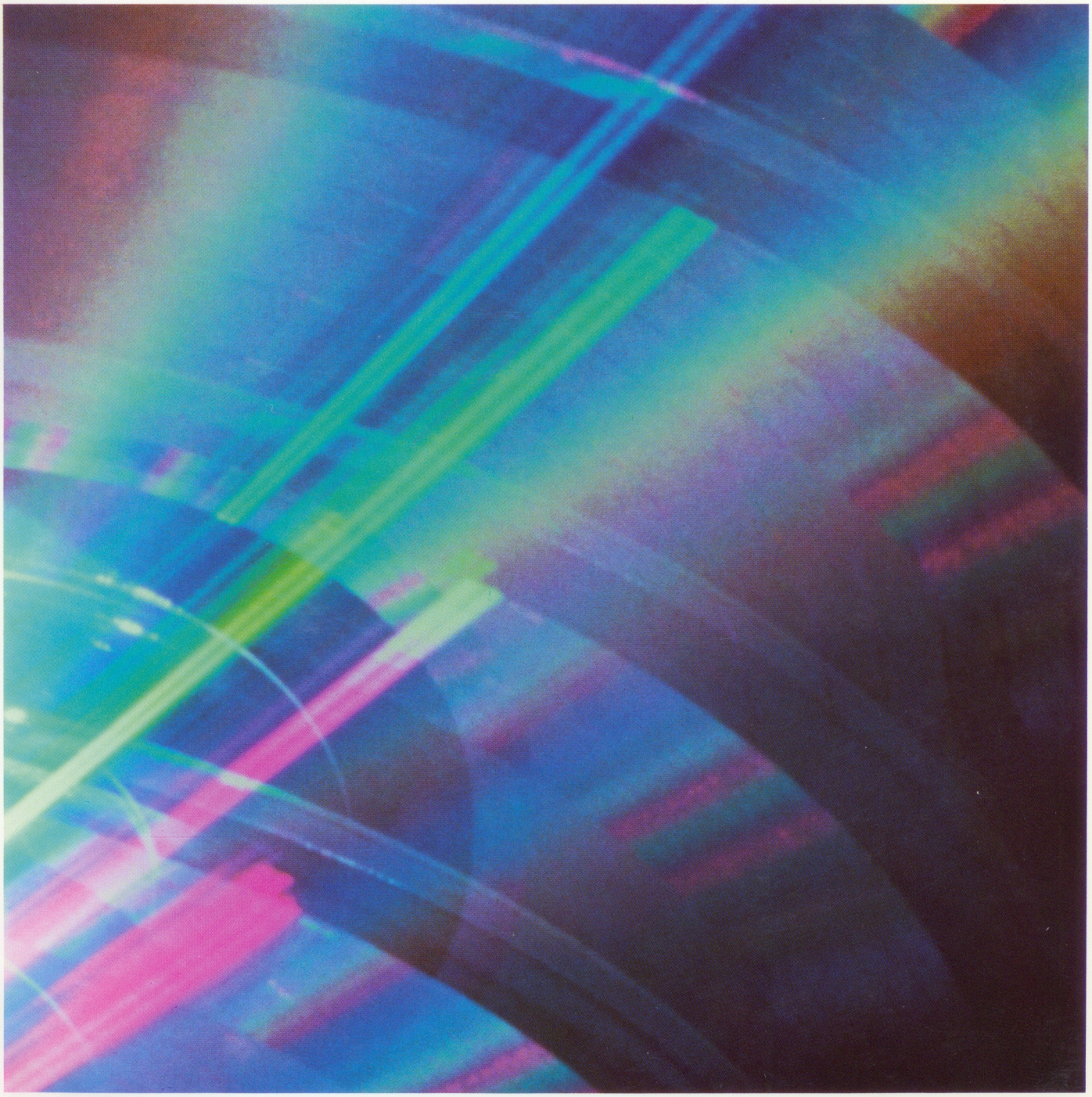
HEWLETT PACKARD

HP 3000/930 and HP 9000/840 Computers

Procedure Calling Conventions Manual



HP 3000/930 and HP 9000/840 Computers

Procedure Calling Conventions

Reference Manual

Manual Part Number: 09740-90015 Printed in U.S.A. November 1986

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

The software code printed alongside the date indicates the version level of the software product at the time the manual or update was issued. Many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition . . . . . Nov 1986

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the most recent version of each page in the manual. To verify that your manual contains the most current information, check the dates printed at the bottom of each page with those listed below. The date on the bottom of each page reflects the edition or subsequent update in which that page was printed.

Effective Pages	Date
all	Nov 1986

Contents

Chapter 8
Inter-Module Procedure Calls

Chapter 9
Millicode Calls

Chapter 10
Stack Unwinding

Appendix A
Code Examples

Appendix B
Summary of Assembler Procedure Control

1

Overview

Modern programming technique encourages programmers to practice well-structured decomposition, which entails the use of a greater number of smaller, more specialized procedures rather than larger, more complex routines. While this creates more adaptable and understandable programs, it also increases the frequency of procedure calls, thus making the efficiency of the procedure calling convention crucial to overall system performance.

Many modern machines provide instructions to perform many of the tasks necessary to make a procedure call, but this is not the case in the HP Precision Architecture (HPPA). Instead of using an architected mechanism, the procedure call is accomplished through a software convention which uses the machine's simple hardwired instructions, a solution that ultimately provides more flexibility and efficiency than the more complex (microcoded) instruction set additions.

Besides the obvious branch-and-return interruption that occurs in the flow of control as a result of a procedure call, many other provisions must be made in order to achieve an effective calling convention. The call mechanism must also pass parameters, save the caller's environment, and establish an environment for the called procedure (the "callee"). The procedure return mechanism must restore the caller's previous environment and save any return values.

Although the Precision Architecture machines are essentially register-based, by convention a stack is necessary for data storage. As a basis for discussion of the Procedure Calling Convention, we will first examine a straightforward calling mechanism in this environment, one in which the calling procedure (caller) acquires the responsibility for preserving its own state. This simplified model employs the following steps for each call:

NOTE

These steps are NOT the exact implementation used in the HPPA, but are given as a general basis for the discussion of the actual Procedure Calling Convention that will follow.)

- Save all registers whose contents must be preserved across the procedure call. This prevents the callee, which will also use and modify registers, from affecting the caller's state. On return, those register values are restored.
- Evaluate parameters in order and push them onto the stack. This makes them available to the callee, which, by convention, knows how to access them.
- Push a frame marker, which is a fixed-size area containing several pieces of information. Included is the static link, which provides information needed by the callee in order to address the local variables and parameters of the caller, as well as the return address of the caller.
- Branch to the entry point of the callee.

And to return from a call in this model, it is necessary that:

- The callee extract the return address from the frame marker and branch to it, and
- The caller then remove the parameters from the stack and restore all saved registers before the program flow continue.

This model correctly implements the basic steps needed to execute a procedure call, but is relatively expensive. The caller is forced to assume all responsibility for preserving its state, which is a conservative and safe approach, but causes an excessive number of register saves to occur. To optimize the program's execution, the compiler makes extensive use of registers to hold local variables and temporary values; these registers must all be saved at a procedure call and restored at the return. A high overhead is also incurred by the loading and storing of parameters and linkage information. The procedure call convention implemented in the Precision Architecture focuses on the need to reduce this expense by maximizing register usage and minimizing direct memory references.

The HPPA compilers attempt to alleviate this problem by introducing a procedure call mechanism that divides the register sets into "partitions". The registers are partitioned into "caller-saves" (the caller is responsible for saving and restoring them), "callee-saves" (the callee must save them at entry and restore them at exit), and "linkage" registers. In the general purpose register set, sixteen of the registers comprise the callee-saves partition and thirteen are available for use as caller-saves registers.

Thus the responsibility for saving registers is divided between the caller and the callee, and some registers are also available for linkage. The floating point registers and space registers are also partitioned in a similar manner.

The register allocator avoids unnecessary register saves by using caller-saves registers for values that need not be preserved across a call, while values that must be preserved are placed into registers from the callee-saves partition. At procedure entry, only those callee-saves registers used in the procedure are saved; this minimizes the number and frequency of register loads and stores during the course of a call. The register partitions are not inflexible; if more registers are needed from a particular partition than are available, registers can be borrowed from the other partition. The penalty for using these additional registers is that they must be saved and restored, but this overhead is incurred only in the special circumstance where excess registers are needed, which happens relatively infrequently.

In the simple model outlined above, all parameters are passed by being placed on the stack, which is expensive because direct memory references are needed in order to push each parameter. In the Precision Architecture procedure calling convention, this problem is lessened by the compilers, which allocate a permanent parameter area (in memory) large enough to hold the parameters for all calls performed by the procedure, and minimize memory references when storing parameters by using a combination of registers and memory to pass parameters. Four registers from the caller-saves partition are used to pass user parameters, each holding a single 32-bit value or half of a 64-bit value. Since procedures frequently have few parameters, the four registers are usually enough to accommodate them all. This removes the necessity of storing parameter values in the parameter area before the call. If more than four 32-bit parameters are passed, the additional ones are stored in the preallocated parameter area, or if a parameter is larger than 64 bits, its address is passed and the callee copies it to a temporary area.

Additional savings on memory access are gained when the callee is a leaf procedure (one that does not make any other calls). In this situation, the register allocator uses the caller-saves registers to hold variable values, thus eliminating the need to save callee-saves registers that it might have used in a non-leaf procedure. Furthermore, since a leaf procedure will not make subsequent procedure calls, there is no need to allocate a stack frame for it, because the return address and other values can remain in registers during the entire life of the call. (Actually, there are rare exceptions to this; a stack frame may be necessary for a leaf procedure if more local space is needed than is available in registers.)

Types of Procedure Calls

Procedure calls can be grouped into three categories, depending on the location of the callee. The possibilities are:

1. Procedures residing in the same load module (Intra-Module Call)
2. Procedures residing in other load modules (Inter-Module Call)
3. Operating System or Subsystem procedures

NOTE

Throughout the rest of this document, "Local" is used as a synonym for "Intra-Module" and "External" is used interchangeably with "Inter-Module" or "OS/Subsystem".

In order to simplify code generation, all three types of calls are mapped into the local (intra-module) case, thus requiring the compiler to recognize only a single type of call. This is accomplished through the use of two types of "stubs" ("calling stubs" and "called stubs") which establish the external branching and linking necessary for inter-module calls. Inter-module calls and specific stub usages are discussed in more detail in section 8 ("Inter-Module Calls").

2

Interfaces

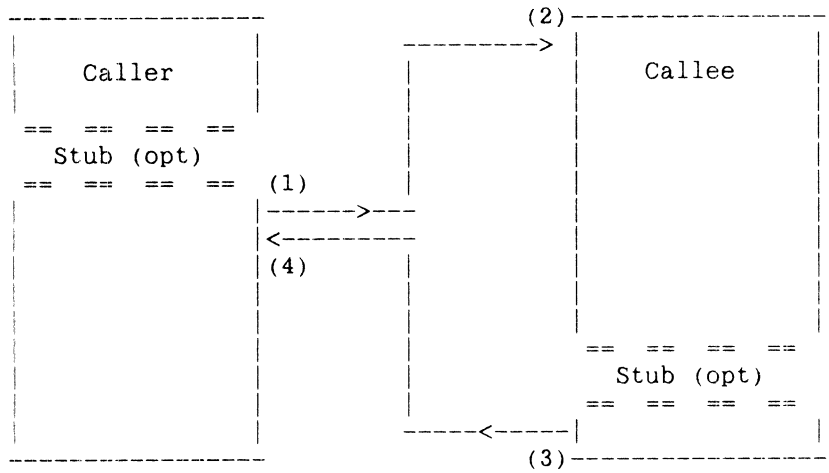


Figure 2-1. Procedure Call Convention Architected Interfaces.

Figure 2-1 shows the units involved in a procedure call, and the critical interfaces (numbered) that will be discussed throughout this document, and their definitions follow:

CALLER : The calling code, origin of the call.
CALLEE : The calling code, object of the call.

STUB: A piece of code that may be attached to the caller and/or callee that enables an otherwise impossible call to occur. Stubs are used primarily with Inter-Module calls. See Section 8 for details.

(1) Call : The transfer of program control to the callee.

(2) Entry : The point of entry into the callee.

(3) Exit : The point of exit from the callee.

(4) Return : The point of return of program control to the caller.

3

Stack Usage

Leaf / Non-Leaf Procedures

All procedures can be classified in one of two categories: leaf or non-leaf. For the purpose of definition, a leaf procedure is one that makes no additional calls, while a non-leaf procedure is one that does make additional routine calls. Although simple, the distinction is essential because the two cases entail considerably different requirements regarding (among other things) stack allocation and usage. Every non-leaf procedure requires the allocation of an additional stack frame in order to preserve necessary execution values and arguments, while this is not necessary for a leaf procedure. The recognition of a procedure as fitting into either the leaf or non-leaf category, as well as the allocation of all necessary stack space, is done at compile time. As will become evident throughout this document, it is often the case that, due to these stack allocation conventions, much of a callee's state information is saved in the caller's frame; this is one way in which unnecessary stack usage can be avoided. A general picture of the top of the stack for one call, including the frames belonging to the caller (previous) and callee (new), is shown in Figure 3-1.

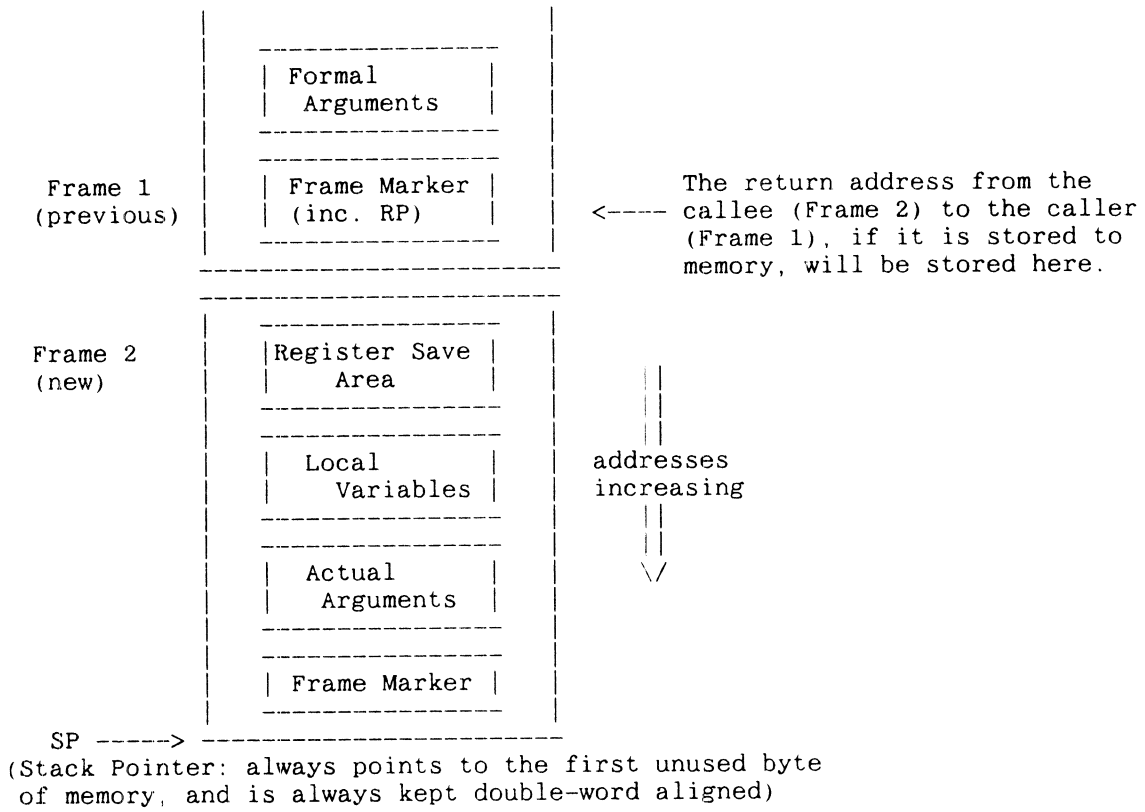


Figure 3-1. General Stack Layout.

Storage Areas Required for Call

The elements of a single stack frame that must be present in order for a procedure call to occur are shown below. The stack addresses are all given as byte offsets from the actual SP (stack pointer) value; for example, 'SP-36' designates the address 36 bytes below the current SP value.

NOTE

Fields shown in Table 3-1 are explained in the text following the table.

Table 3-1. Elements of Single Stack Frame Necessary for Procedure Call.

Offset	Contents	
Variable Arguments (optional; any number may be allocated)		
SP-(4*(N+9))	arg word N	
:	:	
:	:	
SP-56	arg word 5	
SP-52	arg word 4	
Fixed Arguments (must be allocated; may remain unused)		
SP-48	arg word 3	
SP-44	arg word 2	
SP-40	arg word 1	
SP-36	arg word 0	
Frame Marker		
SP-32	External Data Pointer (DP)	(set after Call)
SP-28	External sr4	(set after Call)
SP-24	External/stub RP (RP')	(set after Call)
SP-20	Current RP	(set after Entry)
SP-16	Static Link	(set before Call)
SP-12	Clean Up	(set before Call)
SP- 8	Calling Stub RP (RP'')	(set after Call)
SP- 4	Previous SP	(set before Call)
Top of Frame		
SP- 0	Stack Pointer (points to next available address)	
	< top of frame >	

Frame Marker Area

This eight-word area is allocated by any non-leaf routine prior to a call. The exact size of this area is defined because the caller uses it to locate the formal arguments from the previous frame. (Any standard procedure can identify the bottom of its own frame, and can therefore identify the formal arguments in the previous frame, because they will always reside in the region beginning with the ninth word below the top of the previous frame.)

Previous SP: contains the old (procedure entry) value of the Stack Pointer. It is only required that this word be set if the current frame is noncontiguous with the previous frame. Calling (a.k.a. Import) Stub RP (RP''): Reserved for use by a calling stub that must store an RP value, so the stub can be executed after the exit from the callee, but before return to the caller. See Section 8 for detailed discussion of stubs. Clean Up: area reserved for use by language processors; possibly for a pointer to any extra information (i.e. on the heap) that may otherwise be lost in the event of an abnormal

interrupt. Static Link: Used to communicate static scoping information to the callee that is necessary for data access. It may also be used in conjunction with the SL register, or to pass a display pointer rather than a static link, or it may remain unused. Current RP: reserved for use by the called procedure; this is where the current return address can be stored if the procedure wants/needs to use RP (gr2) for any other purpose.

External/Stub RP (RP'),
External sr4, and
External DP : all three of these words are reserved for use by the intermodular (external) calling mechanism. See Section 8 for more details.

Fixed Arguments Area

These four words are reserved for holding the argument registers, should the callee wish to store them back to memory so that they will be contiguous with the memory-based parameters. All four words must be allocated for a non-leaf routine, but may remain unused.

Variable Arguments Area

These words are reserved to hold any arguments that can not be contained in the four argument registers. Although only a few words are shown in this area in the diagram, there may actually be an unlimited number of arguments stored on the stack, continuing downward in succession (with addresses that correspond to the expression given in the diagram.) Any necessary allocation in this area must be made by the caller.

4

Register Usage

Register Partitioning

In order to provide adequate register resources for normal usage, the general and floating point register sets have been divided into partitions, with each set consisting of (among other fixed values) groups designated as Callee-Saves and Caller-Saves. Callee-Saves registers are those that the callee saves immediately after procedure entry, if necessary, and restores before exit, and are guaranteed to be preserved across procedure calls. Caller-Saves registers are those that the caller will (if necessary) save before call and restore after return, and are NOT guaranteed to be preserved across calls.

The Caller-Saves registers may be used as temporaries which need to be saved only if they contain values which are live across a call. The Callee-Saves registers may be actually used by the callee only if it insures the restoration of their original entry values before exit.

These partitions (along with other dedicated registers) are shown in Figure 4-1 on the following page.

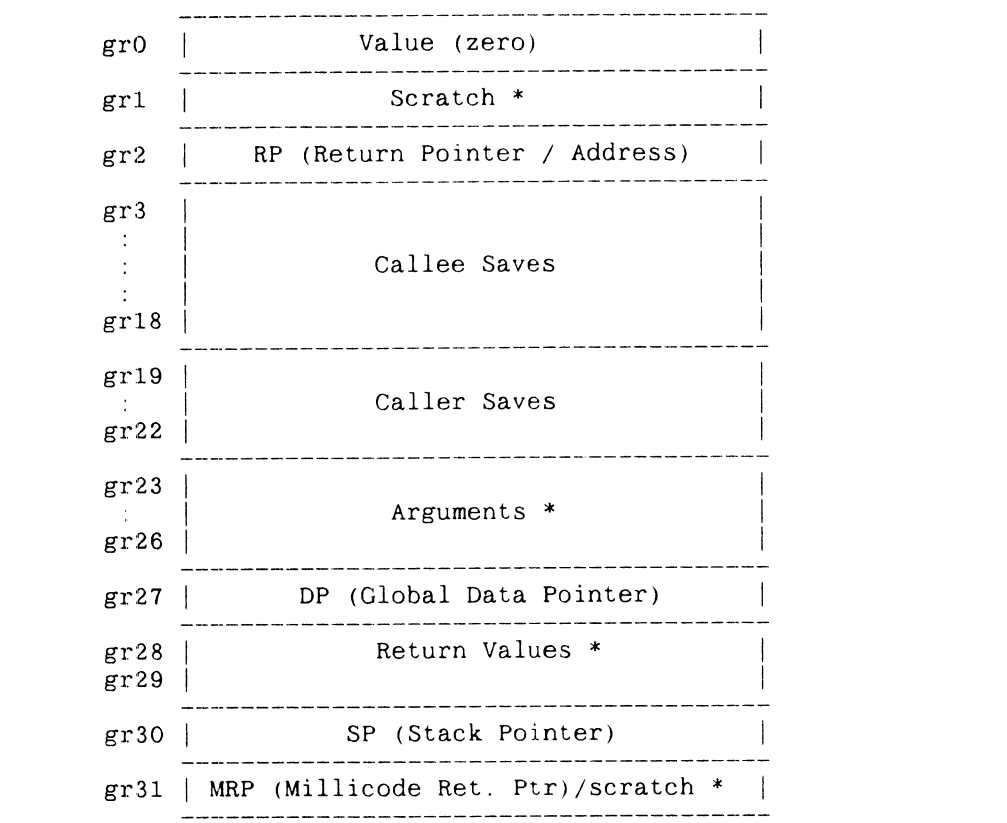


Figure 4-1. Register Partitioning.

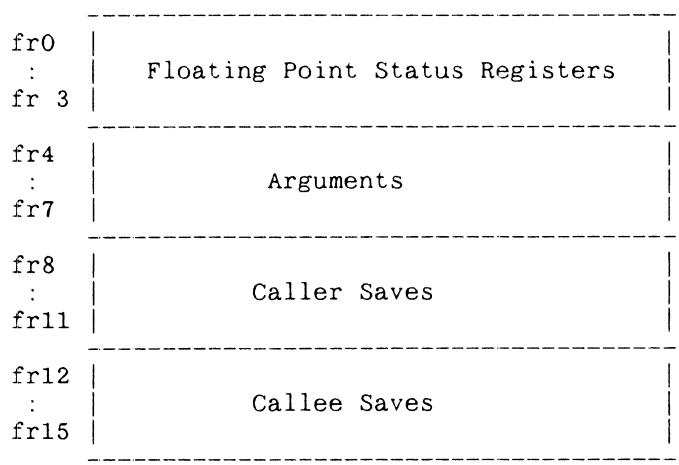


Figure 4-2. Floating Point Registers.

Other Register Conventions

The following are guaranteed to be preserved across calls:

1. The procedure entry value of SP.
2. The values of DP, and sr4-7.
3. The value of fr0, unless it is explicitly modified by the callee.

The following are NOT preserved across calls:

1. Any caller-saves registers that are not saved and restored as part of the caller-saves partition.
2. Floating Point registers 1-3 (fr1, fr2, fr3).
3. Space registers 0-2 (sr0, sr1, sr2).
4. The PSW (Processor Status Word). See Section 2, of the Processor ACD, for further explanation).
5. Any control registers that are modified by privileged software. (e.g. Protection IDs).
6. The state (including internal registers) of any special function unit. (e.g. floating point coprocessor).

Return Values

Values are returned from procedures/functions as listed in Table 4-1.

Table 4-1. Return Values.

Type of Return Value	Return Register
Nonfloating Pt. (32-bit)	ret0 (gr28)
Nonfloating Pt. (64-bit)	ret0 (gr28) - high order word
	ret1 (gr29) - low order word
Floating Pt. (32-bit)	fret (fr4) *
Floating Pt. (64-bit)	fret (fr4) *
Space Identifier (32-bit)	sret (sr1)
Any > 64 bits	short pointer in ret0 (gr28) will point to the value (must assume worst alignment)

* Although not common, it is possible to return floating point values in general registers, as long as the argument relocation bits in the symbol record are set correctly. (Refer to Section 6,'Parameter Relocation',for more details.)

Summary of Dedicated Register Usage

The following table shows the required conventions regarding argument and return value passing. Note that those registers listed in the first section below are the only ones that absolutely cannot be used to hold user values (i.e. arguments), but the user risks the unintentional destruction of necessary data if he does not adhere to the standard conventions.

Table 4-2. Dedicated Register Usage.

```
Registers that cannot be used to hold user values:
--------------------------------------------------

gr0       Zero Value Register (cannot be modified)
gr2    ** RP (return pointer/address for leaf routines)
gr27      DP (global data pointer)
gr30      SP (stack pointer)


Other registers' conventional usage:
------------------------------------

gr26      arg0  (argument register 1)
gr25      arg1  (   "         "     2)
gr24      arg2  (   "         "     3)
gr23      arg3  (   "         "     4)


gr28      ret0  (function return register - at Exit and Return;
                   OR pointer - at Call and Entry)
gr29      SL    (static link register - at Call and Entry) OR
          ret1  (function return register when
                 returning 64-bit values - at Exit and Return)


fr4       fret  (function return register - at Return) OR
          farg0 (floating point argument register 1 - at Call)
fr5       farg1 (   "         "       "          "     2)
fr6       farg2 (   "         "       "          "     3)
fr7       farg3 (   "         "       "          "     4)


sr1       sarg  (argument register - at Call) OR
          sret  (function return register - at Exit)
```

NOTE

If the routine in question is a non-leaf routine, gr2 must be stored because of subsequent calls; hence it is then available to be used as a scratch register by the code generators. Also, although common, it is not absolutely necessary that gr2 be restored before exit; a branch (BV) through another register (e.g. gr19) would be acceptable.Type your own text here, then cut this text.

5

Parameter Passing

Value Parameters

Value parameters are mapped to arg words in the argument list with successive parameters mapping to successive argument words, except 64-bit parameters, which must be aligned on 64-bit boundaries. Irregularly sized data items should be extended to 32 or 64 bits. (The practice that has been adopted is to right-justify the value itself, and then left-extend it.) Non-standard length parameters that are signed integers are sign-extended to the left to 32 or 64 bits. This convention does not specify how 1-31, 33-63-bit data items are passed by value (except single ASCII characters).

Table 5-1 lists the sizes for recognized inter-language parameter data types. The form column indicates which of the forms (space ID, nonfloating point, floating point, or any) the data type is considered to be.

Inter-Language Parameter Data Types and Sizes

Table 5-1. Parameter Data Types and Sizes.

Type	Size (bits)	Form
ASCII character (in low order 8 bits)	32	Nonfloating Pt.
Integer	32	Nonfloating Pt. or Space ID
Short Pointer	32	Nonfloating Pt.
Long Pointer	64	Nonfloating Pt.
Routine Reference (see below for details of Routine Reference)	32	Routine Reference
Long Integer	64	Nonfloating Pt.
Real	32	Floating Pt.
Long Real	64	Floating Pt.
Quad Precision	128	Any

Space Identifier (SID) (32 Bits): One arg word, callee cannot assume a valid SID. **

Nonfloating Point (32 Bits): One arg word.

Nonfloating Point (64 Bits): Two words, double word aligned, high order word in an odd arg word. This may create a void in the argument list (i.e. an unused register and/or an unused word on the stack.)

Floating Point (32 Bits): One word, callee cannot assume a valid floating point number. **

Floating Point (64 Bits): Two words, double word aligned (high order word in odd arg word). Callee cannot assume a valid floating point number.** This may create a void in the argument list.

Any >64 Bits: A short pointer (in sr5 - sr7) to the data item value is passed as a non-floating point 32-bit value parameter. The callee must copy the accessed portion of the value parameter into a temporary register before any modification can be made to the (caller's) data.

NOTE

The point is made that the callee "cannot assume a valid" value in these cases because no specifications are made in this convention that would insure such validity. It therefore becomes mainly the responsibility of the caller to supply valid values, so that the callee does not have to perform constant checking on all floating point values and SIDs that are passed.

Reference Parameters

A short pointer to the referenced data item (in sr4-sr7) is passed as a nonfloating point 32-bit value parameter (alignment is not guaranteed). Note: sr4 can only be used if the call is known to be local, because the external calling convention changes sr4. See Section 8 ("External Calls") for further details.

Value-Result and Result Parameters

It is intended that language processors can use either the reference or value parameter mechanisms for value-result and result parameters.

Routine References

This convention requires that routine references (i.e. procedure parameters, function pointers, external subroutines) be passed as 32-bit nonfloating point values.

It is expected that language processors that require a static link to be passed with a routine reference (i.e. Pascal passing level 2 procedures) will pass that static link as a separate 32-bit nonfloating point value parameter. A language processor is free to maximize the efficiency of static scope linking within the requirements, without impacting other language processors.

See Section 8 for further details on Routine References.

Argument Register Usage Conventions

Parameters to routines are logically located in the argument list. At the Call interface, the first four words of the argument list are passed in registers, depending on the

usage and number of the argument. The first four words of the argument list are then reserved as spill locations for the stack-based argument registers. These requirements imply that the minimum argument list size is 16 bytes; this space must be allocated in the frame, but may not actually be utilized.

The standard register use conventions are shown in Table 5-2.

Table 5-2. Argument Register Use.

	void	SID	nonFP	FP32	FP64
arg word 3	no reg	arg3	arg3	farg3	farg3 {0..31}
arg word 2	no reg	arg2	arg2	farg2	farg3 {32..63}
arg word 1	no reg	arg1	arg1	farg1	farg1 {0..31}
arg word 0	no reg	sarg	arg0	farg0	farg1 {32..63}

definitions:

```
void   - arg word not used in this call
SID    - space identifier value
nonFP  - any 32-bit or 64-bit nonfloating point
FP32   - 32-bit floating point
FP64   - 64-bit floating point
```

Parameter Type Checking

The compilers generate descriptors for every parameter and argument value which contain information defining the type of the value. These descriptors ("symbol descriptors" and "argument descriptors") are then checked by the linker for compatibility; if they do not match, a warning is generated. See the Spectrum Object File Format File Format ACD for further details on these descriptors.

6

Parameter Relocation

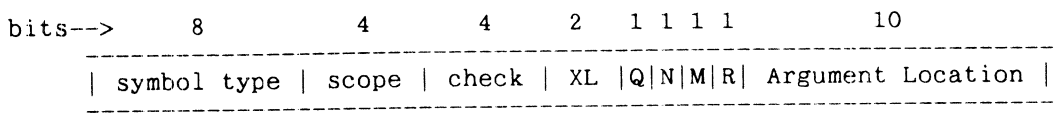
NOTE

Parameter relocation is currently (first release) implemented only by the linker and assembler, and therefore is not used, although it is likely that it will be fully implemented and utilized in the future.

The procedure calling convention specifies that the first four words of the argument list and the function return value will be passed in registers: floating point for real values, general otherwise. However, it would be advantageous to provide an exception to this rule in order to support languages that do not require type checking of parameters, which can lead to situations where the caller and the callee do not agree on the location of the parameters. Problems such as this occur frequently in the C language, where, for example, formal and actual parameter types may be unmatched, due to the fact that no type checking occurs. A parameter relocation mechanism alleviates this problem, and all parameter type checking becomes language-dependent.

The proposed solution to this problem entails the inclusion of an additional code sequence, called a relocation stub, which is inserted between the caller and the callee. When executed, the relocation stub moves any incorrectly located parameters to their expected location. If a procedure is called with more than one calling sequence, a relocation stub is needed for each non-matching calling sequence.

In order for the compiler to communicate the location of the first four words of the parameter list and the location of the function return value to the linker and loader, ten bits of argument location information have been added to the definitions of a symbol and a fix-up request (as defined in the Spectrum Object File Format ACD). The following diagram shows the first word of a symbol definition containing the relocation information. (See section 2.8 of the aforementioned ACD for further details.)



The ten bits of argument location information are further broken down into five location values, corresponding to the argument function return values, as shown below:

```
Bits 22-23 : define the location of parameter list word 0
Bits 24-25 : define the location of parameter list word 1
Bits 26-27 : define the location of parameter list word 2
Bits 28-29 : define the location of parameter list word 3
Bits 30-31 : define the location of the function value return
```

The value of an argument location is interpreted as follows:

```
  00              Do not relocate
  01   arg        Argument register
* 10   fr         Floating point register (bits 0..31)
* 11   frupper    Floating point register (bits 32..63)
```

* For return values, '10' means a single precision floating point value, and '11' means double precision floating point value.

When the linker links a procedure call, it will generate a relocation stub if the argument location bits of the fixup request do not exactly match the relocation bits of the exported symbol, with the exception of the case where either the caller or callee specifies "do not relocate". The relocation stub will essentially be part of the called procedure, and the linker can optionally add a symbol record for the stub so that it can be reused. The symbol record will be the same as the original export symbol record, except that the relocation bits will reflect the input of the stub, the type would be STUB and the symbol value will be the location of the relocation stub

The execution of a relocation stub can be separated into the call path and the return path. During the call path, only the first four words of the parameter list will be relocated, while only the function return will be relocated during the return path. The control flow is shown in Figure 6–1.

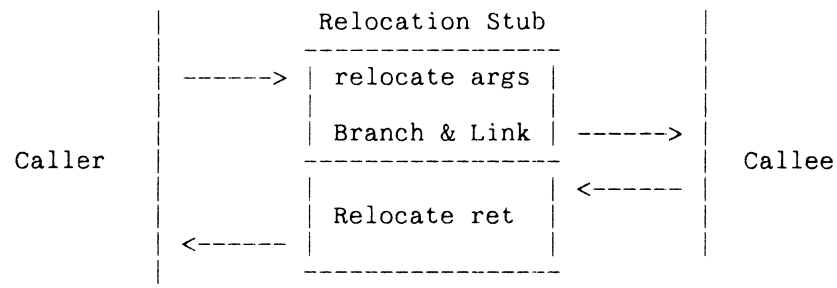


Figure 6–1. Parameter Relocation Stub.

If the function return does not need to be relocated, the return path can be omitted and the branch and link will be changed to a branch. The call path must always be executed, but if the first four words of the parameter list do not need to be relocated, it can be reduced to the code required to establish the return path (i.e save RP and branch and link to the callee).

When multiple stubs occur during a single call (e.g. calling stub and relocation stub), the stubs can be cascaded (i.e. used sequentially); in such a case, both RP' and RP'' would be used. (The relocation stub uses RP''.)

When the linker makes an object module executable, it will generate stubs for each procedure that can be called from another object module (i.e. called dynamically). In addition, a stub will be required for each possible calling sequence. Each of these stubs will contain the code for both relocation and external return, and will be required to contain a symbol definition record.

Both calling and called stubs use a standard interface: calling stubs always relocate arguments to general registers, and called stubs always assume general registers.

In order to optimize stub generation, the compilers should maximize the use of the argument location value 00 (do not relocate). A linker option may be provided that would allow the user to turn stub generation on or off, depending on known conditions. Also, a linker option is provided to allow the user to inhibit the generation of stubs for run-time linking. In this case, if a mismatch occurs, it would be treated as a parameter type checking error (which is totally independent of parameter relocation).

7

The Actual Call

Control Flow of a Standard (Local) Procedure Call

Figure 7-1 shows the progression of a standard procedure call. To summarize, the steps involved are:

1. << previously executing caller's code >>
2. (before call) Pass arguments (put into registers and onto stack as necessary) and save caller-saves registers.
3. (call) Branch (BL) to callee.
4. (after entry) -if it is a non-leaf procedure- Save RP, allocate local frame, and save callee-saves registers.
5. << execute body of callee >>
6. (before exit) Restore RP, restore callee-saves registers, and deallocate local frame.
7. (exit) Branch (BV) back to caller.
8. (after return) Restore caller-saves registers.
9. << resume execution of caller's code >>

NOTE

To read diagram, begin at "Calling Code" and continue downward, following any arrows that may extend from the end of a line.

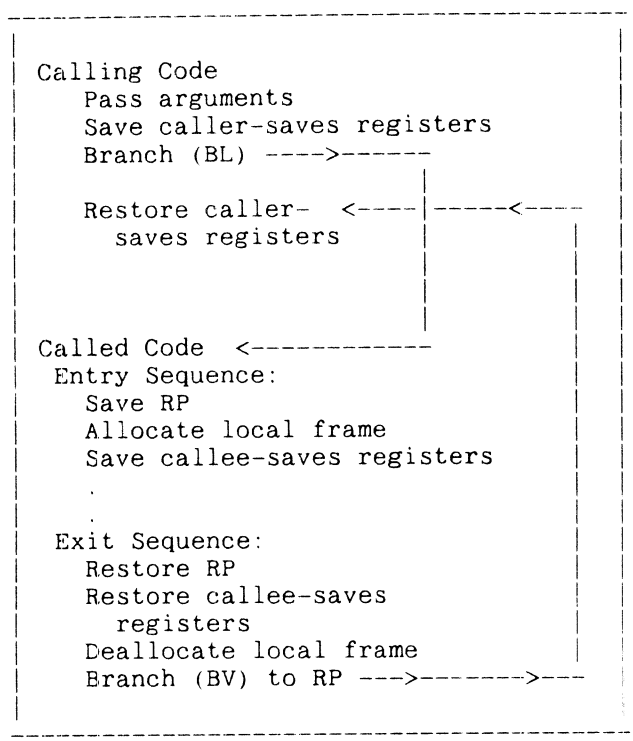


Figure 7-1. Control Flow of a Standard Procedure Call.

Efficiency

The following factors greatly reduce the overhead expense involved in a Precision Architecture procedure call:

1. Allocation of the stack frame and saving of one callee-saves register can be accomplished in a total of one cycle. (The same is also true for the deallocation of the frame and restoration of one register.)
2. Even when not optimizing, the delay slot (i.e. the instruction slot following a Branch instruction) is used for operations such as loading register values and passing parameters. This instruction slot is initially filled by the compiler with a NOP, and later replaced by the optimizer whenever possible in order to avoid wasting a machine cycle.
3. Most of the steps involved in a call are optional if they are unnecessary in a particular situation. (For example, saving and restoring RP, saving and restoring callee-saves registers, allocating stack storage space, etc.)

The Code Involved in a Simple Local Call

A simple example is shown below; the procedure and the call to it are shown first in C (source code) and then in assembly. (NOTE: A more detailed, fully documented example is shown in Appendix A (A.1.).

```
In C:   :
        :
        proc (50,100);      \* call to procedure 'proc'  *\
        :
        :
        proc (x,y)          \* function 'proc'; returns  *\
        int x,y;            \* sum of two numbers        *\
            {
             return x+y;
            }
        :
        :

In assembly:
        :
        LDI   50, gr26       ; load arg word 0 into gr26
     ** BL    proc, gr2      ; branch (call) to 'proc'
        LDI   100, gr25      ; <delay slot> load arg word 1 into gr25
        :
        :
 proc:  BV    0 (gr2)          ; branch (return) back to caller
        ADD   gr26,gr25,gr28   ; <delay slot> add arguments, put result into gr28
        :
```

** In instances where the target of the call is out of the range of the BL (i.e. greater than 256K bytes), there will be a slightly different code sequence. If this situation is identified at compile time, the compiler will generate a two-instruction sequence (LDIL,BLE) that is guaranteed to reach the target, instead of the BL. However, if the information is not known until link time (the compiler has already generated the BL), the BL will be linked to a "long call stub" rather than to the actual procedure. This "stub" is merely the same two-instruction sequence that would have been used if the information had been known earlier.

8

Inter-Module Procedure Calls

Introduction

NOTE

At present (first release), external calls exist ONLY on the MPE-XL operating system; this is due to the fact that on the HP-UX OS, all load modules are combined into a single executable file with no unresolved references. Hence, this section does not apply to HP-UX.

As mentioned earlier, there are three types of procedure calls that can be executed, and they can be classified into two groups: intra-module (local) and inter-module (external) calls. Basically, a local call is one in which the caller and the callee reside in the same load module, and an external call is one in which that is not necessarily the case. There is one exception to this guideline, however: calls which cross privilege level boundaries are always treated as external calls, even if the caller and callee are residing in the same load module. Although external calls are closely related to local calls, several notable differences exist in storage and access conventions; these are explained in the following material.

The inter-module (external) calling sequence is distinguished from the intra-module (local) calling sequence due to the system-wide global virtual addressing of the Precision Architecture processor and the need for code sharing. Unlike most conventional systems where each process has a private virtual address space, all processes in a Precision Architecture system share a finite, although large, number of spaces (2^16 currently, 2^32 eventually). Therefore, it is undesirable to assign virtual spaces to inactive program images (i.e. those on disk). Generally, the assignments of virtual spaces to a program will be delayed until it is activated.

In order to avoid extensive linking at process creation time, it is desirable to have a central data structure (for each process) that will hold the SIDs assigned to the load modules used by the process. Consequently, only the entries in this structure will be updated when virtual spaces are finally assigned to load modules.

Another use for such a central structure is the location of the global data area of a load module. The offset of the global data area in the private data space of a process depends on the combination of load modules called by the process. Although each load module's code is assigned a unique SID, its data is placed at a process-dependent offset within the process' single private data space. In order to share a load module between multiple processes, all references to the global data area must be relative to a base register, which in this case is DP (data pointer). The value of a base register can be stored in a load module's entry in the central structure. The central structure employed in this capacity is the Inter-Module Cross Reference Table (XRT), which will be discussed and diagrammed in detail below.

In terms of the code, an external call uses the same code sequence as a local call, with the addition of a "calling stub" (a.k.a. Import Stub) attached to the calling code, and a "called stub" (a.k.a. Export Stub) attached to the called code. Unlike a local call, execution is not transferred directly from the caller to the callee; in an external call, a millicode sequence (CALLX) is employed to locate and branch to the target procedure. These stubs and the millicode sequence are discussed in greater detail below.

Requirements of an External Call

An external procedure call requires several extra steps in addition to those necessary for a local call, which are outlined below:

1. The base register pointing to the global data area (DP), the SID contained in sr4, and value in gr2 (RP) must all be saved in the caller's allocated frame area. This RP value is referred to as RP' to distinguish it from the usual RP value associated with a local call. The difference between RP and RP' is as follows:

   RP is renamed RP' during the calling stub execution, at which time RP becomes the location in the called stub to which the target procedure (callee) must return. Finally, RP' is renamed back to RP, and DP and the SID are restored.

2. The called load module's entry in the XRT must be located.

3. The short pointer from the XRT entry must be loaded into DP (this is the DP value for the called load module).

4. The SID from the XRT entry must be loaded into sr4, and the offset of the callee must be obtained from the XRT.

5. An external branch and link to the called procedure must be performed. (Actually, an external branch to the CALLX millicode routine that locates the called stub and a branch to the called stub of the callee, which then does a local branch and link to the callee. Stubs, CALLX and linking will be discussed further in sections 8.7 - 8.9.)

Requirements of an External Return

In order to return from an external call, it is required that DP, sr4, and RP be restored. The saved value of RP' will be loaded into RP (gr2), and the return to the caller will be via that register. The DP and SID of the caller will be restored from the caller-save area.

Control Flow of an External Call

Figure 8-1 shows a simplified external procedure call. It uses the same code sequence as the local, but a "calling stub" is attached to the calling code (the caller) and a "called stub" is attached to the entry point of the called code (the callee). Execution is transferred from the calling code to the called code by executing a millicode sequence (CALLX) which uses an XRT to locate and branch to the target procedure. Note: All of these elements of an external call are covered in more detail in the following subsections.

NOTE

To read diagram, follow the arrows and numbers, beginning with number 1.

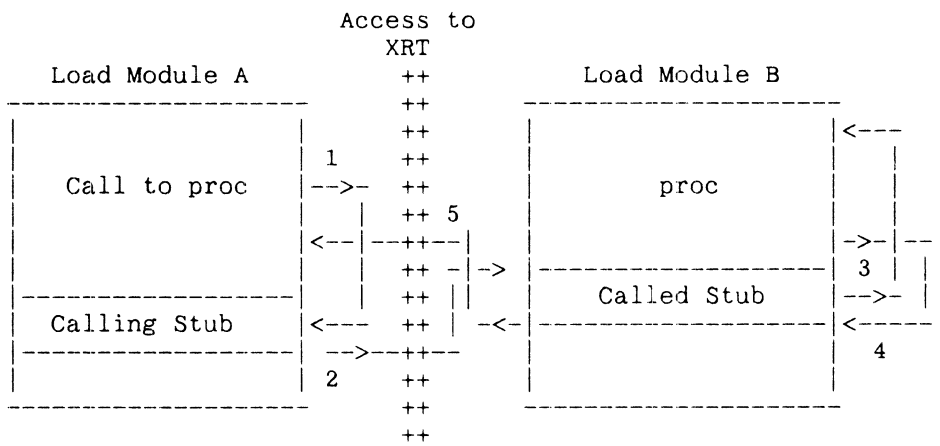


Figure 8-1. Simplified External Procedure Call.

Calling Code

The calling code generated by the compiler to perform a procedure call will be the same regardless of whether the call is actually local or external. If the linker locates

the procedure being called within the current executable object module, it will make the call local by patching the BL instruction to directly reference the entry point of the procedure. If the linker is unable to locate the procedure, it will make the call external by attaching a calling stub to the calling code, and patch the BL instruction to branch to the stub.

Before the call, the calling code must save any caller-saves registers that contain active data. The parameter list for the callee is stored in the current stack frame between the register spill area and the frame marker. As in a local call, the parameter list is stored in reverse order, such that the first word is at SP-36, the second is at SP-40, etc. Note: the first four words of the parameter list are passed in registers, but the space for the argument list is allocated, even if it remains unused. Also, all parameters are always passed in general registers, with the linker including any necessary relocation information in the stubs.

Called Code

The called code is responsible for allocating a new stack frame on the top of the stack (the frame must be double-word aligned); the actual size of the frame will be determined by the compiler, and will be the summation of:

- The amount of space needed by the register allocator for the register spill area;
- The amount of space needed for the local variables of the current procedure;
- The amount of space needed to store the longest parameter list of any procedure called by this procedure; and
- The frame marker has 32 bytes.

If this procedure is callable by a less-privileged procedure, each page of the stack frame must be PROBEd (a privilege-checking mechanism) before any information is stored into the frame. The PROBE instructions must be generated by the compiler. (This is not currently implemented).

When a procedure is entered, gr2 (RP) will contain the offset portion of the return address. Whether the procedure was called locally or externally, the two low order bits of RP will always contain the execution level of the caller. From the source code level it is difficult, if not impossible, to determine if RP is valid or if it has been stored into the stack frame. Therefore, compilers that support multiple privilege levels will need to provide a mechanism for returning the execution level of the caller.

If the current procedure calls another procedure, RP must be saved sometime before the call, probably in the procedure entry sequence. The called code is responsible for insuring the validity of its own input. sr5, sr6, and sr7 can only be set by privileged code and therefore can be assumed to be correct at all times. In addition, DP, LP (linkage pointer), and sr4 are not changed during a local call, or they are set by the procedure call millicode during an external call and therefore may also be assumed to be correct. The value of SP and the parameters passed must be validated by the called code.

NOTE

Only those fields necessary in the frame marker of the current procedure will contain valid data; others will be undefined. For example, during a local call, contents of the external return link pointer field will be undefined.

Outbound/Inbound Transfer Code Stubs

As previously mentioned, the compiler only recognizes a single type of procedure call (local), a characteristic that is made possible by the use of stubs. Stubs are pieces of code that are attached to the caller and/or callee that enable the original calling and called code to remain unchanged through the external call process. There are two types of stubs used in this procedure calling convention: Outbound (calling) and Inbound (called). These are defined here, and explained in detail in the following sections.

1. Outbound Transfer Code Stub (Calling Stub) -- a locally-linked stub that enables inter-module and OS/Subsystem calls to appear (to the compiler) as local calls. If the linker determines that a call is external, it will attach a calling stub to the procedure and patch the BL (Branch and Link) instruction to branch to the stub. There is usually one calling stub for each procedure referenced in the module (which can accommodate all calls to a specific procedure), but it is possible to have a separate stub for each CALL to a procedure.

2. Inbound Transfer Code Stub (Called Stub) -- enables a called routine to avoid the problem of having multiple return sequences (i.e. different for local and external calls). There is one called stub for each external procedure of a load module. Inter-module calls will enter the called stub, which in turn will enter the called procedure (callee). The callee can thus return to its called stub (which is local) rather than being concerned with the external return. The calling stubs can be generated by the linker (or obtained from a "stub library") and then linked to their respective routines.

NOTE

As earlier mentioned, calling and called stubs are sometimes referenced (in other documents) as Import and Export stubs. There are no different implications attached to either naming convention; calling and called have been used in this document in order to (hopefully) improve understandability.

Calling Stub

The calling stub will load gr1 with a pointer to the procedure XRT table entry (XRT pointer) of the called procedure and then branch to an external procedure call millicode sequence. Since the location of the XRT for an object module may be different for separate executions of the object module, the XRT entry pointer will be computed in the calling stub. The XRT entry pointer is computed by adding the XRT entry offset to the value of the Linkage Pointer (LP), which is stored at DP-4, pointing to the base of the XRT for the current object module.

For permanently bound calls to the operating system, a calling stub is not necessary; instead, the BL instruction in the call is replaced with a BLE instruction that branches to a system entry point branch table. This eliminates much of the linking that is normally performed when an object module is loaded.

Although the external procedure call diagram shows that DP, RP', and sr4 are saved by the CALLX millicode (see next section for discussion of CALLX), DP and RP' will actually be saved in the calling stub, and sr4 will by copied to a general register in the calling stub. This is done to eliminate two interlocks and fill a branch delay slot that would otherwise be left unused. The code sequence of the calling stub will be similar to that shown below:

```
LDW          -4(DP),gr1                ; Load LP
STW          DP,-32(SP)                ; Save DP
ADDIL *      L'XRToff,gr1              ; Add XRT offset to LP
LDO   *      R'XRToff(gr1),gr1         ;
LDW          16(gr1),gr20              ; Load address of CALLX
STW          RP,-24(SP)                ; Save RP'
BE           (sr7,gr20)                ; Branch to CALLX
MFSP         sr4,gr21                  ; Move sr4 to gr21
```

```
* Can be eliminated in cases where they would effectively be NOPs and
  therefore removed by the linker (except in cases where tools are
  being used that assume fixed-length stubs).
```

External Procedure Call Millicode (CALLX)

The CALLX millicode sequence is primarily a transition mechanism that facilitates the successful location and access of the desired external routine. The address of the CALLX routine is obtained from the XRT entry, and is assigned by the loader. Several variations of CALLX are available, depending on the possible privilege promotions. It is called from the calling stub, and operates as follows:

1. Saves DP, RP', and sr4 (if necessary).
2. Alters the privilege level if necessary (Gateway).
3. Checks the XRT pointer to insure that it points to a valid XRT table entry.
4. Loads the LP, DP, Offset and sr4 (of new procedure) values.
5. Branches to the called stub in the external module.

Called Stub

A called stub is used to enable the compiler to generate the same exit code sequence whether the procedure will be called locally, externally or both. If the linker determines that a procedure can be called from another object module, it will attach a called stub to the procedure. The stub will be the external entry and return point for the procedure. Local calls to the procedure will be unaffected.

Although the stub is the external entry point, its primary purpose is to be executed during an external procedure call exit/return. The stub is entered before the procedure so that RP can be set to the address of the stub, which will cause the local return in the procedure to exit to the stub. When the stub is executed during the return, it will restore DP and sr4, and return to the caller.

NOTE

The stub executes at the caller's execution level.

The code sequence for the called stub will be similar to the following:

```
MFSP *      sr0,gr0                    ; NOP to identify stub
BL          disp,gr2                   ; Branch to local entry point
DEP         gr31,bit31,len2,gr2        ; Depcsit caller's Exec. Level in link
LDW         -28(SP),gr21               ; Restore sr4  (part 1)
LDW         -24(SP),gr2                ; Restore return address (RP')
MTSP        gr21,sr4                   ; Restore sr4  (part 2)
BE          0(sr4,gr2)                 ; Branch back to caller
LDW         -32(SP),gr27               ; Restore DP
```

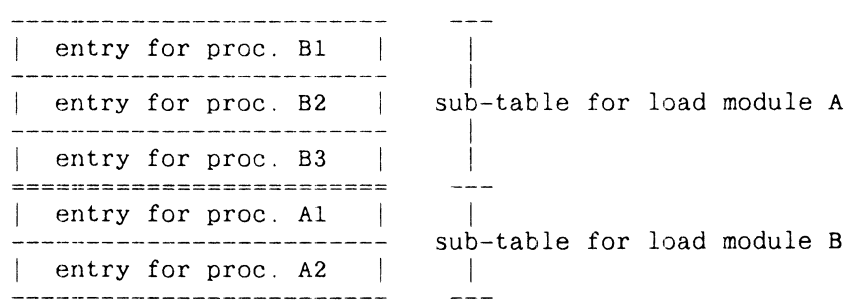
* This instruction is not executed; the entry into the stub is actually at the second instruction. (This instruction is a remnant of an early stack unwind effort -- it will be removed in the future.)

Inter-Module Cross Reference Table (XRT)

The Cross Reference Table (XRT) is used to link the external procedure calls of an object module. Every process has an XRT area reserved from its process space (pointed to by sr5) for the XRTs of the object modules being executed by the object module. The table contains a sub-table for each load module referenced during the process execution. Each sub-table for a load module contains entries for all the procedures called by that load module. A sample XRT is shown in Figure 7 on the next page (in this example, the process has two load modules: 'A' and 'B'. 'A' calls procedures B1, B2 and B3; 'B' calls procedures A1 and A2):

The Layout of the XRT

One XRT might be visualized as shown in Figure 8-2. (This diagram corresponds to the calling situation described in the last sentence of the previous section.)



#include note2.tag

Figure 8-2. Layout of the XRT.

One XRT Entry

An entry for a procedure within a sub-table of a load module in the XRT (e.g. the entry for B1) is eight words long, and contains the following information:

1. The SID of the module to which it belongs.

2. The entry offset for the procedure. This is a 32-bit offset, and is the address of the entry point (relative to the base of the SID of its load module) of the procedure's called stub. The last two bits (30 and 31) of this word must be zero in order to insure word alignment of the address.

3. The DP value for the load module to which it belongs (the value of the base register pointing to the load module's global data area).

4. The LP value of the module in which the called procedure is contained. This is a pointer to the beginning of the XRT sub-table of that load module.

5. The address of the CALLX millicode routine(s).

6. ,7., 8. (These three words are presently undefined and are reserved for future use.)

Thus, the XRT entry for procedure B1 (which is called from A and would appear in A's sub-table of the XRT) would look as is shown in Figure 8-3.

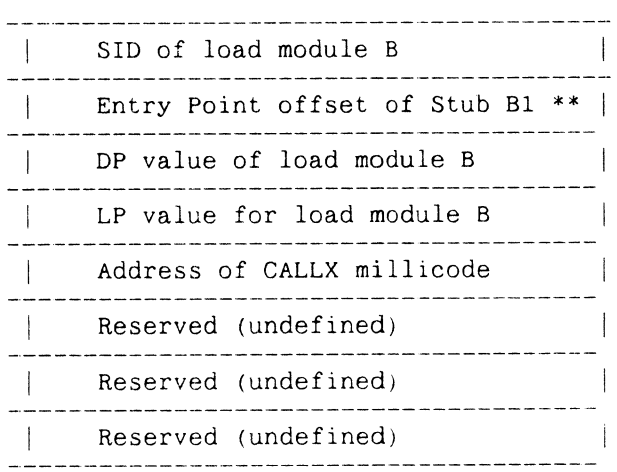


Figure 8-3. One XRT Entry.

** The last two bits of the Entry Point Offset must be set to zero in order to ensure word alignment.

NOTE

If a load module contains no external references, its sub-table in the XRT will be empty.

Linkage Pointer

A single value, the Linkage Pointer, resides in the word directly below the global data area of a load module, at the location pointed to by DP-4. This pointer is private to the load module, and is a short pointer to the beginning of the load module's XRT sub-table. An entry for a called procedure in the XRT is pointed to as follows:

1. The LP points to the beginning of the sub-table in the XRT of the load module containing the called procedure.

2. The calling stub for the caller has the offset to the called procedure's entry relative to the XRT sub-table of the caller's load module. This offset, added to the LP value, provides a pointer into the called procedure's entry. This LP-relative XRT offset is assigned by the linker.

The reason for the indirection employed by using the LP is that load modules can be shared by different processes whose XRTs may also be different. To allow the same code to reference the same load module in different processes' XRTs, it is necessary to provide a uniform interface to the XRT entries; this is provided by the LP.

In addition to the XRT area in the process space, there is an XRT area in the system space (pointed to by sr7) that is reserved for the XRTs of system load modules. Like any other load module, a system load module also uses LP to locate its XRT. The system XRT area can also contain a special XRT that is used for calling system procedures by intrinsic number.

Linker/Loader Interaction with XRT

The XRT will be set up by the loader. The values in the XRT will be supplied by the loader, based on the mapping of the files relevant to the process into virtual memory (i.e. SID allocation, the data offsets in sr5 space, etc). The linker may provide some of the values that are to be contained in sr5, based on the information it may have at link time concerning the specific load modules that are involved in the process' executable image.

When a process is loaded, the loader will protect all the pages in the XRT to read level 3, write level 0. Although it is not necessary, the process protection ID will be assigned to the pages of the process XRT area. The protection of the LP will be the same as the object module specifies for its global data area. Since the LP is set whenever an object module is called by an untrusted procedure, it does not need any special protection.

The XRT of every process and the system XRT must be at the same offset of their corresponding quadrant, and every XRT must be the same length. These two restrictions allow the procedure call millicode to use a very simple masking algorithm to perform bounds checking on any XRT pointer used with an external call. The location and size of the area can be changed when the system is restarted, but the new values must be reflected in the procedure call millicode (because it uses constant values to do bounds checking on the XRT entry pointer).

Stub Conventions for External Calls

By providing a consistent interface between calling and called stubs across implementations, code portability (i.e. moving executable files between operating system implementations) can be achieved at the load module level. This means that load modules can be relocated without relinking, and, eventually, that common tools can be shared. In the case of a distributed (i.e. commercially-owned) system, it means that an application can use an executable file without having to copy it over to its own mass storage system (which it may not even have) and subsequently having to also relink it. If the specifications below are not followed for the implementation of this interface, relinking will always be necessary in order to move a program between different operating systems, or even different versions of the same OS.

Interface Between Calling and Called Stubs

The exact distribution of all operations between the calling and called stubs, or whether or not the calling stub uses a centralized system routine to accomplish these tasks, is not architected. Much more importantly, and specified in detail, is the work that the called stub can expect to have already been done before it is entered. Adhering to these requirements facilitates the loading of DP, LP, and SID (if desired) of the called load module. These expectations are as follows:

1. gr1 contains a pointer to the called procedure's XRT entry. (This is actually the called procedure's XRT entry in the sub-table of the calling load module.)

NOTE

Recall that the caller's LP points to the caller's XRT sub-table, which contains entries for all of the routines that may be called. The offset into that sub-table, which indexes to the called routine's entry, is bound as an immediate in the calling stub. The pointer to the specific XRT entry is calculated as follows:

```
(caller's LP value) + (offset to called procedure's
entry) = (pointer to callee's entry in XRT sub-table of
caller)

This pointer is the value that should be found in gr1
when the called stub is entered.
```

2. The SID of the called load module has been loaded into sr4. (The called stub is free to check the validity of that SID, to reload it, or to leave it as is. The important point is the assumption that it has already been loaded, and DOES NOT need to be checked.)

Summary of an External Procedure Call

Figure 8-4 shows a detailed picture of the flow of control associated with an external procedure call.

NOTE

To read diagram, begin at the upper left-hand corner ("Calling Code"), and read downward; whenever an arrow extends from a line, follow it, and continue downward from the point where the arrow ends.

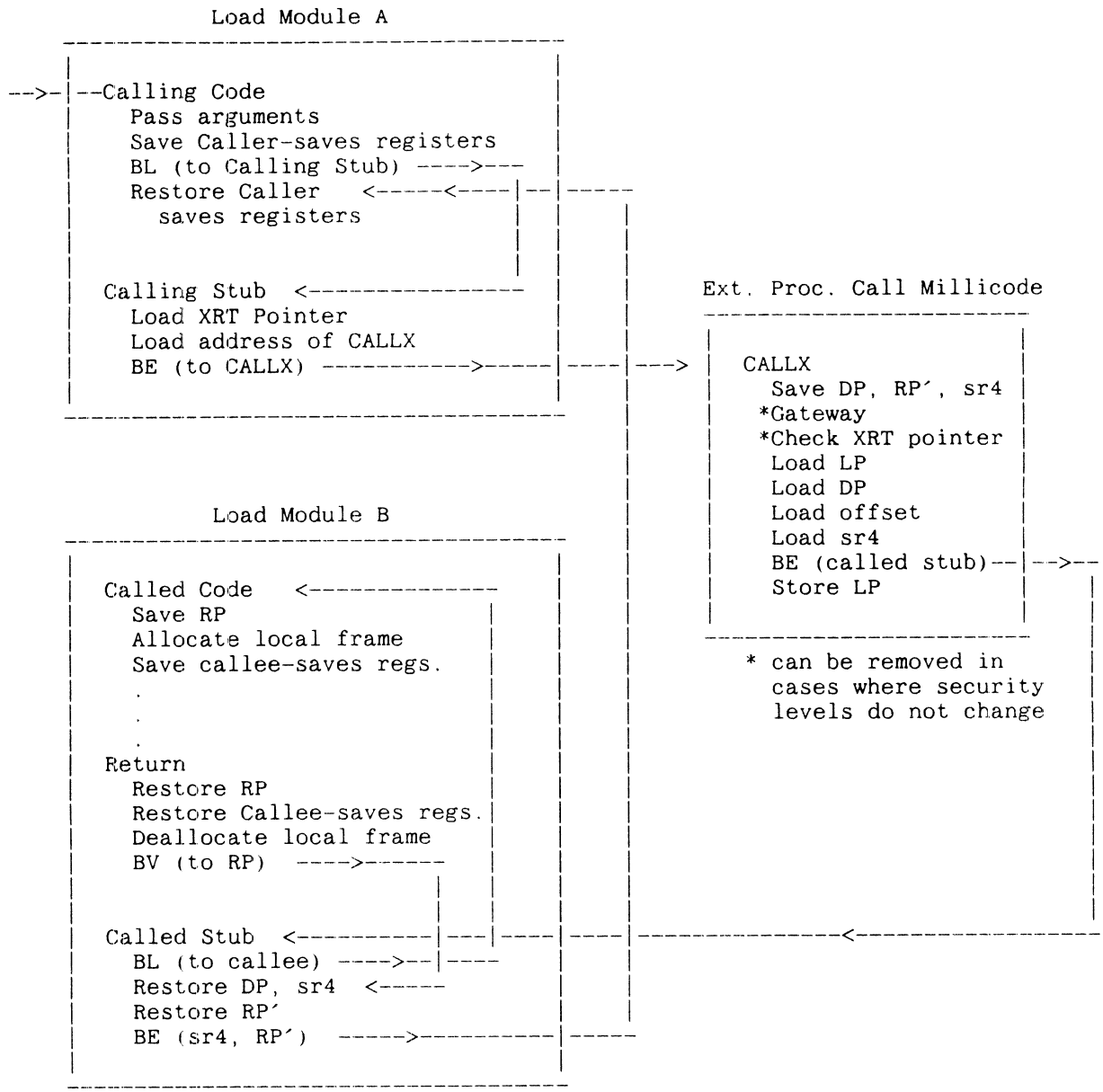


Figure 8-4. Summary of External Procedure Call.

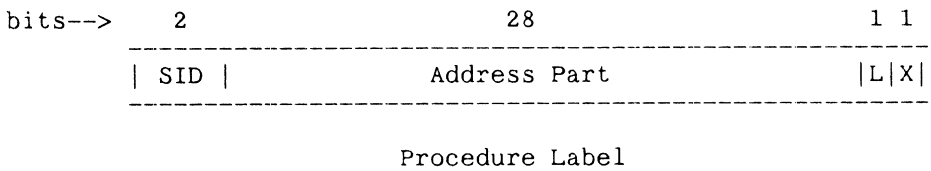
Dynamic Linking

Dynamic linking is run-time linking to routines, which may be necessary due to run-time routine calls. A run-time procedure call is one where the target procedure is unknown at compile time, or where the target of a procedure call can change while the code is executing. Dynamic linking will be carried out through explicit protocols (e.g. the FINDPROC system call in MPE-XL).

If the dynamically called routine resides in a load module that has not yet been loaded, the load module is loaded dynamically. In order to dynamically load a load module, a global data area for it may need to be allocated in sr5 space. This data space is allocated by the loader, and may be allocated from any unused virtual space in sr5.

Procedure Labels

A procedure label is a specially-formatted variable that is used to link dynamic procedure calls. The format of a procedure label is shown below:



Procedure Label

The X field in the address section of the procedure label is the XRT flag, which is used by compilers to determine if the procedure label is local (off) or external (on). In the case of a local procedure label, the address part will be a pointer to the entry point of the procedure, while in the external case, the address part will be a pointer to an XRT entry for the procedure.

The L field in the address part of the procedure label is the static link flag, which is used by the compilers to indicate whether or not a static link must be passed on a call.

NOTE

In the current (first release) implementation, the L-bit is never turned "on" and is therefore effectively unused. This situation may result in a future change in either the specification or the implementation itself.

The dynamic procedure call millicode will actually determine if a procedure label is local or external, and take the appropriate action. (A local procedure label can only be used to call procedures within the current object module.) The following pseudo-code sequence demonstrates the process used for dynamic calls (note the similarity between this sequence and the calling stub sequence):

```
IF (X-bit in Plabel) = 0 THEN
    Branch Vectored using Plabel
ELSE BEGIN
    Clear X-bit;
    Calculate address of XRT entry (LP + Plabel value)
    Save DP;
    Load address of CALLX;
    Save RP';
    Move sr4 to old DP register;
    Branch to CALLX;
END.
```

NOTE

The X and L flags must be zero during an external call, or they will cause a misaligned data reference trap when accessing the XRT. (As earlier mentioned, the L flag is currently unused, so it must be assumed to be zero).

An external procedure label can be used in conjunction with the external procedure call millicode to call any procedure within the process or the operating system (subject to XLeast checking to insure adequate execution level). The procedure call millicode

only uses the address part of the procedure label, but it may point to either the process or system XRT.

The procedure FINDPROC may be used to get an external procedure label for any level 1 procedure in a process. If the compiler or linker determines the need for an external label, it is communicated to the loader by a normal import request or an explicit call to FINDPROC.

Although a procedure label pointing to a system XRT entry is valid for all processes, it will be unloaded when its reference count drops to zero. Therefore, these procedure labels should not be considered as global procedure labels. The procedure GET_SYS_LABEL will return a global procedure label for any procedure in a system object module, but it requires privilege level 1 to be called.

9

Millicode Calls

Overview

Background

In a complex instruction set computer, it is relatively easy at system design time to make frequent additions to the instruction set based almost solely on the desire to achieve a specific performance enhancement, and the presence of microcode easily facilitates such developments. In a reduced instruction set computer, however, this microcode has been eliminated because it has been shown to be potentially detrimental to overall system performance (not only is instruction decode complicated, but the basic cycle time of the machine may be lengthened).

So while the functionality of these complex microcoded instructions (e.g. string moves, decimal arithmetic) is still necessary, a RISC-based system is confronted with a classic space-time dilemma: if the compilers are given sole responsibility for generating the necessary sequences, the resulting in-line code expansion becomes a problem, but if procedure calls to library routines are used for each operation, the overhead expense incurred (i.e. parameter passing, stack usage, etc) is unacceptable.

In an effort to retain the advantages associated with each approach, the alternative concept of "millicode" was developed. Millicode is the Precision Architecture's simulation of complex microcoded instructions, accomplished through the creation of assembly-level subroutines that perform the desired tasks. While these subroutines perform comparably to their microcoded counterparts, they are architecturally similar to any other standard library routines, differing only in the manner in which they are accessed. As a result, millicode is portable across the entire family of HPPA machines, rather than being unique to a single machine (as is usually the case with traditional microcode).

There are many advantages to implementing complex functionality in millicode, most notably cost reduction and increased flexibility. Because millicode routines reside in system space like other library routines, the addition of millicode has no hardware cost, and consequently no direct influence on system cost. It is relatively easy and inexpensive to upgrade or modify millicode, and it can be continually improved in the future. Eventually, it may be possible for individual users to create their own millicode routines to fit specific needs.

Because it is costly to architect many variations of an instruction, most fixed instru ction sets contain complex instructions that are overly general. Examples of this are the MVB (move bytes) and MVW (move words) instructions on the HP3000, which are capable of moving any number of items from any arbitrary source location to any target location. Although the desired functionality is achieved with such generalized complex instructions, the code that is produced often lacks the optimization that could have been achieved if all information available at compile time had been utilized. On microcoded machines, this information (concerning operands, alignment, etc) is lost after code generation and must be recreated by the microcode during each execution, but on the Precision Architecture machines, the code generators can apply such information to select a specialized millicode routine that will produce a faster run-time execution of the operation than would be possible using a generalized routine. For example, the Move routines can execute much faster if they can assume a specified alignment, and therefore eliminate any error checking of that type.

The size of millicode routines and the number that can exist are not constrained by considerations of the size of available control store, because millicode resides in the system as subroutines in normally managed memory, either in virtual memory where it can be paged into and out of the system as needed, or in resident memory, as performance considerations dictate. A consequence of not being bound by restrictive space considerations is that compilers can be developed with many more specialized instructions in millicode than would be possible in a microcoded architecture, and thus are able to create more optimal solutions for specific source code occurrences.

Millicode routines are accessed through a mechanism similar to a procedure call, but with several significant differences. In general terms, the millicode calling convention stresses simplicity and speed, utilizing registers for all temporary argument storage and eliminating the need for the creation of excess stack frames. Thus, a great majority of the overhead expense associated with a standard procedure call is avoided, thereby reducing the cost of execution. (However, there are exceptions to these conventions, which are discussed in more detail throughout this chapter.)

The guidelines for the inclusion of a routine in the millicode library are not completely determined, but the general considerations are frequency of usage, processor expense (number of cycles necessary for execution), and size. Most routines perform common, specific tasks (such as integer multiply or divide), and require very little or no memory access.

The Millicode Hierarchy

In an effort to define and classify the various types of millicode, Figure 9-1 shows a conceptual schematic layout of the existing millicode "family". The labels in the boxes are briefly described following the diagram, and will be discussed in greater detail throughout the remainder of the chapter.

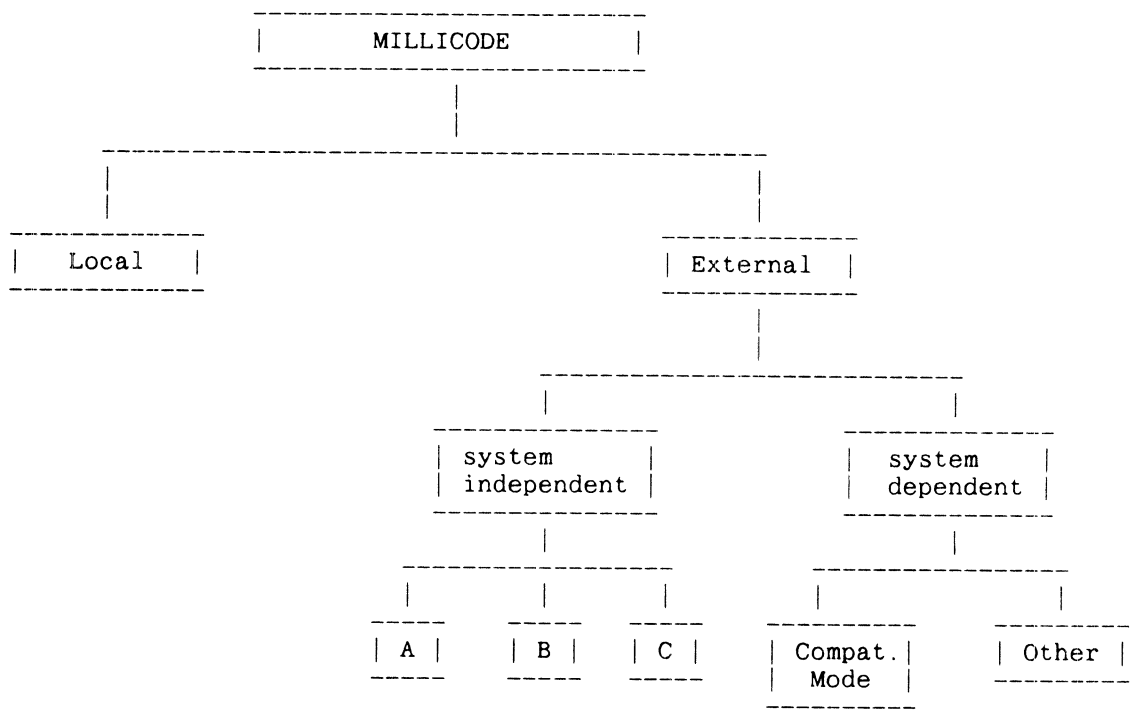


Figure 9-1. Millidcode Overview.

Descriptions

Local: The millicode routines contained in the executable code of each process requiring them (i.e. not shared). Similar to local library routines, but with the faster access advantage.

External: True, shared millicode, residing in system space.

System Dependent: Millicode which is useful to only one operating system; can be used at the discretion of the operating system.

System Independent: Millicode which is intended to run on more than one operating system (i.e., routines required by language code generators).

Compatibility Mode: System dependent code (on MPE-XL) which assists both the emulator and translator.
System Dependent - Other: Additional system dependent routines.

A, B, C : Classes of millicode that differ by location in virtual space and accessibility. The classifications are made on the basis of performance and size considerations.

Introduction to Local and External Millicode

As pictured in Figure 10 (tree diagram in the main introduction), millicode is generally divided into two main categories: local and external. Although it appears that the two types are co-existing, this is NOT the case; at present (first release) all millicode is local, whereas in the future it is likely that the great majority, if not all, of millicode will be external.

The two types of millicode are easily differentiated by the way in which they are accessed and used; local millicode is linked with and executed by any process that requires it, while external millicode is handled in much the same manner as a standard shared library (i.e. one system-resident copy that is shared by all processes as is necessary).

Efficiency Factors

There are several conditions that contribute to the increased efficiency of millicode calls. The primary one is the fact that any standard routine that makes only millicode calls is considered to be a leaf routine, thereby eliminating the overhead expense (i.e. frame allocation) that would have been added by the presence of another standard procedure call. A higher percentage of leaf routines improves overall efficiency, since a standard procedure call is much more costly than a millicode call. (in terms of stack frame allocation and usage, etc). This and other major factors contributing to this efficiency are summarized below:

1. The compiler is able to identify whether a routine is a leaf or not; it only builds a complete stack frame for non-leaf routines. In the diagram below, a stack frame is created upon the call to Proc1 in both Fig. A and Fig. B. In Fig. A, another frame is then allocated for Proc2 when the compiler realizes that Proc2 will be subsequently calling Proc3, whereas in Fig. B, no additional frame is necessary because the compiler realizes that Proc2 is only making a millicode call.

(Fig. A) (Fig. B)

2. More parameters can be passed in registers to a millicode routine than to a standard proc call. (See Millicode document, section 4, for more details of register usage and parameter passing.)

3. The compiler knows more about millicode routines at compile time than it does about user-defined procedures, so it can perform some inter-procedural optimizations which cross the millicode call.

4. The millicode calling mechanism is often faster than the standard procedure call; a millicode routine is called through a branch (BL) directly to the routine or to a pointer to the routine.

Making a Millicode Call

A call to a millicode routine can only be made from assembly language (no high-level language access), and it is made through a branch directly to the routine or to a branch to the routine.

It is intended that the standard register conventions be followed, with two exceptions:

1. The return address (MRP) is carried in gr31; and

2. The return value is carried in gr29.
   There are, however, many non-standard practices regarding millicode register usage; see the Millicode document for further details.

   Local millicode can be accessed with three different methods, depending on its location relative to currently executing code.

These three methods are:

1. A standard Branch and Link (BL), if the millicode is within 256K bytes of the caller,
2. A BLE instruction, if the millicode is within 256K bytes of a predefined code base register, and

3. The two-instruction sequence (LDIL,BLE) that can reach any possible address, or a BL with a linker-generated stub.

Nested Millicode Calls

Millicode routines may call other millicode routines, but (at present) cannot call other standard user-defined routines. In order for nested millicode calls to occur, however,

the millicaller must allocate a stack frame for the millicallee, and, upon call to the millicallee, save the MRP in the static link word (SP-20) of the frame marker area of the new frame. The layout of a frame generated for a nested millicode call is shown in Figure 9-2.

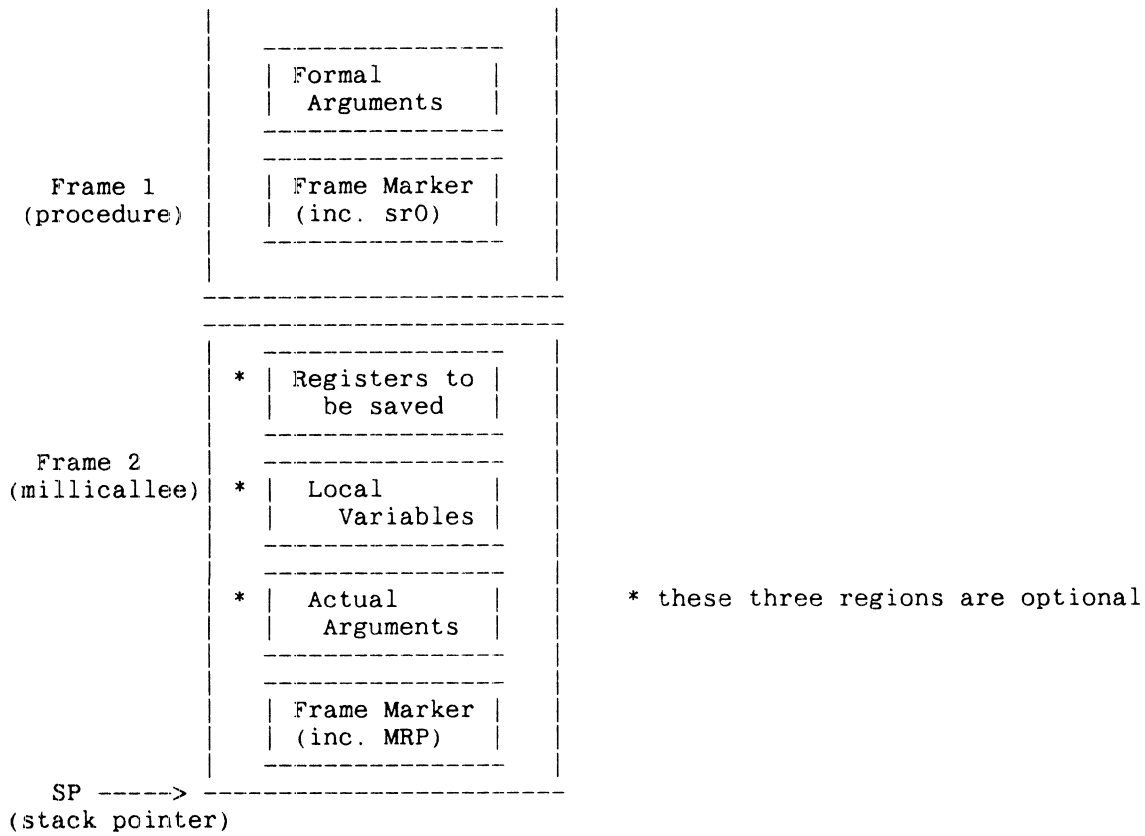


Figure 9-2. Millicode Stack Storage Layout.

10

Stack Unwinding

Introduction

Stack unwinding is the Precision Architecture's implementation of procedure traceback and context restoration, a process that has many possible applications for any executing program, both at the system level and the user interface. Unwinding is necessary because in the event of any type of interruption of execution, there is insufficient information immediately available to perform a comprehensive stack trace, which is the basic operation behind state restoration.

Other processes that are heavily dependent on the presence of the unwinding facility are system dump analysis, which is performed specifically to determine the cause of system crashes, debugging, and certain language-specific features such as the escape and the non-local GOTO in Pascal. A dump analysis examines all system processes that were running at the time of the crash, an operation which encompasses multiple stack traces. Debugging is the analysis of the current and past status of a program, either at the system or user level, with the objective of locating an unintended occurrence (i.e. an error).

This section will detail the specifications that must be followed in order to enable stack unwinding of procedure calls that are assumed to conform to the other aspects of the calling convention. For a more thorough discussion of stack unwinding, refer to the Stack Unwinding document.

Requirements for Unwinding from a Local Procedure Call

Unwinding is dependent upon the ability to identify each instruction in an entry or exit sequence that modifies SP, and the point at which RP gets restored in an exit sequence. Furthermore, it is necessary that all registers be saved in the specified areas, and that any other necessary conditions (i.e. procedure-specific) be satisfied.

In order to guarantee unwindability, the standard procedure call convention (as described earlier in this document) must be followed by both the caller and callee. It is mandatory that the procedure entry and exit sequences conform to the standard specifications, a condition that is insured by the compilers.

All compiled code will automatically conform to these requirements, but handcoded routines must also meet these standards. (It is the responsibility of the author of the code to use the assembler directives .ENTER, .LEAVE and .CALLINFO to generate the standard entry/exit sequences, or to hand-code the correct entry/exit sequences.)

See the Appendices of the Stack Unwind document for examples of standard entry/exit sequences.

Requirements for Performing a Stack Trace

The minimum requirements to successfully perform a stack trace are as follow:

1. The specified point of the interrupt must lie within a standard code sequence, as specified above.
2. Call-save registers must be saved and restored across a call.
3. Unwind table entries must be generated for each routine, and for any discontinuous sections of code.
4. The frame size must be as stated in the unwind descriptor (see section 2.2 of Stack Unwind for details of unwind descriptors).
5. The RP (or MRP) must conform to the specifications stated in the unwind descriptor.

The minimum requirements to successfully perform an escape or a non-local GOTO are as follows:

1. All requirements for a stack trace (as above) must be met.
2. The state of the entry-save registers must conform to the specifications given in the unwind descriptor.

Assembler Interaction

The .ENTER and .LEAVE directives will cause an entry/exit sequence to be produced by the assembler. The assembler generates these according to the .CALLINFO directive, which causes the necessary information (assuming it is available) to be put into the unwind descriptor. The unwind descriptor is a two-word structure which lies in the four-word unwind table. The table is formatted as follows:

```
-------------
| word #1   |  .PROC   (start address of the procedure)
-------------
| word #2   |  .PROCEND (end address of the procedure)
-------------
| word #3   |  \
-------------      .CALLINFO  (unwind descriptor)
| word #4   |  /
-------------
```

See Sections 2.1 and 2.2 of the Stack Unwind document for further discussion of unwind tables and descriptors.

Unwinding From an External Procedure Call

Unwind Table for Stubs

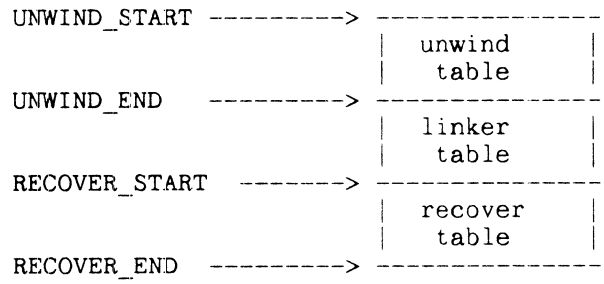
The linker builds two-word unwind descriptors for stubs. Each type of stub involves its own unwind descriptor, and there is a different type of descriptor used for each type of stub. The unwind descriptor for a stub contains the following information:

```
word 1: address of the first instruction of the stub
word 2:
        bits 0..4 - reserved
        bits 5..7 - type of stub
        bits 8..10 - reserved
        bits 11..15 - used only for parameter relocation stub;
                      contains the number of the instruction
                      which stores RP on the stack in the stub.
        bits 16..31 - length (# of words) of stub area
```

NOTE

In some cases, a contiguous sequence of calling, called, or long branch stubs can be covered by a single unwind descriptor.

The unwind, stub and recover tables are pointed at by the UNWIND and RECOVER subspaces, and arranged in code space as follows:



Unwinding from Stubs

Calling Stubs: None of the significant registers are modified; RP still contains the return address.

```
Called Stubs: All significant registers are on the stack. RP and DP are
              stored by calling stub, and SID is stored by CALLX.
              Their locations are:
                 RP      : SP-24
                 SID     : SP-28
                 DP      : SP-32

Parameter Relocation Stubs:  It must be determined (from the unwind
              descriptor) if the current address is before or after
              the instruction which stores RP on the stack. If it
              is before, RP still contains the return address;
              otherwise the return address will be stored on the stack.

Long Branch Stubs: No changes have been made.
```

Calling Situations That May Not Support Unwinding

The main type of call from which unwindability cannot be guaranteed is one to millicode, because the assembler cannot automatically generate the standard entry and exit sequences for millicode routines that utilize stack space. This does not present a major problem, however, because relatively few millicode routines necessitate the creation of a stack frame (see section 3 of Millicode document). It is possible, however, to support unwinding from such situations (i.e. nested millicode calls), provided that the millicode routine which will use stack space is written so that it will independently generate the correct entry and exit sequences. It is the responsibility of the author of the specific routine to incorporate these provisions into the actual code.

Other instances in which unwinding may fail:

1. Procedures that have multiple entry points. (secondary entry/exit sequences).

2. Code sequences in which the DP is modified. As a precaution, the DP must never be altered by user code, only by system code as is absolutely necessary.

For more details on Unwinding, see the Stack Unwind document.

A

Code Examples

Standard Procedure Calls

The assembly listing on the following pages was produced by the Pascal compiler (WITHOUT optimization) when given the source code shown below. The approximately equivalent C and FORTRAN source code is shown on the next two pages, and significant differences are noted (either in the source code or the documentation of the assembly code) where appropriate.

```
*************************************************************************

program test;

function mul (a,b : integer): integer;
begin
    mul := a * b;
end;

procedure proca (    a,b : integer;
                 VAR c,d : integer;
                     e,f : integer);
begin
    c := a + b;
    d := mul(a,b);
end;


procedure one;
var a,b,c,d,e,f : integer;
begin
    a := 5;
    b := 10;
    proca (a,b,c,d,e,f);
    e := c + d;
    f := e;
end;


begin
    one;
end.
```

C Source Code Equivalent

```
#include <stdio.h>

main()
{
    one();
}

one()
{
    int a, b, c, d, e, f;
    a = 5;
    b = 10;
    proca (a, b, &c, &d, e, f);
    e = c + d;
    f = e;
}

proca(a, b, c, d, e, f)
int a, b, *c, *d, e, f;
{
    *c = a + b;
    *d = mul (a,b);
}


mul(a,b)
int a,b;
{
    return (a*b);
}
```

FORTRAN Source Code Equivalent

```
      call one
      end

      subroutine one
      integer*4 a,b,c,d,e,f
      a = 5
      b = 10
      call proca(a,b,c,d,e,f) -->   Note: In FORTRAN, all parameters are passed
      e = c + d                           by reference, so it is impossible to
      f = e                               simulate the difference between
      return                              Pascal's VAR and Value parameters.
      end

      subroutine proca(a,b,c,d,e,f)
      integer*4 a,b,c,d,e,f
      c = a + b
      d = mul(a,b)
      return
```

```
      end

      function mul(a,b)
      integer*4 a,b
      mul = a*b
      return
      end
```

Assembly Listing

NOTE

The numbers and letters in parenthesis are used as labels for the documentation that follows the listing.

```
;******************************************************************************

$L0     .EQU    -48                 (a)
$L1     .EQU    -40
$L1000  .EQU    -48
$L1000  .EQU    -80

        .SPACE  $TEXT$              (b)
        .SUBSPA $CODE$,QUAD=0,ALIGN=8,ACCESS=44,CODE_ONLY

mul
        .PROC                       (c)
        .CALLINFO CALLER,FRAME=-8,ENTRY_SR=1
        .ENTRY
;       .EXPORTED

        LDO     40(30),30       ;   (11)
        STW     26,-76(0,30)    ;
        STW     25,-80(0,30)    ;
        LDW     -76(0,30),26    ;
        .CALL                   ;
        BL      $$muloI,31      ;
        LDW     -80(0,30),25    ;

        STW     29,-40(0,30)    ;   (12)
        LDW     -40(0,30),28    ;
00002711 (dummy label)
        BV      0(2)            ;   (13)
        .EXIT                   ;   (d)
        LDO     -40(30),30      ;
        .PROCEND;in=25,26;out=28;
```

```
proca
        .PROC                           (c)
        .CALLINFO CALLER,FRAME=0,ENTRY_SR=1,SAVE_RP
        .ENTRY
;       .EXPORTED

        STW     2,-20(0,30)     ;   (7)
        LDO     48(30),30       ;

        STW     26,-84(0,30)    ;   (8)
        STW     25,-88(0,30)    ;
        STW     24,-92(0,30)    ;
        STW     23,-96(0,30)    ;

        LDW     -84(0,30),1     ;   (9)
        LDW     -88(0,30),31    ;
        ADDO    1,31,19         ;
        LDW     -92(0,30),20    ;
        STW     19,0(0,20)      ;

        LDW     -84(0,30),26    ;   (10)
        .CALL ; in=25,26; out=28;
        BL      mul,2           ;
        LDW     -88(0,30),25    ;

        LDW     -96(0,30),21    ;   (14)
        STW     28,0(0,21)      ;
00002712 (dummy label)
        LDW     -68(0,30),2     ;   (15)
        BV      0(2)            ;
        .EXIT                   ;   (d)
        LDO     -48(30),30      ;
        .PROCEND; in=23,24,25,26;

one
        .PROC                           (c)
        .CALLINFO CALLER,FRAME=32,SAVE_RP
        .ENTRY
;       .EXPORTED

        STW     2,-20(0,30)     ;   (3)
        LDO     80(30),30       ;

        LDI     5,22            ;   (4)
        STW     22,-60(0,30)    ;
        LDI     10,1            ;
        STW     1,-64(0,30)     ;

        LDW     -60(0,30),26    ;   (5)
        LDW     -64(0,30),25    ;
        LDO     -68(30),24      ;
        LDO     -72(30),23      ;
        LDW     -76(0,30),31    ;
        LDW     -80(0,30),19    ;
        STW     31,-52(0,30)    ;
        STW     19,-56(0,30)    ;

        .CALL   ; in=23,24,25,26;   (6)
        BL      proca,2         ;
        NOP                     ;

        LDW     -76(0,30),20    ;   (16)
```

```
        LDW     -80(0,30),21    ;
        ADDO    20,21,22        ;
        STW     22,-76(0,30)    ;
        LDW     -76(0,30),1     ;
        STW     1,-80(0,30)     ;
00002713   (dummy label)
        LDW     -100(0,30),2    ;   (17)
        BV      0(2)            ;
        .EXIT                   ;   (d)
        LDO     -80(30),30      ;
        .PROCEND                ;


PROGRAM
_start
        .PROC                           (c)
        .CALLINFO CALLER,FRAME=0,SAVE_SP,SAVE_RP
        .ENTRY
;       .EXPORTED

        STW     2,-20(0,30)     ;   (1)
        LDO     48(30),30       ;
        STW     0,-4(0,30)      ;
                :
                :
        < calls to system process initialization procedures >
        < these would not appear in the C compiler output   >
                :
                :
        .CALL                   ;    (2)
        BL      one,2           ;
        NOP                     ;
                :
                :
        < calls to system process termination procedures    >
        < these would not appear in the C compiler output   >
                :
                :
        LDW     -68(0,30),2     ;   (19)
        BV      0(2)            ;
        .EXIT                   ;   (d)
        LDO     -48(30),30      ;
        .PROCEND    ; in=24,25,26;

                                (e)
        .SUBSPA $UNWIND$,QUAD=0,ALIGN=8,ACCESS=44
        .WORD   mul
        .WORD   mul+40  ; = 0x28
        .WORD   32768   ; = 0x8000
        .WORD   5       ; = 0x5
        .WORD   proca
                :
                :
           < unwind table information continues >


        .SPACE  $TEXT$                          (b)
        .SUBSPA $CODE$

        .EXPORT mul,ARGW0=GR,ARGW1=GR,RTNVAL=GR   (f)
                :
                :
```

```
        < .EXPORT list continues >

        < .IMPORT list begins    >          (g)
                  :
.IMPORT $$muloI,MILLICODE
.END
```

Documentation

Code Description

NOTE

The relevant assembler directives are summarized in Appendix B (page 60), and the other compiler-generated information is briefly explained following the code comments.

(Numbers below correspond to those accompanying the blocks of code; they appear in the order in which they would be executed. In other words, the code documentation follows the program flow of control.)

1. The beginning of the main program block (note that the main program is handled in much the same manner as a standard procedure). Because other procedures will be subsequently called, it is necessary to store the Return Address and allocate a stack frame. The Return Pointer (RP), which is currently in gr2, is first stored onto the stack at SP-20, and then SP (gr30) is incremented (by 48 bytes) in order to create the new frame. Also, the 'Previous SP' field is initialized to zero (recall that gr0 is the zero value register), in order to signify the termination point for stack unwinding. (In the compiled C code, this initialization would not appear because the outer block is handled differently.)
2. CALL to the procedure 'one'. The return pointer (RP), which is the address of the second instruction following the BL, is put into gr2. The delay slot (i.e. the instruction following the branch) is followed by a NOP because there is no operation that the compiler could have inserted there.
3. ENTRY to procedure 'one'. Again, this is a non-leaf procedure, so it is necessary to store RP onto the stack at SP-20 and then allocate a new frame by incrementing SP (this time the increment is 80 bytes in order to accommodate the local variables).
4. The immediate values 5 and 10 are loaded into gr22 and gr1 respectively, and these registers are stored onto the stack at SP-60 and SP-64. This block correspond to statements 'a:=5' and 'b:=10'.

5. Loading arguments (into caller-saves registers). This can be divided into three categories. First, the values stored on the stack at SP-60 and SP-64 (corresponding to 'a' and 'b') are loaded into arg0 and arg1 (gr26 and gr25). Second, the addresses SP-68 and SP-72 are loaded into arg2 and arg3 (gr24 and gr23). Corresponding to variables 'c' and 'd', these two arguments are loaded with addresses rather than actual values due to the fact that they are being passed by reference (i.e. VAR parameters). Third, the values stored on the stack at SP-76 and SP-80 (corresponding to 'e' and 'f') are loaded into gr31 and gr19 respectively (these two are serving as scratch registers), and then stored onto the stack at SP-52 and SP-56. Note that these two parameters must be stored onto the stack because the argument registers have already been filled. (In the compiled FORTRAN code, it would become evident that all parameters are passed by reference, as in the Second category above, as is dictated by the FORTRAN language.)

6. CALL to procedure 'proca'. Note that the .CALL directive is followed by a note indicating that arguments will be passed to the procedure in gr23-26. The delay slot is filled with a NOP, although it could have been filled with another operation (e.g. one of the preceding STW or LDW instructions). As with all BL instructions, the return address is simultaneously loaded into gr2 (or gr31 for millicode).

7. ENTRY to procedure 'proca'. As before, this is a non-leaf procedure, so it is necessary to store RP at (SP-20) and allocate an additional stack frame by incrementing SP (48 bytes in this case).

8. The values held in the four argument registers (gr26-23) are stored onto the stack in the fixed arguments area of the PREVIOUS (caller's) frame. This is determined by subtracting the size of the current frame (48 bytes) from the offset (84, 88,..), and using the result as the offset into the previous frame. These words correspond to the parameters 'a' through 'd'. (Note that these Store operations are actually unnecessary, and would probably be removed by the optimizer.)

9. The words at SP-84 and SP-88 (parameters 'a' and 'b') are loaded into gr1 and gr31 respectively and the add operation (a+b) is performed, with the result being put into gr19. After SP-92 (which contains the address of 'c') is loaded into gr20, the gr19 value is stored at that address.

10. CALL to function 'mul'. After the two parameters ('a' and 'b') are loaded into arg0 and arg1 (gr26 and gr25), the branch is made to 'mul', and the return address is put into gr2. Note that in this case, the delay slot is filled with an operation (the loading of the 'b' value).

11. ENTRY to function 'mul' and CALL to millicode routine 'mulol'. Although 'mul' is a leaf routine (it makes only a millicode call), a temporary local frame is allocated because the function return value will later be temporarily stored onto the stack, but RP is not stored onto the stack because no additional procedure calls will be made. (This temporary frame is actually unnecessary, and would be removed by the optimizer.) The two arguments ('a' and 'b', in arg0 and arg1 are stored onto the stack, and then reloaded into registers in order to be sent to the millicode routine that will perform the multiply operation. Then the branch is made to the millicode routine ('$$mulol'), with the return address being stored in gr31 (MRP).

NOTE

This code would be further optimized by accessing the arguments directly from the registers in which they enter 'mul', thereby eliminating the argument stores and loads.

12. The millicode return value (in gr29) is stored onto the stack, and subsequently loaded into gr28, which is the procedure return register (ret0). This sequence would probably be optimized to be a simple 'COPY 29,28' instruction.

13. EXIT from function 'mul'. Deallocate the local frame, and return back to the caller ('proca'). The BV (Branch Vectored) instruction, which also uses gr2 as the return pointer, accomplishes this return.

14. RETURN from 'mul' to 'proca'. The value stored at SP-96 (the address of 'd') is loaded into gr21, and then the return value (in gr28) is stored at that address.

15. EXIT from procedure 'proca'. The return address is loaded into gr2 from the 'RP' field of the Previous frame, and the branch is made to that address. The delay slot is filled with the instruction that deallocates the local frame by decrementing SP.

16. RETURN from 'proca' to 'one'. The values stored at SP-68 and SP-72 (the current values of 'c' and 'd') are loaded into gr20 and gr21, and the add operation (c+d) is performed, with the result being put into gr22. This result is then stored onto the stack at SP-76, which is the location assigned to 'e'. Finally, the value stored in the 'e' word is reloaded (into gr1), and then stored into SP-80, which is the location of 'f'. (This is the 'f:=e' operation.)

17. EXIT from 'one'. The return address is taken from its memory location (SP-100) and loaded into gr2, the local frame is deallocated by decrementing SP, and the branch is taken to the return point in the main program.

18. EXIT from main program. The return address (i.e. to the system) is loaded into gr2, the local frame is deallocated by decrementing SP, and the branch is made to the system address.

Other Compiler-Generated Information

(Letters correspond to those accompanying directive blocks.)

a. The .EQU directives indicate the size of each stack frame that is built within the process.

b. SPACE and .SUBSPA specify the proper space and subspace in the system that contains the current information.

c. This four-directive sequence appears at the beginning of every procedure. The directives are summarized in Appendix B.

d. The directives .EXIT and .PROCEND appear at the end of every procedure. Their functions are summarized in Appendix B.

e. The information necessary for stack unwinding is stored here.

f. The .EXPORT list, which is the list of all procedures contained within this process that can be globally accessed.

g. The .IMPORT list, which is the list of all procedures that this process is dependent upon (includes the system initialization and process termination procedures mentioned in the main program code).

External Calls

The assembly code on the page after next was produced by the MPE-XL Pascal compiler from the Pascal source code shown on the next page. A few additional notes concerning the code sample:

- In the source code, an external call situation has been simulated by assuming that the callee ('one') resides in a different load module than the caller ('two').
- The assembly listing has been abbreviated to include only the code associated directly with the source code. In the complete listing, there would be calling and called stubs for all calls to process initialization and termination procedures (which occur in the outer block/main program as noted below) preceding the section of code shown here.
- Because the use of the CALLX millicode is transparent to the user, it has been just referenced as being 'in system space' to avoid all of the excess detail that would be necessary to use actual addressing. A similar liberty has been taken in a few other cases; where actual offsets appear in the code, they have been eliminated to achieve simplicity.
- The code sample is accompanied (in the left margin) by arrows that follow the flow of control. These arrows function exactly as those used in the flow diagrams in the text; in this case, the starting point is in the main program block, near the bottom of the code sample. Furthermore, all 'critical points' have been labeled with numbers and documented on the page following the assembly code (just as was done in the local call example).
- The assembly code is lightly documented; the code used in the stubs is documented in detail in sections 8.7.1 - 8.7.3, and the small amount of code present other than in stubs is basically the same as that used in any local procedure call.

```
*************************************************************************
```

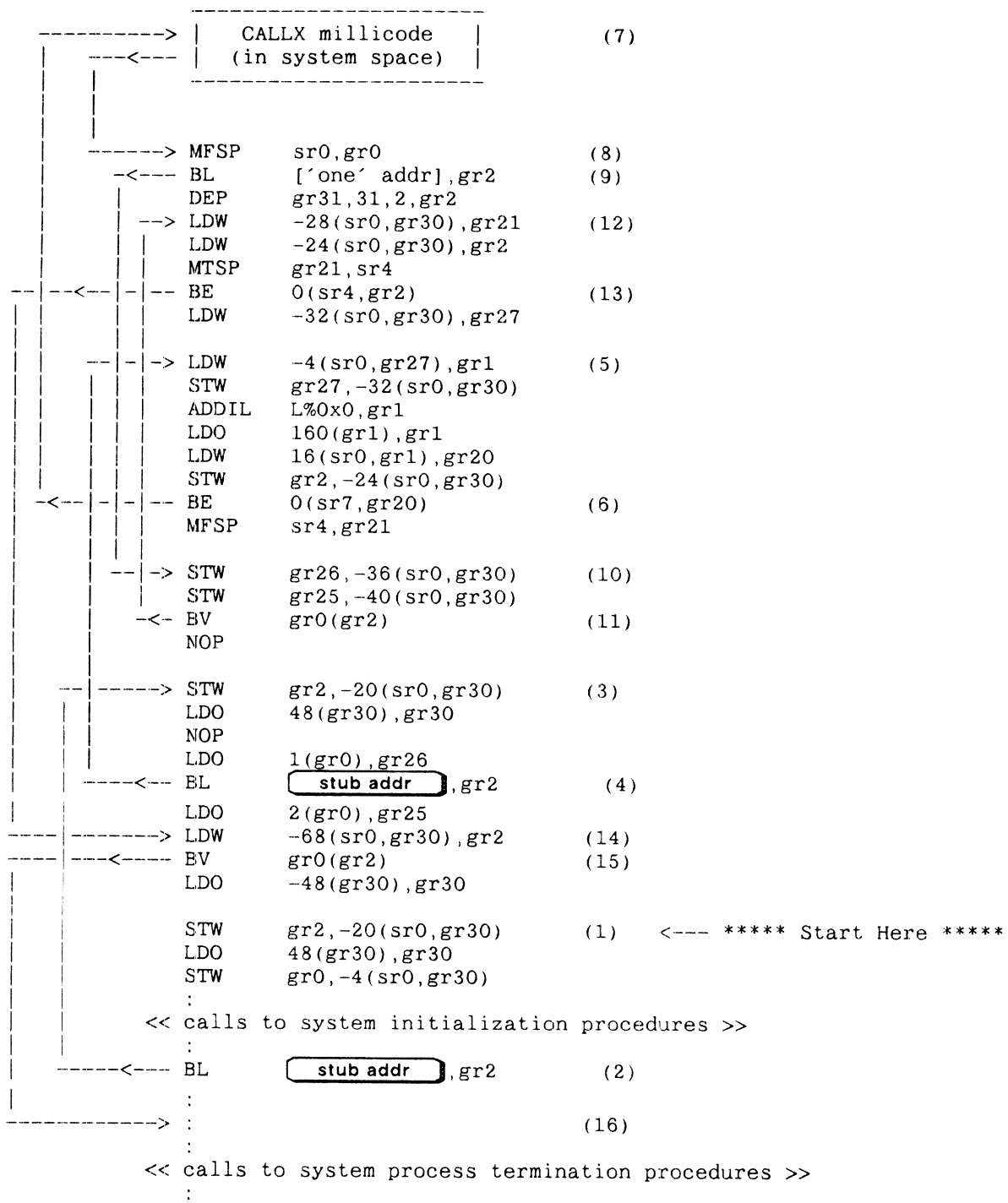
Pascal Source Code

```
program extcall;
:
:
       (* procedure 'one' is in module 2; therefore an external call
          is necessary in order for the call from 'two' to be successful *)

procedure one (a,b : integer);
begin
end;
:
:
       (* procedure 'two' is in module 1, i.e. the same module
          as the main program block. 'Bar' calls 'one'  *)

procedure two;
begin
    one (1,2);
end;

begin
    two;
end.
```

Assembly Code

NOTE

The numbers in parenthesis follow the flow of C and are used as labels for the documentation that follows.


CALLX millicode
(in system space)
MFSP sr0,gr0 (8)
BL ['one' addr],gr2 (9)
DEP gr31,31,2,gr2
LDW -28(sr0,gr30),gr21 (12)
LDW -24(sr0,gr30),gr2
MTSP gr21,sr4
BE 0(sr4,gr2) (13)
LDW -32(sr0,gr30),gr27
LDW -4(sr0,gr27),gr1 (5)
STW gr27,-32(sr0,gr30)
ADDIL L%0x0,gr1
LDO 160(gr1),gr1
LDW 16(sr0,gr1),gr20
STW gr2,-24(sr0,gr30)
BE 0(sr7,gr20) (6)
MFSP sr4,gr21
STW gr26,-36(sr0,gr30) (10)
STW gr25,-40(sr0,gr30)
BV gr0(gr2) (11)
NOP
STW gr2,-20(sr0,gr30) (3)
LDO 48(gr30),gr30
NOP
LDO 1(gr0),gr26
BL stub addr ,gr2 (4)
LDO 2(gr0),gr25
LDW -68(sr0,gr30),gr2 (14)
BV gr0(gr2) (15)
LDO -48(gr30),gr30
STW gr2,-20(sr0,gr30) (1) <--- ***** Start Here *****
LDO 48(gr30),gr30
STW gr0,-4(sr0,gr30)
:
<< calls to system initialization procedures >>
:
BL stub addr ,gr2 (2)
:
: (16)
:
<< calls to system process termination procedures >>
:

Assembly Documentation

1. Start of outer block/main program. RP is saved, the stack frame is allocated, and the unwind delimiter is initialized.

2. Call to procedure 'two'. This is a local call, so the branch goes directly to the procedure, with no stub interaction.

3. Entry to procedure 'two'. RP is saved, the frame is allocated, and the constant values 1 and 2 are loaded into argument registers 1 and 2 (gr26 and gr25).

4. Branch to Calling Stub. As far as the procedure 'two' is concerned, this is the call to the procedure 'one', but the branch actually goes to the calling stub that is necessary because this is an external procedure call.

5. Entry to Calling Stub attached to 'two'. The calling stub performs as is documented in detail in section 8.7.1 of the text.

6. Call to CALLX millicode. The Branch External instruction is used in order to reach the CALLX millicode routine.

7. Execution of CALLX millicode; it performs exactly as is documented in section 8.7.2 of the text, and then branches to the called stub that is attached to procedure 'one'.

8. Entry to called stub attached to procedure 'one'. The first instruction here is effectively a NOP used to identify the beginning of the stub.

9. Branch to procedure 'one'. The standard Branch and Link instruction is used to reach the actual code for the external procedure 'one'.

10. Entry to procedure 'one'. The arguments in gr26 and gr25 are stored onto the stack (this is not necessary, and only remains because the code has not been optimized.

11. Exit from 'one' / Branch back to called stub. The standard return instruction (BV) is used here, although the branch is actually going to the called stub, and not directly to the caller.

12. Re-entry to called stub. The remainder of the called stub performs as is documented in detail in section 8.7.3 of the text.

13. Exit from called stub / branch back to procedure 'two'. The Branch External instruction is used to reach to actual code for the procedure 'two' (the caller).

14. Return to procedure 'two' from called stub. The previous RP is loaded into gr2 from the stack, and the frame is deallocated.

15. Exit from procedure 'two' / branch back to main program block. The standard procedure return is made, because this is a local return.

16. Return to main program block from procedure 'two'. After return, calls are made to the system process termination procedures, and then the frame is deallocated and the return is made to the system.

B

Summary of Assembler Procedure Control

The following table summarizes the Precision Architecture assembler directives that are used to control procedure calling:

Directive	Function
.CALL	Specifies that the next statement is a procedure call.
.CALLINFO	Provides information necessary for generating Entry and Exit code sequences and for creating unwind descriptors.
.ENTRY	Marks the entry point of the current procedure. (compiler-generated)
.EXIT	Marks the return point of the current procedure. (compiler-generated)
.KEEP	Marks the beginning of a procedure's entry code. (compiler-generated)
.EKEEP	Marks the end of a procedure's entry code. (compiler-generated)
.ENTER	Marks the entry point of the procedure being called; causes the assembler to produce the entry code sequence.
.LEAVE	Marks the exit point of the procedure being called; causes the assembler to produce the exit code sequence.
.PROC	Marks the first statement in a procedure.
.PROCEND	Marks the last statement in a procedure.

READER COMMENT SHEET

HP 3000/930 and HP 9000/840 Computers

Procedure Calling Conventions
Reference Manual

09740-90015 November 1986

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below, and use additional pages if necessary.

Is this manual technically accurate? ☐ **Yes** ☐ **No**

Are the concepts and wording easy to understand? ☐ **Yes** ☐ **No**

Is the format of this manual convenient in size, arrangement, and readability? ☐ **Yes** ☐ **No**

Comments:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM: **Date** _____

Name _____

Company _____

Address _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 718 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
ITG Hardware Documentation
19447 Pruneridge Avenue
Cupertino, California 95014

FOLD FOLD

READER COMMENT SHEET

HP 3000/930 and HP 9000/840 Computers

Procedure Calling Conventions
Reference Manual

09740-90015 November 1986

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below, and use additional pages if necessary.

Is this manual technically accurate? ☐ **Yes** ☐ **No**

Are the concepts and wording easy to understand? ☐ **Yes** ☐ **No**

Is the format of this manual convenient in size, arrangement, and readability? ☐ **Yes** ☐ **No**

Comments:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM: **Date** _____

Name _____

Company _____

Address _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 718 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
ITG Hardware Documentation
19447 Pruneridge Avenue
Cupertino, California 95014

FOLD FOLD

READER COMMENT SHEET

HP 3000/930 and HP 9000/840 Computers

Procedure Calling Conventions
Reference Manual

09740-90015 November 1986

We welcome your evaluation of this manual. Your comments and suggestions help us to improve our publications. Please explain your answers under Comments, below, and use additional pages if necessary.

Is this manual technically accurate? ☐ **Yes** ☐ **No**

Are the concepts and wording easy to understand? ☐ **Yes** ☐ **No**

Is the format of this manual convenient in size, arrangement, and readability? ☐ **Yes** ☐ **No**

Comments:

This form requires no postage stamp if mailed in the U.S. For locations outside the U.S., your local HP representative will ensure that your comments are forwarded.

FROM: **Date** _____

Name _____

Company _____

Address _____

_____

_____

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 718 CUPERTINO, CALIFORNIA

POSTAGE WILL BE PAID BY ADDRESSEE

Publications Manager
Hewlett-Packard Company
ITG Hardware Documentation
19447 Pruneridge Avenue
Cupertino, California 95014

FOLD FOLD

Manual Part Number 09740-90015
Printed in U.S.A., November 1986

HEWLETT PACKARD