

PowerPC Run-Time Organization

This chapter describes the PowerPC *function-calling sequence* (the communication that occurs at run time between a function that is called and the function that calls it) and explains how that communication uses the stack frame. It contains the following sections:

§11.1: *Stack-Frame Layout*

§11.2: *Register Usage and Naming Conventions*

§11.3: *Parameter Passing*

§11.4: *Variable Argument Lists*

§11.5: *Application Binary Interface*

§11.6: *Return Values*

§11.7: *Prolog and Epilog Code*

Note: This chapter provides only a brief description of the PowerPC function calling sequence. For a more detailed explanation, plus discussion of other topics not covered in this chapter, see the **System V Application Binary Interface, PowerPC Processor Supplement** (referred to in this chapter as the **ABI Supplement**).

11.1 Stack-Frame Layout

The PowerPC run-time model implements a stack frame in memory and a fixed number of integer and floating-point registers. The stack frame is used for local variables that cannot be stored entirely in registers, and as temporary storage for the contents of certain registers.

Figure 11.1 shows the memory stack frame organization for the PowerPC system. The stack frame grows downward from high to low addresses, and is 16-byte aligned.

Note: The **ABI Supplement** might have lesser constraints on the stack-frame alignment.

Stack-frame header The standard calling sequence does not define a maximum stack frame size. The minimum stack frame consists of the stack-frame header (the *LR save word* and the *back-chain word*), with padding to the required 16-byte alignment.

- Before a function calls another function, the calling function must save the contents of the link register, as they were at the time of entry to the called function, in the LR save word of the caller's stack frame. The called function must also establish its own stack frame.
- The back-chain word always contains a pointer to the previously allocated stack frame.

If a function does not call any other functions and does not require any of the other parts of the stack frame, it does not need to establish a stack frame.

The stack frame can include the following areas as required by any function `fnc`:

- Floating-point register save area — non-volatile floating-point registers modified by `fnc`
- General register save area — non-volatile general registers modified by `fnc`
- CR save area — condition register fields modified by `fnc`
- FPSCR save area — floating-point status and control register bits modified by `fnc`
- Local variable space — local variables of `fnc` not mapped to registers
- Parameter list area — allocated by the caller of `fnc`; must be large enough to contain the arguments that the caller stores in it
- LR save word — contents of the link register as they were at the time of entry to a function called by `fnc`
- Back-chain word — pointer to the previous stack frame's back-chain word

Note: The contents of the parameter list area are not preserved across calls.

Any padding of the frame as a whole must occur within the local variable area. The parameter list area must immediately follow the stack frame header. The register save areas must contain no padding.

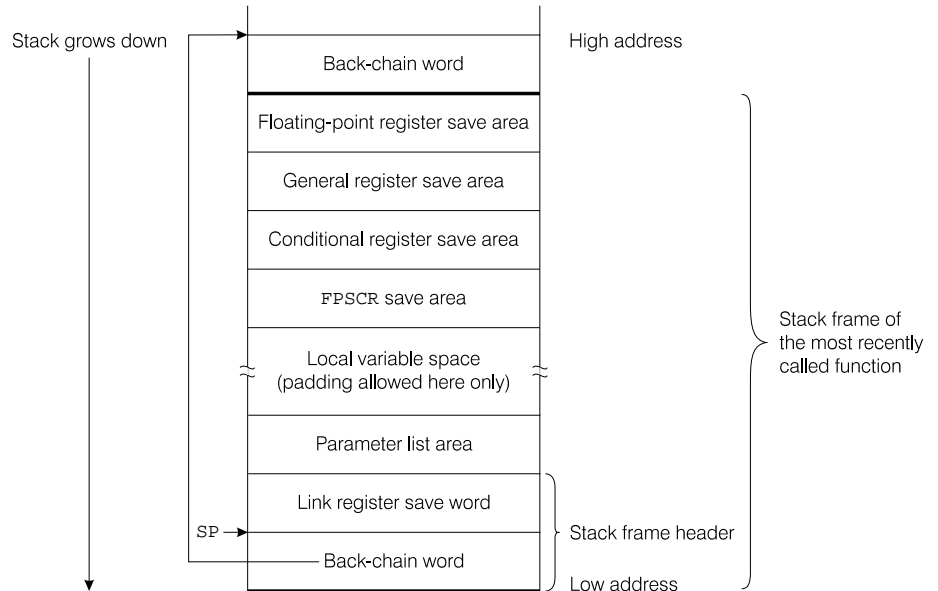


Figure 11.1 Standard Stack Frame

SP in Figure 11.1 denotes the stack pointer of the called function after that function has executed code establishing its stack frame. Note the following about the stack pointer:

- It maintains 16-byte alignment.
- It points to the “back-chain” word (the first word of the lowest allocated stack frame). This forms a linked list of stack frames.
- It is decremented by the called function in the function’s prolog, if required, and restored immediately prior to return (that is, before the **blr** or **blr1** instruction that returns to the caller).

11.2 Register Usage and Naming Conventions

The PowerPC provides 32 word-sized (32-bit) general-purpose registers, 32 double-word-sized (64-bit) floating-point registers, and several special-purpose registers.

All the registers are global to all functions in a running program.

Table 11.1 PowerPC Processor Registers

Register Name	Status	Usage
%r0	Volatile	Language-specific purposes
%r1	Dedicated	Stack frame pointer, always valid
%r2	Dedicated	Reserved for system use
%r3 – %r4	Volatile	Parameter passing and return values
%r5 – %r10	Volatile	Parameter passing
%r11 – %r12	Volatile	Language-specific purposes
%r13	Reserved	Small data area pointer
%r14 – %r30	Non-volatile	Local variables
%r31	Non-volatile	Local variables or “environment pointer”
%f0	Volatile	Language-specific purposes
%f1	Volatile	Parameter passing and return value
%f2 – %f8	Volatile	Parameter passing
%f9 – %f13	Volatile	Scratch
%f14 – %f31	Non-volatile	Local variables
CR0	Volatile	Condition Register fields, each four bits wide (Bit 6: Floating-point invalid operation exception)
CR1	Volatile	
CR2	Non-volatile	
CR3	Non-volatile	
CR4	Non-volatile	
CR5	Volatile	
CR6	Volatile	

Table 11.1 PowerPC Processor Registers (Continued)

Register Name	Status	Usage
CR7	Volatile	
LR	Volatile	Link Register
CTR	Volatile	Count Register
XER	Volatile	Fixed-Point Exception Register
FPSCR0 – 23	Volatile	Floating-Point Status and Control Register
FPSCR24 – 31	Modifiable	(Exception-enable and rounding-control bits)

Nonvolatile registers Registers %r1, %r14 through %r31, and %f14 through %f31 are non-volatile (they “belong” to the calling function). A called function saves the values in these registers before changing them and restores their values before returning.

Volatile registers Registers %r0, %r3 through %r12, %f0 through %f13, and special-purpose registers LR, CTR, and XER are volatile (not preserved across function calls).

Reserved register Register %r2 is reserved for system use and must not be changed by application code. On Solaris systems, this register points to thread-local storage. If small-data code generation is enabled, this register contains the address of the base of the .sdata2/.sbss2 sections.

Small-data area pointer Register %r13 is the *small-data area pointer*. If an executable uses 16-bit offset addressing relative to %r13 to reference data in the small-data area, the process start-up code for that executable must load the base of the small-data area into %r13. The base of the small-data area is the value of _SDA_BASE_, a loader-defined symbol. Shared objects must not alter the value in %r13. For additional information about small-data support, see §11.5.4: *Small-Data Sections*.

Environment pointers Languages that require environment pointers use register %r31 for that purpose.

CR: Condition register Condition-register fields CR2, CR3, and CR4 are non-volatile (value on entry must be preserved on exit); the rest are volatile (value does not need to be preserved).

FPSCR: The VE, OE, UE, ZE, XE, NI, and RN (rounding mode) bits of the FPSCR are non-volatile, except that they can be changed by a called function (for example, `fpsetround()`) that has the documented effect of changing them.
Floating-point status and control register The rest of the FPSCR is volatile.

11.2.1 Registers Used in the Standard Calling Sequence

The following registers and bits have assigned roles in the standard calling sequence: `%r1`, `%r3` through `%r10`, `%f1` through `%f8`, `CR1`, and `LR`. Their roles are discussed in the following sections.

Stack pointer: %r1 The stack pointer (stored in `%r1`) maintains 16-byte alignment, always points to the lowest allocated valid stack frame, and grows toward low addresses. The contents of the word at that address always point to the previously allocated stack frame.

The PowerPC architecture supports dynamic stack allocation within a stack frame for those languages that require it. (C does not require dynamic stack allocation.) A called function can, if necessary, decrement the stack pointer; this enables programs written in languages that need dynamic stack frame sizes to call C functions, and vice versa.

See the **ABI Supplement** for more information.

Registers %r3 – %r10 and %f1 – %f8 Because these registers can be modified across function invocations, the calling function presumes that they have been destroyed. These volatile registers are used for passing parameters to the called function; see §11.3: *Parameter Passing* for more information.

Registers `%r3`, `%r4`, and `%f1` are used to return values from the called function; see §11.6: *Return Values* for more information.

Floating-point invalid exception: CR1 Field `CR1` (bit 6), the “floating-point invalid exception” field, is set by the caller of a variable-argument-list function.

Link register: LR The link register contains the address to which a called function normally returns. Functions return by means of a `blr` or `blrl` instruction, so `LR` is non-volatile across function calls.

11.2.2 Functions Called During Signal Handling

Signals can interrupt processes. Functions called during signal handling have no unusual restrictions on their use of registers. If a signal-handling function returns, the process resumes its original execution path with all registers restored to their original values. Thus, programs and compilers can freely use all registers listed in Table 11.1 (except those reserved for system use) without the danger of signal handlers inadvertently changing their values.

11.2.3 Register Save Areas

Before changing the value in any non-volatile floating-point register, general register, condition-register field, or FPSCR bit, a function must first save the values in that and other registers, fields, or bits in the relevant register save area, as follows:

- **Any non-volatile floating-point register $\%fn$:**
The function must save the values in $\%fn$ and all higher numbered floating-point registers, as they were at the time of entry to the function, in order, in consecutive doublewords in the floating-point register save area. The value of $\%f31$ must be stored in the doubleword immediately below the word addressed by the back chain.
- **Any non-volatile general register $\%rn$:**
The function must save the values in $\%rn$ and all higher numbered general registers as they were at the time of entry to the function, in order, in consecutive words in the general register save area. The value of $\%r31$ must be stored in the word immediately below the floating-point register save area.
- **Any non-volatile field in the condition register:**
The function must save the values in all the non-volatile fields of the condition register at the time of entry to the function in the CR save area.
- **Any non-volatile bits in the FPSCR:**

The function must save the values in all the non-volatile bits of the FPSCR, as they were at the time of entry to the function, in the FPSCR save area.

11.3 Parameter Passing

A maximum of eight words of scalar parameters can be passed in general purpose registers %r3 through %r10, and a maximum of eight floating-point arguments can be passed in floating-point registers %f1 through %f8. If the number of parameters passed is less than the maximum, the unneeded registers are not loaded; they contain undefined values on entry to the called function.

If the arguments passed from a function do not fit in the registers provided (eight general-purpose and eight floating-point), the function must allocate space for arguments in its stack frame. The function should allocate only enough space to hold the arguments that do not fit into registers.

11.4 Variable Argument Lists

When you program for the PowerPC, you cannot assume that all arguments are passed on the stack and appear there in increasing order. Functions written with these assumptions do not work on PowerPC-based systems, because the PowerPC allows most arguments to be passed in registers. To write portable C and C++ programs, use header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists, per the ANSI C Standard.

A caller of a function that takes a variable argument list must set condition register bit 6 to 1 if the called function passes one or more arguments in the floating-point registers.

Note: If the called function does not pass arguments in the floating-point registers, the caller should clear CR bit 6.

See toggle `FP_varargs` for more information.

11.5 Application Binary Interface

This section documents aspects of the High C/C++ compiler that conform to the **PowerPC Embedded Application Binary Interface** (Version 1.0, Motorola, 1995). These include the following:

- Global Offset Table (GOT)
- Procedure Linkage Table (PLT)
- Small-Data Sections
- Tags

11.5.1 Overview of Global Offset Table and Procedure Linkage Table

The PowerPC uses the Global Offset Table and Procedure Linkage Table techniques to redirect a program's position-independent address calculations and function calls, respectively, to absolute locations.

For a more detailed description of the PowerPC's dynamic relocation techniques, refer to the ELF section in the **AT&T UNIX System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools** (also referred to as the **AT&T UNIX VR4 Programmer's Guide**).

11.5.2 Global Offset Table

Programs access non-local symbols via the *global offset table* (the GOT). The GOT redirects position-independent symbol references to absolute locations. Executable files and shared object files have separate global offset tables. The GOT section of an object file resides in the ELF `.got` section.

A GOT holds absolute addresses in private data. An entry in the GOT provides direct access to the absolute address of a symbol without compromising position-independence and shareability. A program references its GOT using position-independent addressing and extracts absolute values.

Initially, the GOT holds information as required by its relocation entries. When the loader creates memory segments for a loadable object file, it processes all relocation entries, some of which refer to the GOT. The loader does the following before passing control to the running process:

- determines the relocation entries' associated symbol values
- calculates their absolute addresses
- sets the GOT entries to the proper values

The symbol `_GLOBAL_OFFSET_TABLE_` accesses the global offset table. This symbol can reside in the middle of the GOT section, allowing positive and negative subscripts into the array of addresses.

These words in the global offset table are reserved:

- `_GLOBAL_OFFSET_TABLE_[-2]` contains the tag word for the instruction that follows.
- `_GLOBAL_OFFSET_TABLE_-1]` contains a `blrl` instruction.
- `_GLOBAL_OFFSET_TABLE_[0]` holds the address of the dynamic structure, referenced with the symbol `_DYNAMIC`.
- `_GLOBAL_OFFSET_TABLE_[1]` and `_GLOBAL_OFFSET_TABLE_[2]` are reserved for future use.

11.5.3 Procedure Linkage Table

Programs access non-local function calls via the *procedure linkage table* (the PLT). The procedure linkage table resides in the ELF `.plt` section.

The PLT redirects position-independent function calls to absolute locations. Executable files and shared object files have separate procedure linkage tables.

The PLT is not initialized in the executable or shared object file. The linker reserves space for the PLT. The loader then initializes the PLT and manages it according to the loader's own needs, with the following constraints:

- The PLT's first 18 words (72 bytes) are reserved for use by the loader. There must be no branches from the executable or shared object into these words.
- If the executable or shared object requires n entries in the PLT, the linker reserves $3n$ words following the 18 reserved words. The first $2n$ of these words are the PLT entries themselves. The remaining n words are reserved for use by the loader.

11.5.4 Small-Data Sections

High C/C++ supports *small-data sections*. When the compiler allocates storage for data in a small-data section, references to the data are faster, because such data can be addressed with 16-bit signed offsets from the base of the small-data section.

*Number of
instructions used to
access small-data
sections*

If a non-**auto** variable, for example `var`, is assigned to a small-data section, the compiler can use a single instruction to load or store `var` or take `var`'s address. This single instruction is called *small-data code*. Small-data code is more efficient than a regular access instruction for accessing non-**auto** variables. Normally, the compiler either uses a two-instruction sequence to access data, or loads data from memory using register indirect access, which uses three instructions. Either of these code sequences involves more overhead than small-data code.

*Referencing
variables in
small-data sections*

Variables assigned to a small-data section can be referenced by small-data code because the base address of each small-data section is contained in a dedicated register that points to the data section throughout execution of the program. Therefore, when a variable is referenced, only its offset needs to be loaded.

*Size limits of
small-data
variables*

Because the size of a small-data section is limited, the compiler assigns only variables up to a certain size (which you specify) to a given small-data section. The compiler allocates storage for defined variables of a suitable size in predefined small-data sections. The compiler assumes that variables that are declared but not defined (for example, variables declared **extern**)

are assigned to one of the small-data sections. That is, small-data code is generated for such a variable, even though the variable is not defined.

Where to specify small-data variables You must make sure that a variable is assigned to one of the small-data sections *in the compilation unit where the variable is defined*. At link time, the linker ensures that variables referenced with small-data code are, in fact, in a small-data section.

11.5.4.1 Assigning Data to Small-Data Sections

You control the assignment of data to a small-data section with pragmas `Push_small_data` and `Pop_small_data`. Pragma `Push_small_data` designates the small-data section into which variables declared in a given block of code are stored, and how big these variables can be. Pragma `Pop_small_data` “pops” the most recent `Push_small_data` specification and restores the previous small-data specification. See the entries for these pragmas in §5.2: *Pragma Reference* for information about their syntax and usage.

Setting small-data specifications via the command line The easiest way to use small-data sections is to employ the same setting for pragma `Push_small_data` for all compilation units, and invoke it at the command line. For example:

```
hc file.c -Hpragma="Push_small_data(8;12)" -c
```

or

```
hc file.c -Hpragma="Push_small_data(sdata0,8;
sdata2,12)" -c
```

Setting small-data specifications in source code For finer-grained control, you can bracket blocks of source code with pragmas `Push_small_data` and `Pop_small_data`.

Note: To avoid link errors, put pragma `Push_small_data` in any header file that contains the declaration of variables shared among compilation units.

11.5.4.2 More About Small-Data Sections

Pragma `Push_small_data` accepts several keywords or arguments that you use to assign **const** and non-**const** data to specified small-data sections. These keywords are `sdata0`, `sdata`, and `sdata2`. (See the

reference entry for pragma `Push_small_data` in §5.2: *Pragma Reference* for details on how to use these keywords.) What follows is some additional information about these small-data sections.

Keyword `sdata0` specifies small-data sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0`. These sections must start at location 0 after the final link, because instructions referencing variables in these sections use register `r0` as their base register. These special sections hold initialized and uninitialized data, respectively, that contributes to the program memory image, and whose addresses are all within a 16-bit signed offset of address 0. The combined size of sections `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0` must not exceed 32K.

Keyword `sdata` specifies small-data sections `.sdata` and `.sbss`. Instructions referencing these sections use a dedicated base register whose contents remain unchanged throughout program execution. The compiler uses section `.sbss` for uninitialized non-**const** variables and `.sdata` for initialized non-**const** variables. The combined size of sections `.sdata` and `.sbss` must not exceed 64K.

Keyword `sdata2` specifies small-data sections `.sdata2` and `.sbss2`. Instructions referencing these sections use a dedicated base register whose contents remain unchanged throughout program execution. The compiler uses `.sbss2` for uninitialized **const** variables and `.sdata2` for initialized **const** variables. The combined size of sections `.sdata2` and `.sbss2` must not exceed 64K.

11.5.4.3 Scope of Small-Data Specifications

Until it is explicitly respecified, a specification for a given small-data section applies for all subsequent variable definitions *in the same compilation unit*, even if a different `Push_small_data` specification (or no specification) is in effect at the point where you define a variable. For example:

```
#pragma Push_small_data(8) // Setting is (sdata,8).
                           // Assign all non-const
                           // variables of size <= 8
                           // to section .sdata

extern double i, j;
#pragma Pop_small_data(); // Setting is still
...                       // (sdata,8) for i and j
#pragma Push_small_data(4) // Setting is (sdata,4)
```

```

...
double i = 7.0;           // Put in section .sdata
double j;                 // Put in section .sbss

```

Note that the `Push_small_data` specification at the definition of `i` and `j` was overridden in favor of the specification that was in effect when `i` and `j` were declared.

The compiler assigns a variable to the small-data section that was in effect when the variable was defined. The current small-data section has no effect on variables that are declared but not defined (for example, variables declared **extern**), because the small-data section, if any, to which they are assigned must be specified in the compilation unit where they are defined. For example, a module could contain the following code:

```

#pragma Push_small_data(8)
extern const a,b,c;
extern char *p;
extern int **q;
#pragma Pop_small_data()

```

even though `a`, `b`, and `c` might be assigned to one small-data section and `p` and `q` to another.

11.5.4.4 Using Keywords `_Near` and `_Far` with Small-Data Specifications

Effect of using keyword `_Near` You can use the **`_Near`** keyword (in normal non-ANSI mode) to designate that a variable should be assigned to a small-data section. For example:

```

extern _Near int i;
extern const char * _Near i;

```

Be sure, however, to define the variable somewhere where a small-data specification is in effect.

Note: Be careful with your syntax. If you write:

```
extern _Near const char * i;
```

this makes `i` a pointer to a **`const char`** variable that is assigned to a small-data area, but has no effect on where `i` itself is stored.

*Limitations of using keyword **_Near*** The use of **_Near** by itself does not assign a variable to a particular small-data area. For example, assume that you have declared variable `var` to be of type **_Near**, and that you did not previously declare the variable with a small-data specification in effect. Assuming that a small-data specification is in effect at the definition, the compiler places `var` in the currently specified small-data section. Under these conditions, the compiler ignores the size criterion of the current specification. For example:

In a header file:

```
extern _Near int i;
```

In a `.c` file:

```
#pragma Push_small_data(2); // Setting is
                             // .sdata/.sbss,2
int i = 7; // Because of the _Near attribute,
           // put i in .sdata, even though
           // its size is greater than 2.
```

*Effect of using keyword **_Far*** **Note:** If you use **_Far** in a variable declaration, a small-data specification has no effect on the variable.

11.5.4.5 Limitations of Small-Data Specifications

Limitation on multiple definitions of global variables At global scope, you cannot define variables subject to small-data specification more than once. For example:

```
int i;
```

defines `i`. There cannot be more than one such definition if a small-data specification is in effect.

The small-data area occupies the boundary between initialized and uninitialized data in the data segment of a file. For shared objects, only data in the local scope can be assigned to a small-data section. For executables, data in both global and local scopes can be assigned to a small-data section.

11.5.4.6 Using Symbols to Access Small-Data Sections

The base address of the small-data area for sections `.sdata` and `.sbss` is represented by the symbol `_SDA_BASE_`. Register GPR13 contains the value of `_SDA_BASE_`. In the case of a shared object, the value of the base address

corresponds to the base address of the global offset table, represented by the symbol `_GLOBAL_OFFSET_TABLE_`. (See §11.5.2: *Global Offset Table*.)

The base address of the small-data area for sections `.sdata2` and `.sbss2` is represented by the symbol `_SDA2_BASE_`. Register GPR2 contains the value of `_SDA2_BASE_`.

11.5.5 Accessing Symbols with Assembler Identifier Attributes

Table 11.2 lists the assembler identifier attributes, which indicate different relocation options. The identifier attributes derive information from the symbols, in order to specify access to the GOT, PLT, small-data sections, etc. The derived information can be any of the following:

- Upper and lower half-words from of the value of the symbol
- Base address of the section containing the symbol
- Offset of the symbol in the section
- Index of the symbol's information in the symbol table and/or string table

(Note that this list is not exhaustive.)

For more information about assembly-language programming for the PowerPC, see the MetaWare **ELF Assembler User's Guide for PowerPC**. Refer to §11.5.4: *Small-Data Sections* for more information on small-data processing.

Table 11.2 Assembler Identifier Attributes

Attribute	Description
@got	Address of the Global Offset Table (GOT) entry for the identifier
@h	Upper half of the identifier address
@ha	Upper half of the identifier address, adjusted so the lower half can be used by instructions that interpret the lower half as signed
@l	Lower half of the identifier address
@local	Address of a local identifier, as opposed to a global identifier of the same name

Table 11.2 Assembler Identifier Attributes (Continued)

Attribute	Description
@off	Offset of the identifier in the section where the identifier resides; causes the assembler to generate a relocation entry
@plt	Address of a function's Procedure Linkage Table (PLT) entry
@sda	Offset of an identifier in the .sdata or .sbss section
@sda2	Offset of an identifier in the .sdata2 or .sbss2 section
@sda2i	Offset of a pointer to an identifier in the .sdata2 or .sbss2 section
@sdai	Offset of a pointer to an identifier in the .sdata or .sbss section
@sdax	Offset of the identifier from the base address of the .sdata section
@sdaxr	Combination of the base register for accessing the identifier and the offset of the identifier in the .sdata or .sbss section
@sect	Base address of the section in which the identifier is stored
@sectoff	Offset of the identifier; same as @off
@stridx	Index of the entry for the identifier in the ELF string table
@symidx	Index of the entry for the identifier in the ELF symbol table

11.5.6 Tag Support

The compiler and linker support tags, as specified in the **PowerPC Embedded Application Binary Interface**. Tag support is implemented through toggles `Save_incoming_regs`, `Tag_table`, and `Tag_table_in_text`. For details on the use of these toggles, refer to §4.2: *Toggle Reference*.

Tags aid in the debugging of optimized code. Tags help to determine the contents of non-volatile registers, as they were when a function was entered. Given the address of the next instruction to be executed and the tag, if any, applicable to that address, a debugger or exception handler can determine the register contents upon function entry.

With tags, you can determine which stack frame is associated with a section of code and which non-volatile registers at the time of entry to the function are within the register save areas, rather than in the registers themselves.

A full discussion of tags and how they are implemented is beyond the scope of this manual. Refer to the **ABI Supplement** for more information on this topic. (See the note at the beginning of this chapter for information about the **ABI Supplement**.)

11.6 Return Values

Table 11.3 specifies how function values are returned on the PowerPC.

Table 11.3 *PowerPC Function Return Values*

Function Return Type	Returned in Register	Comment
float	%f1	
double		
int	%r3	Returned as unsigned or signed integer (as appropriate), zero- or sign-extended to 32 bits if necessary
long		
enum		
short		
char		
Pointer to any type		
long long	%r3 and %r4	Returned with the lower addressed word in %r3 and the higher addressed word in %r4
unsigned long long		
struct (≤ 8 bytes)	%r3 and %r4	See §11.6.1: <i>Returning Structures and Unions up to Eight Bytes in Size</i>
union (≤ 8 bytes)		
long double	Storage buffer	Address of this buffer is passed as a hidden argument in %r3; see §11.6.2: <i>Values Returned in a Storage Buffer</i>
struct		

11.6.1 Returning Structures and Unions up to Eight Bytes in Size

If the size of a **struct** or **union** is less than or equal to eight bytes, it is returned in general registers %r3 and %r4 as if the following steps had occurred:

1. The **struct** or **union** was first stored in an eight-byte aligned memory area.
2. The low-addressed word was loaded into %r3.
3. The high-addressed word was loaded into %r4.

Bits beyond the last member of the **struct** or **union** are not defined.

11.6.2 Values Returned in a Storage Buffer

If the function return type is either a **struct** that does not meet the requirements for being returned in registers, or a **long double**, it is returned in a storage buffer allocated by the caller. The address of this buffer is passed as a hidden argument in %r3.

11.7 Prolog and Epilog Code

A function's prolog and epilog code establishes the environment needed by the body of the function.

- The *prolog* establishes a stack frame, if necessary, and can save any non-volatile registers the function uses.
- The *epilog* generally restores registers that were saved in the prolog code, restores the previous stack frame, and returns to the caller.

If a function establishes a stack frame, it must use one of the `Store Word with Update` instructions to update the back-chain word of the stack frame atomically with the stack pointer (`%r1`).

- For small stack frames (32K or less), this update can be accomplished with a `Store Word with Update` instruction with an appropriate negative displacement.
- For larger stack frames, the prolog computes the two's complement of the frame's size (with `addis` and `addi` or `ori` instructions), loads a volatile register with that size, then issues a `Store Word with Update Indexed` instruction.

PowerPC `Load Multiple` and `Store Multiple` instructions should not be used for the following reasons:

- On little-endian implementations, they cause alignment exceptions.
- On some big-endian implementations, they are slower than the register-at-a-time saves. (However, this might not be true for some embedded systems.)

Inline code can be used to save or restore non-volatile general or floating-point registers that the function uses. However, if there are many registers to save or restore, it might be more efficient to call one of the system subroutines. (See the **ABI Supplement** for details.)

A position-independent function that makes external data references should load a pointer to the GOT into a non-volatile register. (This load can be omitted if the function makes no external data references.) If the function's only external data references are within conditional code, the function can defer loading the GOT pointer until the pointer is needed.

Example 11.1 and Example 11.2 show the prolog and epilog code generated for a simple leaf function and for a non-leaf function, respectively.

Example 11.1 Prolog and Epilog for a Simple Leaf Function

```
/* ----- C code ----- */
int inc(int i) {
    return ++i;
}
```

```

/* ----- Assembly code ----- */
    .text
    .align 2
    .globl inc
inc:
    addic    %r3,%r3,1 ! add 1 to arg reg, %r3
    bclr     20,0      ! return to caller
    .type inc, @function
    .size inc, . - inc

```

Example 11.2 Prolog and Epilog for a Simple Non-Leaf Function

```

/* ----- C code ----- */
int add_one(int i) {
    return inc(i);
}

/* ----- Assembly code ----- */
    .globl add_one
add_one:
    mflr     %r0          ! get link register value in %r0,
    stw      %r0,+4(%r1)  ! save it in previous stack frame
    stwu     %r1,-16(%r1) ! allocate stack in 16-byte
                        ! increment
    bl       inc          ! call a function (pass argument
                        ! and return value in %r3)
    addic     %r1,%r1,16   ! deallocate allocated stack
    lwz      %r0,+4(%r1)  ! load link register value back
                        ! in %r0,
    mtlr     %r0          ! and move to link register
    bclr     20,0         ! return to caller
    .type add_one, @function
    .size add_one, . - add_one

```
