SYSTEM V

APPLICATION BINARY INTERFACE

SPARC™ Version 9 Processor

Supplement

May 17, 1996

Delta Document 1.33x

DRAFT

SPARC International Confidential

Revision History

Rev	Date	Author	Description
-0	10/01/91	HaL ABI Working Group	First Draft
-0.0	10/15/91	HaL ABI Working Group	Added Preface
-1.00	01/02/92	SI ABI Working Group	First Review
-1.10	02/02/92	SI ABI Working Group	Expanded calling convention/Minor editing changes
-1.20	03/09/92	SI ABI Working Group	Updated user traps and chapter 6.
-1.30	02/22/93	SI ABI Working Group	Updated user traps, added a relocation type,ELF flags
-1.31	04/06/93	SI ABI SIG	Deleted passing of small structures and unions in registers
-1.32	01/23/96	Aspen Group	Shortning of int to 32 bits.
1.33x	05/17/96	Terrence Miller	rough inclusion of changes by V9SCD meetings

3. LOW-LEVEL SYSTEM INFORMATION

3.1. Machine Interface

3.1.1. Processor Architecture

The SPARC™ Architecture Manual, Version 9 defines the processor architecture. Programs intended to execute directly on the processor use the instruction set, instruction encodings, and instruction semantics of the architecture. Four points deserve explicit mention.

- A SPARC V9 ABI conforming program may not use the IMPDEP1 and IMPDEP2 instructions.
- A program may assume all other documented non-privileged instructions exist
- A program may assume all other documented non-privileged instructions work.
- A program may assume that all documented unrestricted ASI's work.
- A program may use only the non-privileged instructions defined by the architecture, with the exception of IMPDEP1 and IMPDEP2.

In other words, from a program's perspective, the execution environment provides a complete and working implementation of the non-privileged part of the SPARC V9 architecture. Although the IMPDEP1 and IMPDEP2 instructions are part of the V9 architecture, they may not be used by V9 ABI conforming programs because their behavior is undefined.

This does not imply that the underlying implementation provides all instructions in hardware, only that the instructions perform the specified operations and produce the specified results. The ABI neither places performance constraints on systems nor specifies what instructions must be implemented in hardware.

Some processors might support the SPARC V9 architecture as a subset, providing additional instructions or capabilities. Programs that use those capabilities explicitly do not conform to the SPARC V9 ABI. Executing those programs on machines without the additional capabilities gives undefined behavior.

NOTE

For performance reasons it is suggested that the FLUSH instruction not be used. The [TBD-library] routine "flush_instr_mem" is the preferred way to flush instruction memory.

It is suggested that the instructions marked as "deprecated" in "The SPARC(TM) Architecture Manual, Version 9" not be used. These instructions may exhibit poor performance in some Version 9 implementations of the architecture and may not be available in future versions of the architecture.

3.1.2. Data Representation

3.1.2.1. Fundamental Types

Figure 3-1 shows the correspondence between ANSI C's scalar types and the processor's.

'igure 3-1: Scalar Types

Type	C	sizeof	Alignment (bytes)	SPARC
Integral	char signed char	1	1	signed byte
	unsigned char	1	1	unsigned byte
	short signed short	2	2	signed halfword
	unsigned short	2	2	unsigned halfword
	int signed int enum	4	4	signed word
	long signed long	8	8	signed extended-word
	unsigned int	4	4	unsigned word
	unsigned long	8	8	unsigned extended-word
Pointer	*any-type* * *any-type* (*) ()	8	8	unsigned extended-word
Floating-point	float	4	4	single-precision
	double	8	4 required 8 recommended	double-precision
	long double	16	4 required 16 recommended	quad-precision

A null pointer (for all types) has the value zero.

Double and quad-precision values occupy 1 and 2 extended words, respectively. Their natural alignment is the same, meaning their addresses are multiples of 8 and 16. Compilers should allocate independent data objects with the proper alignment; examples include global arrays of double-precision variables, FORTRAN COMMON blocks, and unconstrained stack objects. However, some language facilities (such as FORTRAN EQUIVALENCE statements) may create objects with only word alignment. Consequently, arbitrary double- and quad-precision addresses, such as pointers or reference parameters, might or might not be properly aligned. The system shall efficiently implement all LDDF(A), STDF(A), LDQF(A), and STQF(A) instructions with target addresses that are word aligned, even if they are not extended word aligned. Therefore, compilers should emit LDDF(A), STDF(A), LDQF(A), and STQF(A)instructions unless it is known at compile time that the target address is not extended word aligned.

Figure 3-1++ shows the correspondence between complex floating-point data types and the processor’s.

Figure 3-1++: Floating-point Complex Data Types

Type	Complex Type	sizeof	Alignment (bytes)	SPARC
Floating-point	single complex	8	4	single-precision (real) / single_precision (imaginary)
	double complex	16	4 required 8 recommend	double-precision (real) / double-precision (imaginary)
	quad complex	32	4 required 16 recommend	quad-precision (real) / quad-precision (imaginary)

3.1.2.2. Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component. The size of any object, including aggregates and unions, always is a multiple of the object’s alignment. An array uses the same alignment as its elements. Structure and union objects can require padding to meet size and alignment constraints.

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This may require *internal padding*, depending on the previous member.
- A structure’s size is increased, if necessary, to make it a multiple of the alignment. This may require *tail padding*, depending on the last member.

In the following examples, members’ byte offsets appear in the upper left corners.

Figure 3-2: Structure Smaller Than a Word

```
struct {
   char c;
};
```

Byte aligned, sizeof is 1

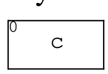


Figure 3-3: No Padding

```
struct {
   char c;
   char d;
   short s;
};
```

Halfword aligned, sizeof is 4

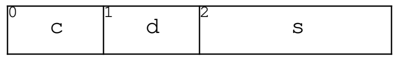


Figure 3-4: Internal Padding

```
union {
   char c;
   short s;
};
```

Halfword aligned, sizeof is 4

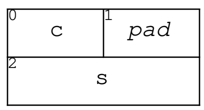


Figure 3-5: Internal and Tail Padding

```
struct {
   char c;
   long i;
   short s;
};
```

Extended word aligned, sizeof is 24

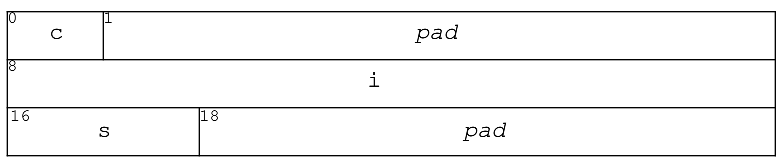
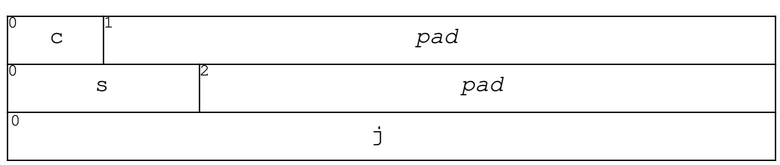


Figure 3-6: Union Allocation

```
union {
   char c;
   short s;
   long j;
};
```

Extended word aligned, sizeof is 8



3.1.2.3. Bit-Fields

C `struct` and `union` definitions may have *bit-fields*, defining integral objects with a specified number of bits.

Figure 3-7: Bit-Field Ranges

Bit-field Type	Width w	Range
signed char char unsigned char	1 to 8	-2^{w-1} to $2^{w-1}-1$ 0 to 2^w-1 0 to 2^w-1
signed short short unsigned short	1 to 16	-2^{w-1} to $2^{w-1}-1$ 0 to 2^w-1 0 to 2^w-1
signed int int unsigned int enum	1 to 32	-2^{w-1} to $2^{w-1}-1$ 0 to 2^w-1 0 to 2^w-1 0 to 2^w-1
signed long long unsigned long	1 to 64	-2^{w-1} to $2^{w-1}-1$ 0 to 2^w-1 0 to 2^w-1

"Plain" bit-fields always have non-negative values. Although they may have type `char`, `short`,`int`, `long`, or `enum` (which can have negative values), these bit-fields are extracted into an extended word with zero fill. Bit-fields obey the same size and alignment rules as other structure and union members, with the following additions.

- Bit-fields are allocated from left to right (most to least significant).
- A bit-field must entirely reside in a storage unit appropriate for its declared type. Thus a bit-field never crosses a unit boundary.
- Bit-fields may share a storage unit with other `struct`/`union` members, including members that are not bit-fields. Of course, `struct` members occupy different parts of the storage unit.
- Unnamed bit-fields' types do not affect the alignment of a structure or union, although individual bit-fields' member offsets obey the alignment constraints.

The following examples show `struct` and `union` members' byte offsets in the upper left corners; bit numbers appear in the lower corners.

Figure 3-8: Bit Numbering

`0x8899AABBCCDDEEFF`

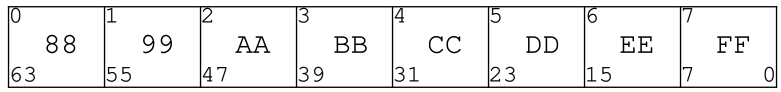


Figure 3-9: Left to Right Allocation

```
struct {
   long j:5;
   long k:6;
   long m:7;
};
```

Extended word aligned, sizeof is 8

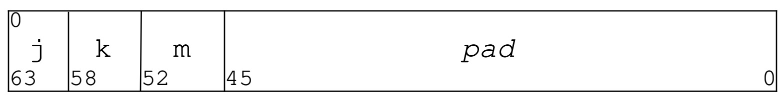


Figure 3-10: Boundary Alignment

```
struct {
   short s:9;
   long j:9;
   char c;
   short t:9;
   short u:9;
   char d;
};
```

Extended word aligned, sizeof is 8

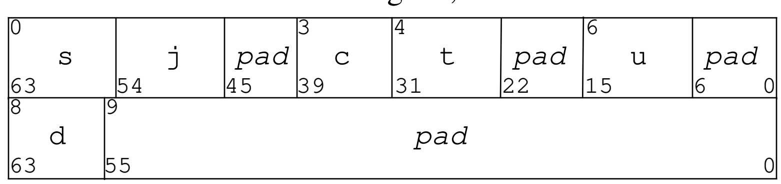


Figure 3-11: Storage Unit Sharing

```
struct {
   char c;
   short s:8;
};
```

Halfword aligned, sizeof is 2

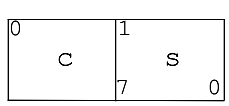


Figure 3-12: `union` Allocation

```
union {
   char c;
   short s:8;
};
```

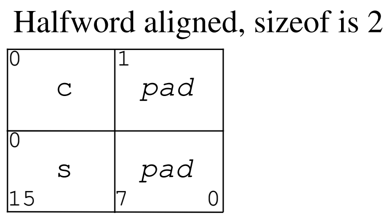
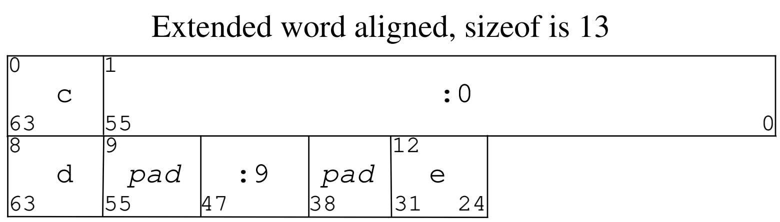


Figure 3-13: Unnamed Bit-fields

```
struct {
   char c;
   long :0;
   char d;
   short :9;
   char e;
   char :0;
};
```



As the examples show, `long` bit-fields (including `signed` and `unsigned`) pack more densely than smaller base types. One can use `char` and `short` bit-fields to force particular alignments, but `long` generally works better.

3.2. Function Calling Sequence

This section discusses the standard function calling sequence, including stack frame layout, register usage, parameter passing, etc. The system libraries described in Chapter 6 require this calling sequence.

NOTE

C programs follow the conventions given here. For specific information on the implementation of C, see “Coding Examples” in this chapter.

3.2.1. Registers and the Stack Frame

In SPARC V9 all floating-point registers and 8 integer registers are global to a running program, as the `save` and `restore` instructions do not affect them. All remaining integer registers are windowed: 24 are visible at any time, and sets of 24 overlap by 8 registers each. The `save` and `restore` instructions manipulate the windows as part of the normal function prologue and epilogue, making the caller's 8 *out* registers coincide with the callee's 8 *in* registers. Each window set also has 8 unshared *local* registers. Generally, each new frame on the dynamic call stack uses a new register window.

Brief register descriptions appear in Figures 3-14 and 3-15; more complete information appears later.

Figure 3-14: A Function's Window Registers

Type	Name			Usage
in		%i7	%r31	return address - 8 †
	%fp	%i6	%r30	frame pointer †
		%i5	%r29	incoming param †
		%i4	%r28	incoming param †
		%i3	%r27	incoming param, †
		%i2	%r26	incoming param, †
		%i1	%r25	incoming param, † (outgoing return value?)
		%i0	%r24	incoming param, † outgoing return value
local		%l7	%r23	local †
		%l6	%r22	local †
		%l5	%r21	local †
		%l4	%r20	local †
		%l3	%r19	local †
		%l2	%r18	local †
		%l1	%r17	local †
		%l0	%r16	local †
out		%o7	%r15	address of call instruction, ‡ temporary value
	%sp	%o6	%r14	stack pointer †
		%o5	%r13	outgoing param ‡
		%o4	%r12	outgoing param ‡
		%o3	%r11	outgoing param, ‡
		%o2	%r10	outgoing param, ‡
		%o1	%r9	outgoing param, ‡ (incoming return value?)
		%o0	%r8	outgoing param, ‡ incoming return value

Figure 3-15: A Function's Global Registers

Type	Name			Usage
		%g7	%r7	global (reserved for system)
		%g6	%r6	global (reserved for system)
		%g5	%r5	global ‡
global		%g4	%r4	global ?
		%g3	%r3	global ?
		%g2	%r2	global ?
		%g1	%r1	global ‡
		%g0	%r0	0
	%q60	%d60,d62		floating-point ‡
	%q56	%d56,d58		floating-point ‡
	%q52	%d52,d54		floating-point ‡
	%q48	%d48,d50		floating-point‡
	%q44	%d44,d46		floating-point ‡
	%q40	%d40,d42		floating-point ‡
	%q36	%d36,d38		floating-point ‡
	%q32	%d32,d34		floating-point ‡
floating-point	%q28	%d28,d30	%f28-f31	parameter ‡
	%q24	%d24,d26	%f24-f27	parameter ‡
	%q20	%d20,d22	%f20-f23	parameter ‡
	%q16	%d16,d18	%f16-f19	parameter ‡
	%q12	%d12,d14	%f12-f15	parameter ‡
	%q8	%d8,d10	%f8-f11	parameter ‡
	%q4	%d4,d6	%f4-f7	parameter, ‡ (return value?)
	%q0	%d0,d2	%f0-f3	parameter, ‡ return value
			%y	Y register ‡
			%ccr	condition code register ‡
special			%asi	(see below)
			%fprs	(see below)
			%fsr	(see below)

NOTE

Registers marked † above are assumed to be preserved across a function call. Registers marked ‡ above are assumed to be destroyed (volatile) across a function call.

In addition to a register window, each function has a frame on the run-time stack. This grows downward from high addresses, moving in parallel with the current register window. Figure 3-16 shows the stack frame organization.

Figure 3-16: Standard Stack Frame

Base	Offset	Contents	Frame
		unspecified . . . variable size	*High Addresses*
%fp+BIAS	+176	(if present) additional incoming argument slots	Previous
%fp+BIAS	+128	6 extended word argument slots	
%fp+BIAS	0	16 extended word save area	
%fp+BIAS	-1	unspecified . . . variable size	
%sp+BIAS	+176	(if needed) additional outgoing argument slots	Current
%sp+BIAS	+128	6 extended word argument slotss	
%sp+BIAS	0	16 extended word save area	
%sp+BIAS %sp	-1 0	volatile memory (do not use)	*Low Addresses*

BIAS = 2047

Several key points about the stack frame deserve mention.

- Every stack frame must be 16-byte aligned.
- Every stack frame must have a 16-extended-word save area for the *in* and *local* registers, in case of window overflow or underflow. This save area always must exist at %sp plus a BIAS of 2047 (0x7ff).
- Arguments that do not fit in the argument registers are passed on the stack.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define a maximum stack frame size, nor does it restrict how a language system uses the “unspecified” areas of the standard stack frame.

NOTE

The stack pointer is offset from the stack frame by a BIAS of 2047 (ox7ff). This BIAS permits stack frame references in the range of %fp-4096 to %fp+2047 and %sp+2047 to %sp+4095 to be made with only immediate offset addressing. By making the BIAS an odd number, the least significant bit of the stack pointer will be set and the register overflow and underflow handlers can easily distinguish a 64-bit register window from a 32-bit register window.

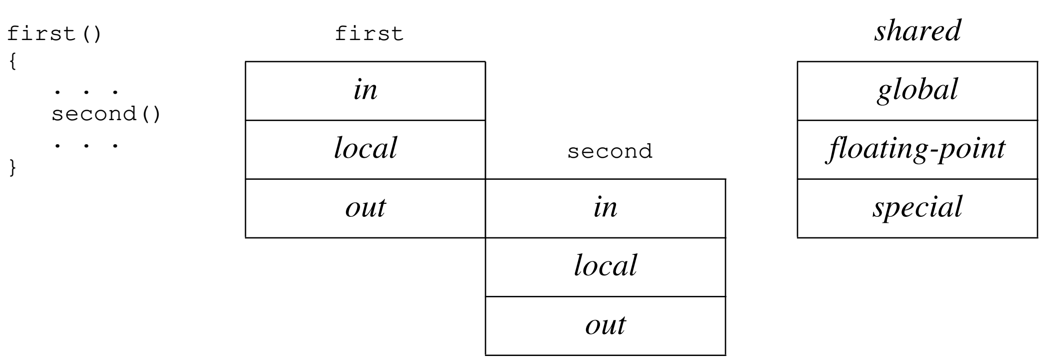
Across function boundaries, the standard function prologue shifts the register window, making the calling function's *out* registers the called function's *in* registers. It also allocates stack space, including the required areas of figure 3-16 and any private space it needs. The lowest 16 extended-words in the stack must—at all times—be reserved as the register save area. The example below illustrates this and allocates 176 bytes for the stack frame.

Figure 3-17: Function Prologue

```
second:
        save %sp, -176, %sp
```

For demonstration, assume a function named `first` calls `second`. The register windows for the two functions appear below.

Figure 3-18: Register Windows



As explained later, the function epilogue executes a restore instruction to unwind the stack and restore the register windows to their original condition.

NOTE Strictly speaking a function does not need the save and restore instructions if it preserves the registers as described below. Although some functions can be optimized to eliminate the save and restore, the general case uses the standard prologue and epilogue.

Some registers have assigned roles.

%sp or %o6	The *stack pointer* plus the stack BIAS determines the limit of the current stack frame, which is the address of the stack's bottommost, valid word. At all times the stack pointer plus the stack BIAS must point to a 16-byte aligned, 16 extended words window save area.
%fp or %i6	The *frame pointer* plus the stack BIAS is the address of the previous stack frame, which coincides with the word immediately above the current frame. Consequently, a function has registers with which it can access both ends of its frame. Incoming overflow arguments reside in the previous frame, referenced as positive offsets from the frame pointer plus the stack BIAS.
%i0 and %o0	*Integral and pointer return values* appear in %i0. A calling function receives values in the coincident *out* register %o0.
%i0,%i1,%i2,%i3 (%o0,%o1,%o2,%o3)	*Structure or union return values of size 32 bytes or less* appear in registers %i0, %i1, %i2 and %i3. A calling function receives values in the coincident *out* registers
%i7 and %o7	The *return address* is the location to which a function should return control. Because a calling function's *out* registers coincide with the called function's *in* registers, the calling function puts a return address in its own %o7, while the called function finds the address in %i7. Actually, the return address register holds the call instruction's address, normally making the return address %i7+8 for the called function. (every call instruction has a delay instruction.) Between function calls, %o7 serves as a scratch register.
%f0,%f1,%f2,%f3 (%d0, %d2) (%q0)	*Floating-point return values* appear in the floating-point registers. Single-precision values occupy %f0; double-precision values occupy %d0; quad-precision values occupy %q0. (Refer to the ***SPARC™*** *Architecture Manual, Version 9* for details on the register numbering scheme). Otherwise, these are scratch registers.

%f0 through %f7 (%d0 through %d6) (%q0 and %q4)	For non-C applications, *aggregate floating-point return values of 32 bytes or less* appear in the floating-point registers. FORTRAN single-complex values occupy %f0 and %f1; double-complex values occupy %d0 and %d2; quad-complex values occupy %q0 and %q4.
%i0 through %i5	*Incoming non-floating-point parameter slots* use up to 6 corresponding *in* registers. Arguments beyond the sixth extended-word appear on the stack.
%o0 through %o5	*Outgoing non-floating-point parameters slots* use up to 6 corresponding *out* registers. Arguments beyond the sixth extended-word appear on the stack.
%f1, %f3 through %f29, %f31 (%d0 through %d30) %q0 through %q28)	*Floating-point arguments* are passed in the floating-point registers.Unpromoted single-precision arguments are passed in the first 16 odd-numbere %f registers, Double-precision arguments are passed in registers %d0 through %d30 Quad-precision arguments are passed in registers %q0 through %q28 These registers are assumed **volatile** across the call.
%l0 through %l7	*Local registers* have no specified role in the standard calling sequence.
%d32 through %d62 (%q32through %q60)	These *floating-point registers* have no specified role in the standard calling sequence. They are assumed **volatile** across function calls
%g0	*Global register* 0 has no specified role in the standard calling sequence.
%g1, %g5	*Global registers* 1 and 5 have no specified role in the standard calling sequence. They are assumed **volatile** across function calls.
%g2, %g3, %g4	*Global registers* 2, 3, and 4 are reserved for application software. System software (including the libraries described in Chapter 6) **preserve** the registers' values for the application. Their use is intended to be controlled by the compilation system and must be consistent throughout the application.
%g6 and %g7	*Global registers* 6 and 7 are reserved for system software.
%ccr, %y	These *special registers* are **volatile** across function calls
%asi	The *address space identifier register* by default holds the value `ASI_PRIMARY_NOFAULT`. If modified, it must be restored to the default value before calling another function or returning.

%fsr	The RD, TEM and NS fields are **preserved** across function calls; the other fields are **volatile**. The **AEXC** bits may be set by a callee, but may not be cleared.
%fprs	The *floating point registers state* is intended for use by a threads interface. An application that uses %fprs may not work with a future threads interface. A threads interface may publish its own rules for use of %fprs.

With some exceptions given below, all registers visible to both a calling and a called function ‘belong’ to the called function. In other words, a called function may use all visible registers without saving their values before it changes them and without restoring their values before it returns. Registers in this category include *global* registers 1 and 5, **volatile** *floating-point* registers, *out* registers (for the calling function), *in* registers (for the called function), the Y register... Correspondingly, if a calling function wants to preserve such a register value across a function call, it must save the value and restore it explicitly. *Local* registers in each window are private. A called function should not change its calling function’s *local* or *in* registers, even though the registers may be visible temporarily. The exceptions are the stack pointer, %sp, *global* registers 2 through 4, 6 and 7. A called function is obligated to preserve the stack pointer for its caller.

Signals can interrupt processes [see `signal`(BA_OS)]. Functions called during signal handling have no unusual restrictions on their use of registers. Moreover, if a signal handling function returns, the process resumes its original execution path with registers restored to their original values. Thus programs and compilers may freely use all non-reserved registers, even *global* and *floating-point* registers, without the danger of signal handlers changing their values. The *address space identifier* register will be set to `ASI_PRIMARY_NOFAULT` on entry to the signal handler.

3.2.2. Function Argument Passing

It is convenient to describe parameter linkage in terms of an array. Conceptually, parameters are assigned into an array of extended words, left-to-right, with an occasional “hole” to satisfy alignment restrictions. Typically, most parameter values will be “promoted” from their memory locations into registers, and we hope that most calls will execute this way with less overhead.

The following diagram shows the correspondence between parameter registers and the “parameter array.”

Memory	Integral	Float	Double	Quad
%sp+BIAS+248		%f31	%d30	
%sp+BIAS+240		%f29	%d28	%q28
%sp+BIAS+232		%f27	%d26	
%sp+BIAS+224		%f25	%d24	%q24
%sp+BIAS+216		%f23	%d22	

%sp+BIAS+208		%f21	%d20	%q20
%sp+BIAS+200		%f19	%d18	
%sp+BIAS+192		%f17	%d16	%q16
%sp+BIAS+184		%f15	%d14	
%sp+BIAS+176		%f13	%d12	%q12
%sp+BIAS+168	%o5	%f11	%d10	
%sp+BIAS+160	%o4	%f9	%d8	%q8
%sp+BIAS+152	%o3	%f7	%d6	
%sp+BIAS+144	%o2	%f5	%d4	%q4
%sp+BIAS+136	%o1	%f3	%d2	
%sp+BIAS+128	%o0	%f1	%d0	%q0

An “integral type” is any eight-bit char, sixteen-bit short, thirty-two bit int, sixty-four bit long, sixty-four bit long long, or a sixty-four bit pointer to any type.

A “floating type” is any thirty-two bit float, sixty-four bit double, or a one-hundred-twenty-eight bit long double.

Structures and unions are categorized only by their size and alignment requirement; structures up to sixteen bytes in size are passed more efficiently than structures larger than sixteen bytes.

To call a function with parameters, a calling function allocates a “parameter array” in its stackframe (see Figure 3-16), sufficiently large to pass all parameters in memory. However, some values are notstored in this array, but are passed only in registers; see below. In the description below, %i and %o register names are used according to context. See the descriptions of the SAVE and RESTORE instructions for the relationship between these.

Every register used to pass parameter values has a corresponding location, at a fixed offset, in the parameter array.

3.2.2.1. Integral and pointer arguments

Int, short, char, long, and long long types will be assigned to one 64-bit word in the parameter array. Types smaller than one 64-bit word, including char, short, and the thirty-two bit int, will be widened according to their signedness, and passed right-justified.

Any integral or pointer parameters assigned to locations %sp+BIAS+128 through %sp+BIAS+168 in the parameter array will be passed in registers %o0..%05. The corresponding locations in the parameter array will have undefined values. The corresponding %f/%d/%q register(s) will also be undefined.

3.2.2.2. Floating arguments

Each single-precision parameter value will be assigned to one 64-bit word in the parameter array, and right-justified within that word. Each double-precision parameter value will be assigned to one 64-bit word in the parameter array. Each long-double-precision parameter value will be assigned to two 64-bit words in the parameter array. Long doubles should belong-double-aligned, and a “hole” may be introduced into the parameter array to force alignment.

When a callee prototype exists, and does not indicate variable arguments, floating-point values assigned to locations %sp+BIAS+128 through %sp+BIAS+248 will be promoted into floating-point registers, as shown above.

When a callee prototype exists and a particular floating argument natches the “...” of a function with varialble arguments, floating values assigned to locations %sp+BIAS+128 through %sp+BIAS+168 will be promoted to %i0..%i5

When no prototype exists for a callee:

> Floating values assigned to locations %sp+BIAS+128 through %sp+BIAS+168 will be passed simultaneously in %i0..%i5 and %d0..%d10 (or %q0..%q8)
>
> Floating values assigned to locations %sp+BIAS+176 through %sp+BIAS+248 will be passed simultaneously in memory and in %f12..%d30

3.2.2.3. Structure and Union arguments

Structure or union types up to eight bytes in size are assigned to one parameter array word, and align to eight-byte boundaries.

Structure or union types larger than eight bytes, and up to sixteen bytes in size are assigned to two consecutive parameter array words, and align according to the alignment requirements of the structure or at least to an eight-byte boundary..

Structure or union types are always left-justified, whether stored in registers or memory. Any structure or union type assigned into locations %sp+BIAS+128 through %sp+BIAS+168 will be promoted to the corresponding %o register.

Note that a sixteen-byte structure assigned to locations %sp+BIAS+168 and %sp+BIAS+176 will be “split,” as the contents of location %sp+BIAS+168 will be promoted to %o5. This is the only situation where a value will be split between registers and memory.

Structures larger than two words are copied by the caller are passed indirectly; the caller will pass the address of a correctly aligned structure value. This sixty-four bit address will occupy one word in the parameter array, and may be promoted to an %o register like any other pointer value. The callee may modify the addressed structure.

The caller can omit the copy if such ommission can not be detected. That requires (at least) that:

- the original aggregate is already properly aligned,
- the original aggregate is not aliased,

- the original aggregate is not used after the call, and
- no language-specific semantics require the copy.

3.2.2.4. Varaible Argument Lists

A function that expect a variable argument list typically uses the stdarg.h mechanism to process the list. That mechanism defines a va_list type that can be passe to another fuction. Figure XXX defines the va_list type.

3.2.3. Function Result Passing

Functions declared to return the void type do not return a value. All other functions return their values according to the following rules.

3.2.3.1. Integral and pointer return values

Integral and pointer return types are returned in integer register %o0. Functions returning integral and pointer return values always return an extended-word, expanding signed and unsigned bytes, halfwords, and words as needed.

3.2.3.2. Floating and complex return values

A return value of floating-point type is passed in %f0, %d0, or %q0 respectively. A return value of complex type is passed in pairs of %f0/%f1, %d0/%d2, or %q0/%q4, respectively, where the first register of the pair holds the real component and the second register holds the imaginary component.

3.2.3.3. Structure or Union return values

The caller allocates an area large enough to hold the return value, and passes a pointer to that area as an implicit first argument (of type pointer-to-data) to the callee. This implicit argument logically precedes the first actual argument, and is allocated according to normal argument passing rules (i.e. into %o0). The callee may modify the designated memory area at any time during its execution; the only requirement is that it hold the return value upon return. If the callee is terminated through any means other than a normal function return (e.g. through a call to the longjmp function), the contents of the memory area are undefined.

Note that the caller may pass a pointer to a program variable as long as it ensures that the above rules cannot cause violation of the program's proper semantics.

Note also that the caller is required to provide the implicit argument and a properly sized receiving area even if it does not wish to use the callee's function result. In that case, the caller may simply pass a pointer to a scratch area.

3.2.4. Examples of Argument Passing

All the following examples assume the caller sees a prototype for the callee.

3.2.4.1. *Integral and Pointer Arguments*

As mentioned, a function receives its first 6 integral and pointer arguments through the *in* registers, %i0 through %i5. Functions pass all integral arguments as extended-words, expanding signed or unsigned bytes, halfwords and words as needed. If a function call has more than 6 integral and pointer arguments the others go on the stack.

Figure 3-19: Integral and Pointer Arguments

Argument	Call	Caller	Callee
1	`g( char,`	`%o0`	`%i0`
2	`char,`	`%o1`	`%i1`
3	`short,`	`%o2`	`%i2`
4	`int,`	`%o3`	`%i3`
5	`char *,`	`%o4`	`%i4`
6	`int,`	`%o5`	`%i5`
7	`int,`	`%sp+BIAS+128`	`%fp+BIAS+128`
8	`void * );`	`%sp+BIAS+136`	`%fp+BIAS+136`

3.2.4.2. *Floating-Point Arguments*

The first floating-point arguments are passed in floating-point registers.

Figure 3-20: Floating-Point Arguments

Argument	Call	Caller	Callee
1	`h( float,`	`%f1`	`%f1`
2	`float,`	`%f3`	`%f3`
3	`double,`	`%d4`	`%d4`
4	`float,`	`%f7`	`%f7`
5	`double,`	`%d8`	`%d8`
6	`float,`	`%f11`	`%f11`
7	`long double,`	`%q12`	`%q12`
8	`float,`	`%f17`	`%f17`
9	`double,`	`%d18`	`%d18`
10	`long double );`	`%q20`	`%q20`

3.2.4.3. *An Example of Mixed Arguments*

Figure 3-20.5: Mixed Arguments

Argument		Caller	Callee
1	`f( char,`	`%o0`	`%i0`
2	`float,`	`%f3`	`%f3`
3	`short,`	`%02`	`%12`
4	`double,`	`%d6`	`%d6`
5	`int,`	`%o5`	`%i5`
6	`float,`	`%f13`	`%f13`
7	`long,`	`%sp+BIAS+184`	`%fp+BIAS+184`
8	`char *,`	`%sp+BIAS+192`	`%fp+BIAS+192`
9	`long,`	`%sp+BIAS+200`	`%fp+BIAS+200`
	`);`		

3.2.5. Examples of Result Passing

3.2.5.1. Functions Returning Scalars or No Value

A function that returns an integral or pointer value places its result in %i0; the calling function finds that value in %o0.

A floating-point return value appears in the floating-point registers for both the calling and the called function. Single-precision uses %f0; double-precision uses %d0; quad-precision uses %q0.

Functions that return no value (also called procedures or `void` functions) put no particular value in any return register. Those registers may be used as scratch registers, however.

A call instruction writes its own address into *out* register %o7. As usual for a control transfer instruction, the call instruction takes a delay instruction that is executed before the instruction of the called function. Because every instruction is one word long, the return address is the address of the call instruction plus 8. The value is %i7+8 for the called function and %o7+8 for the calling function. The following example returns the value contained in *local* register %l4.

Figure 3-23: Function Epilogue

```
jmpl %i7 + 8, %g0
restore %l4,0,%o0
```

If a function returns no value or if the return register already contains the desired value, the next epilogue would suffice.

Figure 3-24: Alternative Function Epilogue

```
jmpl %i7 + 8, %g0
restore %g0,0,%g0
```

3.3. Operating System Interface

3.3.1. Virtual Address Space

Processes execute in a 64-bit virtual address space e; this is mapped as both the primary and secondary address space. That s, all forms of ASI_PRIMARY and ASI_SECONDARY refer to the same address space. Memory management hardware translates virtual addresses to physical addresses, hiding physical addressing and letting a process run anywhere in the system's real memory. Processes typically begin with three logical segments, commonly called text, data and stack. As Chapter 5 describes, dynamic linking creates more segments during execution, and a process can create additional segments for itself with system services.

3.3.1.1. Page Size

Memory is organized by pages, which are the system's smallest units of memory allocation. Page size can vary from one system to another, depending on the processor, memory management unit and system configuration. Processes may call `sysconf`(BA_OS) to determine the system's current page size. The maximum page size for SPARC V9 is 1 MB.

3.3.1.2. Virtual Address Assignments

Conceptually, processes have the full 64-bit address space available. In practice, however, several factors limit the size of a process.

- The system reserves a configuration-dependent amount of virtual space.
- A tunable configuration parameter limits process size.
- A process whose size exceeds the system's available, combined physical memory and secondary storage cannot run. Although some physical memory must be present to run any process, the system can execute processes that are bigger than physical memory, paging them to and from secondary storage. Nonetheless, both physical memory and secondary storage are shared resources. System load, which can vary from one process execution to the next, affects the available amounts.

Figure 3-26: Virtual Address Configuration

0xffff ffff ffff ffff	Reserved . . .	*End of memory*
	Stack and dynamic segments . . .	
0	. . . Loadable segments	*Beginning of memory*

Loadable segments
: Processes' loadable segments may begin at 0. The exact addresses depend on the executable file format [see Chapters 4 and 5].

Stack and dynamic segments
: A process's stack and dynamic segments reside below the reserved area. Processes can control the amount of virtual memory allocated for stack space, as described below.

Reserved
: A reserved area resides at the top of virtual memory.

> NOTE
>
> Although application programs may begin at virtual address 0, they conventionally begin at 0x100000 (1 M), leaving the initial 1 M with an invalid address mapping. Processes that reference this invalid memory (for example by dereferencing a null pointer) generate an access exception trap, as described in the "Trap Interface" section of this chapter. A process may, however, establish a valid mapping for this area using the `mmap`(KE_OS) facilities.

As the figure shows, the system reserves the high end of virtual space with a process's stack and dynamic segments below that. Although the exact boundary between the reserved area and a process depends on the system's configuration, the reserved area shall not consume more than 8 exabytes (EB) from the virtual address space. Thus the user virtual address range has a minimum upper bound of 0x7fff ffff ffff ffff. Individual systems may reserve less space, increasing processes' virtual memory range. More information follows in the section "Managing the Process Stack".

> NOTE
>
> The effects of using load and store alternate instructions with address space identifiers other than ASI_PRIMARY and ASI_PRIMARY_NOFAULT are undefined.

3.3.2. Trap Interface

3.3.2.1. Hardware Trap Types

The operating system defines the following correspondence between hardware traps and the signals specified by `signal`(BA_OS).

Figure 3-27: Hardware Traps and Signals

Trap Name	Signal
instruction_access_exception	SIGSEGV,SIGBUS
instruction_access_MMU_miss	SIGSEGV
instruction_access_error	SIGBUS
illegal_instruction	SIGILL
privileged_opcode	SIGILL
fp_disabled	SIGILL
fp_exception_ieee_754	SIGFPE
fp_exception_other	SIGFPE
tag_overflow	SIGEMT
division_by_zero	SIGFPE
data_access_exception	SIGSEGV,SIGBUS
data_access_MMU_miss	SIGSEGV
data_access_error	SIGBUS
data_access_protection	SIGSEGV
mem_address_not_aligned	SIGBUS
privileged_action	SIGILL
async_data_error	SIGBUS
trap_instruction	see next table

The signal is sent only if no user trap handler is provided. See User Traps.

Two trap types, instruction_access_exception and data_access_exception, can generate two signals. In both cases, the “normal” signal is `SIGSEGV`. Nonetheless, if the access also causes some external memory error (such as parity error), the system generates `SIGBUS`.

Floating point instructions exist in the architecture, but they may be implemented either in hardware or software. If the fp_disabled or fp_exception_other trap occurs because of an unimplemented, valid instruction, the process receives no signal. Instead the system intercepts the trap, emulates the instruction, and returns control to the process. A process receives SIGILL for the fp_disabled trap only when the indicated floating-point instruction is illegal (invalid encoding, etc.).

3.3.2.2. Software Trap Types

The operating system defines the following correspondence between software traps and the signals specified by `signal`(BA_OS).

Figure 3-28: Software Trap Types

Trap Number	Signal	Purpose
0	unspecified	Reserved for the operating system
1	SIGTRAP	Breakpoints
2	SIGFPE	Division by zero
3	unspecified	Reserved for the operating system
4	unspecified	Reserved for the operating system
5	SIGILL	Range checking
6	none	Fix alignment
7	SIGFPE	Integer overflow
8	unspecified	Reserved for the operating system
9	SIGSYS	SVID system calls
10	unspecified	Reserved for the operating system
11	SIGSYS	SPARC-specific system calls
12	SIGSYS	Vendor-specific system calls
13	SIGSYS	OEM-specific system calls
14-15	unspecified	Reserved for the operating system
16-31	SIGILL	Send SIGILL signal
32	unspecified	Reserved for the operating system
33	unspecified	Reserved for the operating system
34	SIGILL	Return from deferred trap
35-127	unspecified	Reserved for the operating system

0 and 8 — Trap types 0 and 8 were used in some pre-V9 SPARC systems to implement operating system service routines. In V9 they are reserved.

NOTE

The ABI does not define the implementation of individual system calls. Instead, programs should use the system libraries that chapter 6 describes. Programs with embedded system call trap instructions do not conform to the ABI.

1 — A debugger can set a breakpoint by inserting a trap instruction whose type is 1.

2 — A process can explicitly signal division by zero with this trap.

3 — Trap type 3 was used in pre-V9 SPARC systems to ask the system to flush all its register windows to the stack. In V9 the `flushw` instruction can be used instead. The trap is reserved.

4 Trap type 4 was used in pre-V9 SPARC systems to cause the system to initialize *local* and *out* registers in all subsequent new windows either to zeros or values placed into them by the calling process. In V9 this behavior is required. The trap is reserved.

5 A process can explicitly signal a range checking error with this trap.

6 Executing a type 6 trap makes the operating system “fix” subsequent unaligned data references. Although the references still generate memory_address_not_aligned traps, the operating system handles the trap, emulates the data references, and returns control to the process without generating a signal. In this context a “data reference” is a load or store operation. Implicit memory references, such as control transfers, must always be aligned properly, and the stack must always be aligned as described elsewhere.

This trap is provided to ease porting of existing code. Its use in new code is deprecated. A user trap handler should be used instead. If a user trap handler for UT_MEM_ADDRESS_NOT_ALIGNED is installed, it takes precedence.

7 A process can explicitly signal integer overflow with this trap. Either a positive or a negative value can cause overflow.

9 Operating system service routines specified in the SVID are implemented using this trap type.

10, 14, 15 The operating system reserves these traps for its own use. Programs that use them do not conform to the ABI.

11 SPARC-specific operating system service routines are implemented using this trap type.

12 Vendor-specific operating system service routines are implemented using this trap type.

13 OEM-specific operating system service routines are implemented using this trap type.

32 Trap type 32 was used in pre-V9 SPARC systems to copy the icc integer condition codes from the PSR register to global register %g1. In V9 the CCR register is not privileged and can be accessed directly. The trap is reserved.

33 Trap type 33 was used in pre-V9 SPARC systems to copy the rightmost four bits from global register %g1 to the PSR icc integer condition codes. In V9 the CCR register is not privileged and can be accessed directly. The trap is reserved.

34 Trap 34 is used to return control to the system from a deferred user trap handler.

35 to 127 The operating system reserves these trap types for its own use. Programs that use them do not conform to the ABI.

3.3.3. User Traps

The operating system can redirect certain traps from non-privileged code back to user trap handlers. The interface for this functionality is declared in the new include file <sys/utrap.h>. See Libraries/System Data Interfaces/Data Definitions, figure 6-57+.

Figure 3-35+: Hardware Traps and User Traps

Trap Name	User Trap	
illegal_instruction	UT_ILLTRAP_INSTRUCTION **or** UT_ILLEGAL_INSTRUCTION	†
fp_disabled	UT_FP_DISABLED	†
fp_exception_ieee_754	UT_FP_EXCEPTION_IEEE_754	†
fp_exception_other	UT_FP_EXCEPTION_OTHER	
tag_overflow	UT_TAG_OVERFLOW	†
division_by_zero	UT_DIVISION_BY_ZERO	†
mem_address_not_aligned	UT_MEM_ADDRESS_NOT_ALIGNED	†
privileged_action	UT_PRIVILEGED_ACTION	†
privileged_opcode	UT_PRIVILEGED_OPCODE	
async_data_error	UT_ASYNC_DATA_ERROR	
trap_instruction	UT_TRAP_INSTRUCTION_16 **through** UT_TRAP_INSTRUCTION_31	† †
instruction_access_exception instruction_access_MMU_miss instruction_access_error	UT_INSTRUCTION_EXCEPTION **or** UT_INSTRUCTION_PROTECTION **or** UT_INSTRUCTION_ERROR	
data_access_exception data_access_MMU_miss data_access_error data_access_protection	UT_DATA_EXCEPTION **or** UT_DATA_PROTECTION **or** UT_DATA_ERROR	

User trap types marked with † above are required and must be provided by all ABI-conforming implementations. The other may not be present on every implementation; an attempt to install a user trap handler for that condition will return EINVAL.

Most user trap types are self-explanatory; a few require a few more words.

UT_ILLTRAP_INSTRUCTION
: This trap is raised by user execution of the ILLTRAP instruction. It is always precise.

UT_ILLEGAL_INSTRUCTION
: This trap will be raised by execution of otherwise undefined opcodes. It is implementation-dependent as to what opcodes raise this trap; the ABI only specifies the interface. The trap may be precise or deferred.

UT_PRIVILEDGED_OPCODE
: All the opcodes declared to be privileged in SPARC V9 will raise this trap. It is

implementation-dependent whether other opcodes will raise it as well; the ABI only specifies the interface.

UT_DATA_EXCEPTION, UT_INSTRUCTION_EXCEPTION
: No valid user mapping can be made to this address, for a data or instruction access, respectively.

UT_DATA_PROTECTION, UT_INSTRUCTION_PROTECTION
: A valid mapping exists, and user privilege to it exists, but the type of access (read, write, or execute) is denied, for a data or instruction access, respectively.

UT_DATA_ERROR, UT_INSTRUCTION_ERROR
: A valid mapping exists, and both user privilege and the type of access are allowed, but an unrecoverable error occurred in attempting the access, for a data or instruction access, respectively. %l1 will contain either BUS_ADDRERR or BUS_OBJERR.

A functional interface is provided to establish the user trap handlers.

```
int sparc_utrap_set(     utrap_entry_t utrap,
                         utrap_handler_t new_precise,
                         utrap_handler_t new_deferred);
```

This function establishes new values for the user trap handlers for the specified trap type.

```
int sparc_utrap_get(     utrap_entry_t utrap,
                         utrap_handler_t *old_precise,
                         utrap_handler_t *old_deferred);
```

This function returns the existing trap handler values without changing them.

```
int sparc_utrap_swap(    utrap_entry_t type,
                         utrap_handler_t new_precise,
                         utrap_handler_t new_deferred,
                         utrap_handler_t *old_precise,
                         utrap_handler_t *old_deferred);
```

This function combines the functionality of the functions, `sparc_utrap_set` and `sparc_utrap_get`, in a single atomic operation.

A new handler address of NULL means no user handler of that type will be installed. A new handler address of UTH_NOCHANGE means that the user handler for that type should not be changed. An old handler pointer of NULL means that the user is not interested in the old handler address.

For all traps, the handler executes in a new window, where the *in* registers are the *out* registers of the previous frame and have the value they contained at the time of the trap. Similarly the *global* registers (including the special registers `%ccr`, `%asi`, and `%y`) and the *floating-point* registers have their values at the time of the trap. If the handler needs scratch space, it should decrement the stack pointer to obtain it. If the handler needs access to the previous frame's *in* registers or *local* registers, it should execute a FLUSHW instruction, and then access them off of the frame pointer. If the handler calls an ABI-conforming function, it must set the %asi register to ASI_PRIMARY_NOFAULT before the call.

3.3.3.1. Precise Traps

On entry to a precise user trap handler %l6 contains the %pc and %l7 contains the %npc at the time of the trap. To return from a handler and reexecute the trapped instruction, the handler would execute:

```
jmpl %l6, %g0
return %l7
```

To return from a handler and skip the trapped instruction, the handler would execute:

```
jmpl %l7, %g0
return %l7+4
```

3.3.3.2. Deferred Traps

On entry to a deferred user trap handler %o0 contains the address of the instruction that caused the trap and %o1 contains the actual instruction, if the information is available. Otherwise %o0 contains the value -1 and %o1 is undefined. For certain cases additional information may be made available as indicated in the following table.

Instructions	**Additional Information**
LD-type LDSTUB	%o2 contains the effective address (*rs1* + *rs2* | *simm13*).
ST-type CAS SWAP	%o2 contains the effective address (*rs1* + *rs2* | *simm13*). %o3 contains the data to be stored if available.
Integer arithmetic	%o2 contains the *rs1* value. %o3 contains the *rs2* | *simm13* value. %o4 contains the contents of %y register.
Floating-point arithmetic	%o2 contains the address of *rs1* value. %o3 contains the address of *rs2* value.
Control-transfer	%o2 contains the target address (*rs1* + *rs2* | *simm13*).
Asynchronous data errors	%o2 contains the address that caused the error. %o3 contains the effective ASI, if a variable, else -1

To return from a deferred trap, the trap handler issues:

```
ta      34      !ST_RETURN_FROM_DEFERRED_TRAP
```

The instruction that causes the trap will NOT be retried.

3.3.3.3. Dispatching Traps

The following pseudo-code explains how the operating system dispatches traps.

```
if (precise_trap) {
        if (precise_handler) {
                invoke(precise_handler);
                /* not reached */
        } else {
                convert_to_signal(precise_trap);
        }
} else if (deferred_trap) {
        if (deferred_handler) {
                invoke(deferred_handler);
                /* not reached */
        } else {
                convert_to_signal(deferred_trap);
        }
}

if (signal)
        send(signal);
```

User trap handlers must preserve all registers except the *locals* (`%l0-7`) and *outs* (`%o0-7`), i.e. `%i0-7`, `%g1-7`, `%d0-62`, `%asi`, `%fsr`, `%fprs`, `%ccr`, and `%y`, except to the extent that modifying the registers is part of the desired functionality of the handler. For example, the handler for `UT_FP_DISABLED` may load floating-point registers.

3.4. Process Initialization

All processes are initiated by the privileged operating system software with the following characteristics:

- 1. Interrupts enabled
- 2. Non-privileged mode
- 3. Normal global registers

3.4.1. Special Registers

The architecture defines three non-privileged state registers and one privileged register to control and monitor the processor. They are the condition code register (CCR), the floating-point registers state (FPRS), the floating-point state register (FSR), and the processor state register (PSTATE). The tables below give the initial state of these registers.

Figure 3-30: Condition Code Register (CCR) Fields

Field	Value	Note
xcc	unspecified	Extended integer condition codes unspecified
icc	unspecified	Integer condition codes unspecified

The architecture defines floating point instructions, and those instructions work whether the processor has a hardware floating-point unit or not. (A system may provide hardware or software floating point facilities.) In either case, however, the processor presents a working floating-point implementation, including an FPRS and an FSR with the following initial values.

Figure 3-30+: Floating-point Registers State (FPRS) Fields

Field	Value	Note
FEF	1	Floating-point unit enabled
DL	0	Lower half of floating point registers are not dirty
DU	0	Upper half of floating-point registers are not dirty

Figure 3-31: Floating-point State (FSR) Register Fields

Field	Value	Note
fcc3	unspecified	Floating-point condition codes unspecified
fcc2	unspecified	Floating-point condition codes unspecified
fcc1	unspecified	Floating-point condition codes unspecified
RD	0	Round to nearest
TEM	0	Floating-point traps not enabled
NS	0	Nonstandard mode off
ver	read only	Implementation version number
ftt	unspecified	Floating-point trap type unspecified
qne	0	Floating-point queue (if any) is empty
fcc0	unspecified	Floating-point condition codes unspecified
aexc	0	No accrued exceptions
cexc	0	No current exceptions

Other non-privileged registers and their initial states are listed in the table below.

Figure 3-31+: Other Non-privileged Registers

Register	Value	Note
%asi	ASI_PRIMARY_NOFAULT	Address space identifier default
%tick	positive	Monotonically increasing
%pc	--	The current program counter
%y	unspecified	Y register unspecified

3.4.2. Process Stack and Registers

When a process receives control, its stack holds the arguments and environment from `exec`(BA_OS).

Figure 3-32: Initial Process Stack

	Unspecified	*High Addresses*
	Information block, including argument strings environment strings auxiliary information . . . (size varies)	
	Unspecified	
	Null auxiliary vector entry	
	Auxiliary vector . . . (2 extended-word entries)	
	0 extended-word	
	Environment pointers . . . (1 extended-word each)	
	0 extended-word	
	Argument pointers . . . (*Argument count* extended-words)	
%sp+BIAS+128	Argument count	
%sp+BIAS+0	Window save area (16 extended-words)	*Low Addresses*

Argument strings, environment strings, and the auxiliary information appear in no specific order within the information block; the system makes no guarantees about their arrangement. The system also may leave an unspecified amount of memory between the null auxiliary vector entry and the beginning of the information block.

Except as shown below, global, floating point, and window registers have unspecified values at process entry. Consequently, a program that requires registers to have specific values must set them explicitly during process initialization. It should *not* rely on the system to set all registers to zero.

`%g1` A non-zero value gives a function pointer that the application should register with `atexit`(BA_OS). If `%g1` contains zero, no action is required.

`%fp` The system marks the deepest stack frame by setting the frame pointer to zero. No other frame's `%fp` has a zero value.

`%sp` Performing its usual job, the stack pointer plus the stack BIAS gives the address of the bottom of the stack, which is guaranteed to be 16-byte aligned.

Every process has a stack, but the system defines *no* fixed stack address. Furthermore, a program's stack address can change from one system to another - even from one process invocation to another. Thus the process initialization code must use the stack address in `%sp`. Data in the stack segment at addresses below the stack pointer contain undefined values.

[The information on auxiliary information is unchanged.]

In the following example, the stack resides below 0x8000 0000 0000 0000, growing toward lower addresses. The process receives three arguments.

- `cp`
- `src`
- `dst`

It also inherits two environment strings (this example is not intended to show a fully configured execution environment).

- `HOME=/home/dir`
- `PATH=/home/dir/bin:/usr/bin:`

Its auxiliary vector holds one non-null entry, a file descriptor for the executable file.

- `13`

The initialization sequence preserves the stack pointer's extended-word alignment.

Figure 3-35: Example Process Stack

	r	/	b	i	n	:	\0	*pad*	*High addresses*
0x7fff ffff ffff fff0	/	b	i	n	:	/	u	s	
	h	o	m	e	/	d	i	r	
0x7fff ffff ffff ffe0	r	\0	P	A	T	H	=	/	
	/	h	o	m	e	/	d	i	
0x7fff ffff ffff ffd0	s	t	\0	H	O	M	E	=	
	c	p	\0	s	r	c	\0	d	
0x7fff ffff ffff ffc0	0								
	0								
0x7fff ffff ffff ffb0	13								
	2								Auxiliary vector
0x7fff ffff ffff ffa0	0								
	0x7fff ffff ffff ffe2								
0x7fff ffff ffff ff90	0x7fff ffff ffff ffd3								Environment vector
	0								
0x7fff ffff ffff ff80	0x7fff ffff ffff ffcf								
	0x7fff ffff ffff ffcb								
0x7fff ffff ffff ff70	0x7fff ffff ffff ffc8								Argument vector
0x7fff ffff ffff ff68	3								Argument count
0x7fff ffff ffff ff60	reserved								
%sp+BIAS 0x7fff ffff ffff fee0	Window save area (16 extended-words)								*Low addresses*

3.5. Coding Examples

This section discusses example code sequences for fundamental operations such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections discuss how a program may use the machine or the operating system, and they specify what a program may and may not assume about the execution environment. Unlike previous material, the information here illustrates how operations *may* be done, not how they *must* be done.

As before, examples use the ANSI C language. Other programming languages may use the same conventions displayed below, but failure to do so does *not* prevent a program from conforming to the ABI.

3.5.1. Architectural Constraints

The SPARC V9 architecture has a number of constraints that make it desirable to use several different code models for different purposes, in order to improve performance and reduce code size. The relevant constraints are:

a) The `call` instruction has a 30 bit signed immediate value. The target address of a `call` instruction may thus be at most 2^{29} instructions (2^{31} bytes) before it or 2^{29}- 1 instructions (2^{31}- 4 bytes) after it.

b) Memory access instructions (e.g., `ldx` and `stx`) and arithmetic and logical instructions (e.g., `add` and `or`) have a 13-bit signed immediate value.

c) The `sethi` instruction has a 22 bit unsigned immediate value that is placed in register bits 31..10. The other register bits are cleared.

3.5.1.1. Code Positionability

There are two code positionability models of interest:

absolute — The virtual addresses of instructions and static data are known at static link time. To execute properly, the program must be loaded at a specific virtual address, making the program's absolute addresses correspond with the process's virtual addresses.

position-independent (PIC) — The virtual addresses of instructions and static data are not known until dynamic link time. PIC uses PC-relative addresses, not absolute addresses. Consequently, the code is not tied to a specific load address, allowing it to execute properly at various positions in virtual memory.

Typically, executables have absolute code and shared objects such as dynamically linked libraries have PIC.

3.5.1.2. Code Size

Because of constraint (a), there are two code size models of interest:

medium The size of the text segment of an executable or shared object is less than 2^{31} bytes (2 GB).

full The only limit on the size of the text segment of an executable or shared object is the available virtual address space.

The limiting case is a CALL instruction at the beginning of a text segment whose target address is at the end of the segment. This limits a medium text segment to 2^{29}-1 instructions (2^{31}-4 bytes). A single CALL instruction can be used for all subroutine calls within a medium text segment; more code is needed for full text segments.

3.5.1.3. Location

Because of constraint (c), there are two location models of interest:

low The executable must be in the low 4 GB of the virtual address space.

anywhere The executable or shared object can be placed anywhere in the virtual address space.

The low model applies only to absolute code. The low model generates the most efficient code for accessing static objects: two instructions and one register always suffice.

3.5.1.4. External Object References

A shared object that references an object external to itself must use indirect addressing. For example, the libc function localtime() references the external variable daylight. At the time the libc shared library is created, the address of daylight is not known, so references to it from libc go through a global offset table. Each shared object has its own global offset table, which is just a vector of addresses. Each object, e.g. daylight, is associated with an index into the global offset table. At dynamic link time, the dynamic linker fills in daylight's element in the global offset table with the absolute address of daylight.

Because of the effects of constraints (b) and (c) on addressing elements in global offset tables, there are three external object reference models. However, only the first two are of practical interest.

small The executable or shared object references at most 1024 external objects.

large The executable or shared object references at most 2^{29} external objects.

huge The size of the global offset table is limited only by the available virtual address space.

The limiting factor is the 13-bit signed immediate in load instructions. Assuming the address of the middle of the global offset table is already in some register, the small model can load any element with one LDX instruction, whereas the large model requires three instructions.

3.5.1.5. Combinations of Practical Interest

The following combinations of models are of practical use. All models use dynamic linking.

Positionability	Code Size	Location	External Object Reference Model
absolute	medium	low	small
absolute	medium	low	large
absolute	medium	low	none
absolute	medium	anywhere	small
absolute	medium	anywhere	large
absolute	medium	anywhere	none
absolute	full	anywhere	large
absolute	full	anywhere	none
PIC	medium	anywhere	small
PIC	medium	anywhere	large
PIC	full	anywhere	large

3.5.1.6. Integer Constant Loading

There are a number of ways to load an integer constant into a register. The examples in the following table assume x is the ones complement of bits 31..10 of the constant (treated as a 64-bit bit vector), y is the binary value 111 followed by the low-order 10 bits of the constant, %hh(c) is bits 63..42 of c, %hm(c) is bits 41..32 of c, %lm(c) is bits 31..10 of c and %lo(c) is bits 9..0 of c. The table is not exhaustive.

Figure x.x: Loading Integer Constants

Range	Code
-2^{12} .. $2^{12}-1$	`or %g0, c, %o0`
0 .. $2^{32}-1$	`sethi %hi(c), %o0` `or %o0, %lo(c), %o0`
-2^{32} .. -1	`sethi x, %o0` `xor %o0, y, %o0`

-2^{63} .. $2^{63}-1$	`sethi %hh(c), %o1`
	`sethi %lm(c), %o0`
	`or %o1, %hm(c), %o1`
	`or %o0, %lo(c), %o0`
	`sllx %o1, 32, %o1`
	`or %o0, %o1, %o0`

NOTE Since the general case costs 6 instructions and a scratch register, loading from a constant table may be more efficient in some cases.

3.5.1.7. Addressing Global Offset Tables

A subroutine in a shared object must obtain the address of the shared object's global offset table before the subroutine can access the table. Typically, this is done in a prologue. The offset between the subroutine's address and the middle of the global offset table must be known when the shared object is created. The following code examples place the address of the middle of the global offset table in `%l7`; other registers can also be used. *offset* is the offset in bytes from the `rd` instruction to the middle of the global offset table. In the medium size case it is assumed to be positive.

Medium Size Code	**Full Size Code**

`rd %pc, %l7`
`sethi %hi(offset), %o0`
`or %o0, %lo(offset), %o0`
 	
`add %l7, %o0, %l7`
	`rd %pc, %l7`
	`sethi %hh(offset), %o1`
	`sethi %lm(offset), %o0`
	`or %o1, %hm(offset), %o1`
	`or %o0, %lo(offset), %o0`
	`sllx %o1, 32, %o1`
	`or %o0, %o1, %o0`
	`add %l7, %o0, %l7`

3.5.1.8. Static Data References from Absolute Code

For medium sized code locatable anywhere, register %g4 is assumed to contain the address of the start of the data segment. All data address constants are then relative to the start of the data segment. %g4 (or any other preserved global register) can be set up once in an executable's startup code (see below).

Figure x.x: Static Data References from Absolute Code

ANSI C	medium/low	medium/anywhere	full/anywhere
extern long s; extern long d; extern long *p;	.global s .global d .global p	.global s .global d .global p	.global s .global d .global p
p = &d;	sethi %hi(d),%o0 or %o0,%lo(d),%o0 sethi %hi(p),%o1 stx %o0,[%o1+%lo(p)]	sethi %hi(d),%o0 or %o0,%lo(d),%o0 add %o0,%g4,%o0 sethi %hi(p),%o1 or %o1,%lo(p),%o1 stx %o0,[%g4+%o1]	sethi %hh(d),%o5 sethi %lm(d),%o0 or %o5,%hm(d),%o5 or %o0,%lo(d),%o0 sllx %o5,32,%o5 or %o0,%o5,%o0 sethi %hh(p),%o5 sethi %lm(p),%o1 or %o5,%hm(p),%o5 or %o1,%lo(p),%o1 sllx %o5,32,%o5 stx %o0,[%o1+%o5]

*p = s;	sethi %hi(s),%o0
	
	ldx [%o0+%lo(s)],%o0
	sethi %hi(p),%o1
	
	ldx [%o1+%lo(p)],%o1
	stx %o0,[%o1]
		sethi %hi(s),%o0
		or %o0,%lo(s),%o0
		
		ldx [%g4+%o0],%o0
		sethi %hi(p),%o1
		or %o1,%lo(p),%o1
		
		ldx [%g4+%o1],%o1
		stx %o0,[%o1]
			sethi %hh(s),%o5
			sethi %lm(s),%o0
			or %o5,%hm(s),%o5
			or %o0,%lo(s),%o0
			sllx %o5,32,%o5
			ldx [%o0+%o5],%o0
			sethi %hh(p),%o5
			sethi %lm(p),%o1
			or %o5,%hm(p),%o5
			or %o1,%lo(p),%o1
			sllx %o5,32,%o5
			ldx [%o1+%o5],%o1
			stx %o0,[%o1]

The following code could be used in the medium/anywhere startup code. *data_start* is the virtual address of the start of the executable's data segment. Because the code is absolute, *data_start* is known at static link time

Figure x.x: Startup Code for Medium/Anywhere Model

```
sethi  %hh(data_start), %g1
sethi  %lm(data_start), %g4
or     %g1, %hm(data_start), %g1
or     %g4, %lo(data_start), %g4
sllx   %g1, 32, %g1
or     %g4, %g1, %g4
```

3.5.1.9. Static Data References from PIC

Figure x.x: Static Data References from Position Independent Code

ANSI C	Small Model	Large Model

`extern long s;`
`extern long d;`
`extern long *p;`
	`.global s`
	`.global d`
	`.global p`
		`.global s`
		`.global d`
		`.global p`

`p = &d;`	
	
	`ldx [%l7+d],%o0`
	
	`ldx [%l7+p],%o1`
	`stx %o0,[%o1]`
		`sethi %hi(d),%o0`
		`or %o0,%lo(d),%o0`
		`ldx [%l7+%o0],%o0`
		`sethi %hi(p),%o1`
		`or %o1,%lo(p),%o1`
		`ldx [%l7+%o1],%o1`
		`stx %o0,[%o1]`

`*p = s;`	
	
	`ldx [%l7+s],%o0`
	`ldx [%o0],%o0`
	
	`ldx [%l7+p],%o1`
	`ldx [%o1],%o1`
	`stx %o0,[%o1]`
		`sethi %hi(s),%o0`
		`or %o0,%lo(s),%o0`
		`ldx [%l7+%o0],%o0`
		`ldx [%o0],%o0`
		`sethi %hi(p),%o1`
		`or %o1,%lo(p),%o1`
		`ldx [%l7+%o1],%o1`
		`ldx [%o1],%o1`
		`stx %o0,[%o1]`

3.5.2. Function Calls

Direct function calls are those where the name of the called function is known at compile time. The following code shows the cases of interest. The `call` instruction can be used in all medium size executables and shared objects.

Figure x.x: Function Calls

ANSI C	medium	absolute/full	PIC/full
`extern void f();`	`.global f`	`.global f`	`.global f`

`f();`	
	
	`call f`
	`nop`
		`sethi %hh(f),%g2`
		`sethi %lm(f),%g1`
		`or %g2,%hm(f),%g2`
		`or %g1,%lo(f),%g1`
		`sllx %g2,32,%g2`
		`jmpl %g1+%g2,%o7`
		`nop`
			`sethi %hi(f),%g1`
			`or %g1,%lo(f),%g1`
			`ldx [%l7+%g1],%g1`
			
			`jmpl %g1,%o7`
			`nop`

For indirect function calls, the address of the function is in a pointer. Appropriate code is used to load the value of the pointer into a register, just as with static data. A `jmpl` instruction is then used.

3.5.3. Branching

Programs use branch instructions to control their execution flow. As defined by the architecture, branch instructions hold a PC-relative value with up to a 2 MB range, allowing a branch to locations up to 1 MB away in either direction.

C switch statements provide multiway selection. The best implementation of a switch statement depends on the distribution of the case label values. When they are dense, as in the C example below then the computed-jump approach shown may generate good code. The example uses several simplifying conventions to hide irrelevant details:

- The selection expression resides in local register %l0.
- case label constants begin at zero.
- case labels and default use assembly names .Lcasei and .Ldef, respectively.

The following example is position-independent, and can also be used in absolute code.

Figure 3-46: Position-Independent `switch` Code

ANSI C	Assembly

```
switch (j)
{
case 0:
     ...
case 2:
     ...
case 3:
     ...
default:
     ...
}
```

```
          subcc  %l0, 4, %g0
          movgu  xcc, 1, %l0
1:
          rd     %pc, %l1
          sllx   %l0, 5, %l0
          add    %l0, (.Lcase0 - 1b), %l0
          jmpl   %l0 + %l1, %g0
          nop
.Lcase0:
          instruction 1
          instruction 2
          instruction 3
          instruction 4
          instruction 5
          instruction 6
          ba     .Lcase0_continued
          instruction 8
.Ldef:
          instruction 1
          instruction 2
          instruction 3
          instruction 4
          instruction 5
          instruction 6
          ba     .Lcase_end
          instruction 8
.Lcase2:
           ...
.Lcase0_continued:
           ...
.Lcase_end:
```

The number of instructions in the legs can be varied. If there is not enough space in a leg, a branch to additional code can be used.

3.5.4. C Stack Frame

Figure 3-47: C Stack Frame

Base	Offset	Contents	Frame
%fp+BIAS	-1	*y* extended words local space: automatic variables . . .	*High addresses*
%fp+BIAS	-8y	other addressable objects	
%sp+BIAS	+128+8x	*x* extended-words compiler scratch temporaries, register save area,	Current
%sp+BIAS	+128	and outgoing argument slots	
%sp+BIAS	0	16 extended word window save area	*Low addresses*

The figure above shows the C stack frame organization. It conforms to the standard stack frame with designated roles for unspecified areas in the standard frame. A C stack frame doesn't normally change size during execution. The exception is dynamically allocated stack memory, discussed below. By convention, a function allocates automatic (local) variables in the top of its frame and references them as negative offsets from %fp+BIAS. Its incoming overflow arguments reside in the previous frame, referenced as positive offsets from %fp+BIAS.

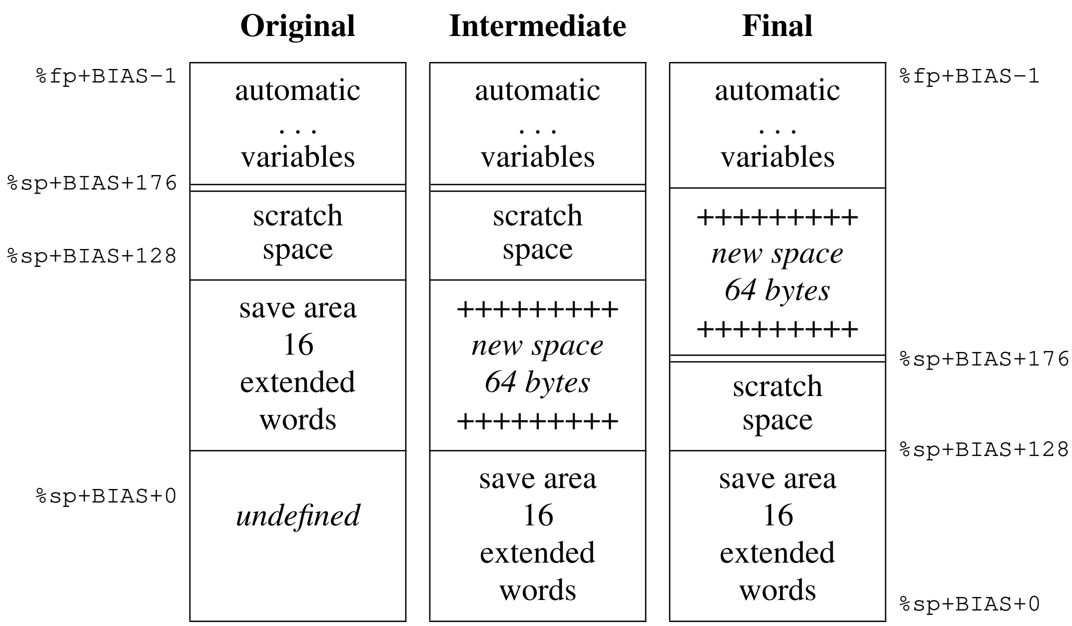
3.5.5. Variable Argument List

Previous sections describe the rules for passing arguments. Unfortunately, some otherwise portable C programs depend on the argument passing scheme, implicitly assuming that 1) all arguments reside on the stack, and 2) arguments appear in increasing order on the stack. Programs that make these assumptions never have been portable, but they have worked on many machines. They do *not* work on SPARC because some of the arguments reside in integer and/or floating point registers. Portable C programs should use the facilities defined in the header files <stdarg.h> or <varargs.h> to deal with variable argument lists (on SPARC and other machines as well).

3.5.6. Allocating Stack Space Dynamically

To illustrate, assume a program wants to allocate 50 bytes; its current stack frame has 24 bytes of compiled scratch space. The first step is rounding the 50 to 64, making it a multiple of 16. Figure 3-49 shows how the stack changes.

Figure 3-49: Dynamic Stack Allocation



New space starts at %sp+BIAS+176 As described, every dynamic allocation in *this* function will return a new area starting at %sp+BIAS+176, leaving previous stack objects untouched (other functions would have different stack addresses). Consequently, the compiler should compute the absolute address for each area, avoiding relative references. Otherwise future allocations in the same frame would destroy the stack's integrity.

4. OBJECT FILES

4.1. ELF Header

For file identification in `e_ident`, **SPARC** requires the following values.

.**Figure 4-1: SPARC V9 Identification, `e_ident`**

Position	Value
`e_ident[EI_CLASS]`	`ELFCLASS64`
`e_ident[EI_DATA]`	`ELFDATA2MSB`

Processor identification resides in the ELF header's `e_machine` member and must have the value 11, defined as the name `EM_SPARC`64.

The ELF headers `e_flags` member holds bit flags associated with the file. **SPARC**64 defines the following flags.

[The following table represents work in progress and is highly likely to change.]

Figure 4-2: SPARC64 V9 flags, `e_flags`

Name	Value	Meaning
EF_SPARC64_MM	0x3	Mask for Memory Model
EF_SPARC64_TSO	0x0	Total Store Ordering
EF_SPARC64_PSO	0x1	Partial Store Ordering
EF_SPARC64_RMO	0x2	Relaxed Memory Ordering
EF_SPARC_SUN_US1	0x000200	Sun UltraSPARC1 extensions
EF_SPARC_HAL_R1	0x000400	HAL R1 extensions

.All unspecified bits are reserved and should be set to zero. The compilation system sets the EF_SPARC64_MM field to the value required for the correct execution of the object. Typically, the programmer specifies what value to use for compiling a given source unit. TSO is the most restrictive memory model, followed by PSO, followed by RMO, in that order.

It is recommended that the default compilation model should be RMO to realize the performance advantages of this memory model. A binder that statically links input objects into a single output object will set EF_SPARC64_MM to the most restrictive model specified by any of the input objects.

At execution time, the dynamic linker will inform the operating system of the most restrictive model required by any of the objects that are part of the execution environment. The operating system will use this information to provide the memory order semantics of that model to the application, if available, or a more restrictive one.

The memory model flag expresses a requirement that the program has on the memory model semantics of the execution environment, but does *not* constrain the implementation in how it provides that model. For example, on a uniprocessor, the implementation can usually ignore the memory model flags, and set the processor into RMO mode because the program can only observe TSO memory ordering semantics.

4.2. Sections

[This section is unchanged.]

4.3. Relocation

The `r_info` field is composed of two 32-bit parts, the symbol table index and the relocation type. The relocation type on SPARC V9 systems is further decomposed into an 8-bit type identifier and a 24-bit type dependent data field. For the existing ELF-32 relocation types, that data field is zero. New relocation types, however, may make use of these bits.

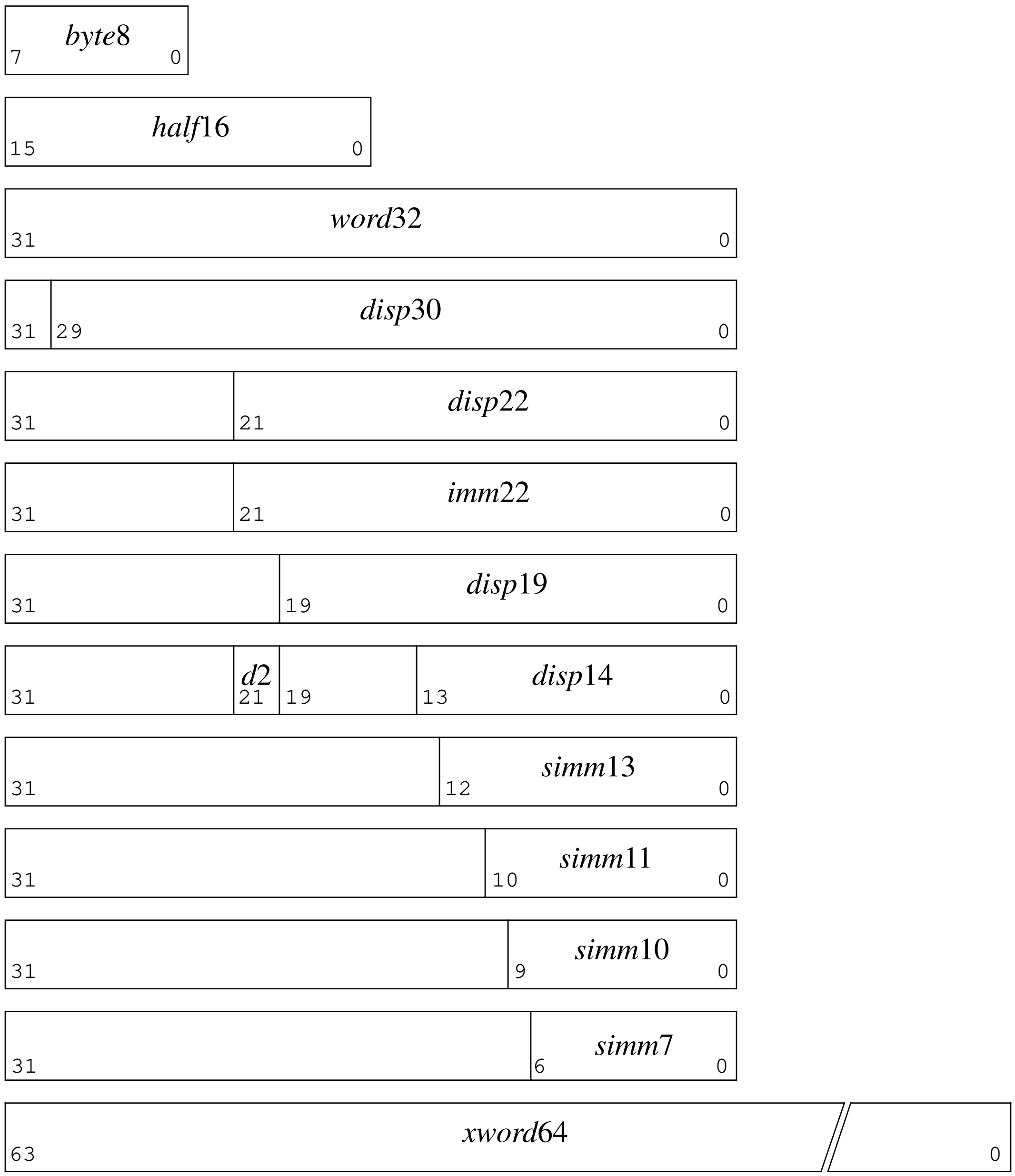
Figure 4-3: Relocation Macros

```
#define ELF64_R_TYPE_DATA(info)        (((Elf64_Xword)(info) << 32) >> 40)
#define ELF64_R_TYPE_ID(info)          (((Elf64_Xword)(info) << 56) >> 56)
#define ELF64_R_TYPE_INFO(data, type)  (((Elf64_Xword)(data) << 8)
                                       + (Elf64_Xword)(type))
```

4.3.1. Relocation Types

An overview of the instruction and data formats from *The SPARC™ Architecture Manual, Version 9* makes relocation easier to understand. Relocation entries describe how to alter the following instruction and data fields (bit numbers appear in the lower box corners).

Figure 4-3: Relocatable Fields



Calculations below assume the actions are transforming a relocatable file into either an executable or a shared object file. Conceptually, the link editor merges one or more relocatable files to form the output. It first decides how to combine and relocate the input files, then updates the symbol values, and finally performs the relocation. Relocations applied to executable or shared object files are similar and accomplish the same result. Descriptions below use the following notation.

A This means the addend used to compute the value of the relocatable field.

B This means the base address at which a shared object has been loaded into memory during execution. Generally a shared object file is built with a 0 base virtual address, but the execution address will be different. See “Program Header” in the System V ABI for more information about base addresses.

G This means the offset into the global offset table at which the address of the relocation entry’s symbol will reside during execution. See “Coding Examples” in Chapter 3 and “Global Offset Table” in Chapter 5 for more information.

L This means the place (section offset or address) of the procedure linkage table entry for a symbol. A procedure linkage table entry redirects a function call to the proper destination. The link editor builds the initial procedure linkage table, and the dynamic linker modifies the entries during execution. See “Procedure Linkage Table” in Chapter 5 for more information.

O This means the secondary addend used to compute the value of the relocation field. The secondary addend is extracted from the `r_info` field in the relocation entry by applying the `ELF64_R_TYPE_DATA` macro.

P This means the place (section offset or address) of the storage unit being relocated (computed using `r_offset`).

S This means the value of the symbol whose index resides in the relocation entry.

Relocation entries apply to bytes (*byte*8), halfwords (*half*16), extended-words, (*xword*64), or words (the others). In any case, the `r_offset` value designates the offset or virtual address of the first byte of the affected storage unit. The relocation type specifies which bits to change and how to calculate their values. SPARC V9 uses only `Elf64_Rela` relocation entries with explicit addends. Thus the `r_addend` member serves as the relocation addend.

NOTE

Field names in the following tables tell whether the relocation type checks for “overflow”. A calculated relocation value may be larger than the intended field, and a relocation type may verify (V) the value fits or truncate (T) the result. As an example, V-*imm*22 means the computed value may not have significant, non-zero bits outside the *imm*22 field.

Figure 4-4: Relocation Types

Name	Value	Field	Calculation
R_SPARC_NONE	0	none	none
R_SPARC_8	1	V-*byte*8	S + A
R_SPARC_16	2	V-*half*16	S + A
R_SPARC_32	3	V-*word*32	S + A
R_SPARC_DISP8	4	V-*byte*8	S + A - P
R_SPARC_DISP16	5	V-*half*16	S + A - P
R_SPARC_DISP32	6	V-*word*32	S + A - P
R_SPARC_WDISP30	7	V-*disp*30	(S + A - P) >> 2
R_SPARC_WDISP22	8	V-*disp*22	(S + A - P) >> 2
R_SPARC_HI22	9	V-*imm*22	(S + A) >> 10
R_SPARC_22	10	V-*imm*22	S + A
R_SPARC_13	11	V-*simm*13	S + A
R_SPARC_LO10	12	T-*simm*13	(S + A) & 0x3ff
R_SPARC_GOT10	13	T-*simm*13	G & 0x3ff
R_SPARC_GOT13	14	V-*simm*13	G
R_SPARC_GOT22	15	T-*imm*22	G >> 10
R_SPARC_PC10	16	T-*simm*13	(S + A - P) & 0x3ff
R_SPARC_PC22	17	V-*imm*22	(S + A - P) >> 10
R_SPARC_WPLT30	18	V-*disp*30	(L + A - P) >> 2
R_SPARC_COPY	19	none	none
R_SPARC_GLOB_DAT	20	V-*xword*64	S + A
R_SPARC_JMP_SLOT	21	none	see below
R_SPARC_RELATIVE	22	V-*word*32	B+ A
R_SPARC_UA32	23	V-*word*32	S + A

Some relocation type have semantics beyond simple calculation.

`R_SPARC_GOT10` This relocation type resembles `R_SPARC_LO10`, except it refers to the address of the symbols global offset table entry and additionally instructs the link editor to build a global offset table.

`R_SPARC_GOT13` This relocation type resembles `R_SPARC_13`, except it refers to the address of the symbols global offset table entry and additionally instructs the link editor to build a global offset table.

`R_SPARC_GOT22` This relocation type resembles `R_SPARC_22`, except it refers to the address of the symbols global offset table entry and additionally instructs the link editor to build a global offset table.

`R_SPARC_WPLT30` This relocation type resembles `R_SPARC_WDISP30`, except it refers to the address of the symbol's procedure linkage table entry and additionally instructs the link editor to build a procedure linkage table.

R_SPARC_COPY — The link editor creates this relocation type for dynamic linking. Its offset member refers to a location in a writable segment. The symbol table index specifies a symbol that should exist both in the current object file and in a shared object. During execution, the dynamic linker copies data associated with the shared object's symbol to the location specified by the object.

R_SPARC_GLOB_DAT — This relocation type resembles R_SPARC_64, except it is used to set a global offset table entry to the address of the specified symbol. The special relocation type allows one to determine the correspondence between symbols and global offset table entries.

R_SPARC_JMP_SLOT — The link editor creates this relocation type for dynamic linking. Its offset member gives a location of a procedure linkage table entry. The dynamic linker modifies the procedure linkage table entry to transfer control to the designated symbol's address [See "Procedure Linkage Table" in chapter 5].

R_SPARC_RELATIVE — The link editor creates this relocation type for dynamic linking. Its offset member gives a location within a shared object that contains a value representing a relative address. The dynamic linker computes the corresponding virtual address by adding the virtual address at which the shared object was loaded to the relative address. Relocation entries for this type must specify 0 for the symbol table index.

R_SPARC_UA32 — This relocation type resembles R_SPARC_32, except it refers to an unaligned word. That is the "word" to be relocated must be treated as four separate bytes with arbitrary alignment, not as a word aligned according to the architecture requirements.

Figure 4-4+: More Relocation Types

Name	Value	Field	Calculation
R_SPARC_PLT32	24	V-word32	L + A
R_SPARC_HIPLT22	25	T-imm22	(L + A) >> 10
R_SPARC_LOPLT10	26	T-simm13	(L + A) & 0x3ff
R_SPARC_PCPLT32	27	V-word32	L + A - P
R_SPARC_PCPLT22	28	V-disp22	(L + A - P) >> 10
R_SPARC_PCPLT10	29	V-simm12	(L + A - P) & 0x3ff
R_SPARC_10	30	V-*simm*10	S + A
R_SPARC_11	31	V-*simm*11	S + A
R_SPARC_64	32	V-*xword*64	S + A
R_SPARC_OLO10	33	V-*simm*13	((S + A) & 0x3ff) + O
R_SPARC_HH22	34	V-*imm*22	(S + A) >> 42
R_SPARC_HM10	35	T-*simm*13	((S + A) >> 32) & 0x3ff
R_SPARC_LM22	36	T-*imm*22	(S + A) >> 10
R_SPARC_PC_HH22	37	V-*imm*22	(S + A - P) >> 42
R_SPARC_PC_HM10	38	T-*simm*13	((S + A - P) >> 32) & 0x3ff
R_SPARC_PC_LM22	39	T-*imm*22	(S + A - P) >> 10
R_SPARC_WDISP16	40	V-*d2/disp*14	(S + A - P) >> 2
R_SPARC_WDISP19	41	V-*disp*19	(S + A - P) >> 2
R_SPARC_GLOB_JMP	42	V-*xword*64	S + A
R_SPARC_7	43	V-*imm7*	(S + A) & 0x7f
R_SPARC_5	44	V-imm5	S + A) & 0x1f
R_SPARC_6	45	V-imm6	S + A) & 0x3f

`R_SPARC_OLO10` This relocation type resembles `R_SPARC_LO10`, except an extra offset is added to make full use of the 13-bit signed immediate field.

`R_SPARC_HH22` This relocation type is used by the assembler when it sees an instruction of the form “*imm22-instruction* ... `%hh`(*absolute*) ...”.

`R_SPARC_HM10` This relocation type is generated by the assembler when it sees an instruction of the form “*simm13-instruction* ... `%hm`(*absolute*) ...”.

`R_SPARC_LM22` This relocation type is used by the assembler when it sees an instruction of the form “*imm22-instruction* ... `%lm`(*absolute*) ...”. This resembles `R_SPARC_HI22`, except it truncates rather than validates.

`R_SPARC_PC_HH22` This relocation type is used by the assembler when it sees an instruction of the form “*imm22-instruction* ... `%hh`(*pc-relative*) ...”.

`R_SPARC_PC_HM10` This relocation type is generated by the assembler when it sees an instruction of the form “*simm13-instruction* ... `%hm`(*pc-relative*) ...”.

R_SPARC_PC_LM22 This relocation type is used by the assembler when it sees an instruction of the form "*imm22-instruction* ... %lm(*pc-relative*) ...". This resembles R_SPARC_PC22, except it truncates rather than validates.

R_SPARC_GLOB_JMP This relocation type resembles R_SPARC_GLOB_DAT, except that it is guaranteed to be associated with a procedure call and therefore the dynamic linker may evaluate the relocation lazily.

R_SPARC_LO7 This relocation type is used by the assembler for 7 bit software trap numbers.