。

説明

$IS_32BITS は，64 ビット値の下位 32 ビットの符号拡張をチェックし，チェック結果に基づいてプログラム・フローを変更します。

例

1. `$is_32bits R9, 10$`

   この例では，デフォルトの一時レジスタ R22 を使用し，コンパイラは R9 に格納されている 64 ビット値の下位 32 ビットの符号拡張をチェックします。分岐の種類とテストの結果に応じて，プログラムは分岐するかそのまま続行します。

2. `$is_32bits 4(R8), 20$, 30$, R28`

   この例では， R28 を一時レジスタとして使用し， 4(R8) に格納されている 64 ビット値の下位 32 ビットの符号拡張をチェックし，チェック結果に基づいて 20$または 30$に分岐します。

$IS_DESC64

指定された記述子をテストして 64 ビット形式の記述子かどうかを判断し，テスト結果に基づいてプログラム・フローを変更します。

フォーマット

$IS_DESC64 *desc_addr, target, size=long | quad*

パラメータ

desc_addr
テストする記述子のアドレスです。

target
記述子が 64 ビット形式の場合に分岐する分岐先ラベルです。

size=long | quad
記述子を指すアドレスのサイズです。デフォルト値はsize=longです。

説明

$IS_DESC64 は， 64 ビット記述子と 32 ビット記述子を区別するフィールドをテストします。 64 ビット形式の場合は，指定したターゲットに分岐します。テストするアドレスは，size=quadを指定しないかぎりロングワードとして読み込まれます。

例

1. `$is_desc64 r9, 10$`

   この例では， R9 が指す記述子をテストし， 64 ビット形式である場合は， 10$に分岐します。

2. `$is_desc64 8(r0), 20$, size=quad`

   この例では， 8(R0) にあるクォドワードを読み込み，それが指す記述子をテストします。 64 ビット形式である場合は， 20$に分岐します。

索引

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

ア

イ

エ

オ

カ

キ

ク

ケ

コ

サ

シ

フ

ヘ

ホ

マ

ミ

メ

モ

ユ

ヨ

ラ

リ

ル

レ

ロ