General registers 32 through 127 form a register stack that is automatically managed across procedure calls and returns. Each procedure frame on the register stack is divided into two dynamically-sized regions—one for input parameters and local variables, and one for output parameters. On a procedure call, the registers are automatically renamed by the hardware so that the caller’s output registers form the base of the callee’s new register stack frame. On return, the registers are restored to the previous state, so that the input and local registers are preserved across the call.

The \texttt{alloca} instruction is used at the beginning of a procedure to allocate the input, local, and output regions; the sizes of these regions are supplied as immediate operands. A procedure is not required to issue an \texttt{alloca} instruction if it does not need to store any values in its register stack frame. It may still read values from input registers, but it may not write to a stack register without first issuing an \texttt{alloca} instruction.

Figure 6-1 illustrates the operation of the register stack across an example procedure call. In this example, the caller allocates eight input, twelve local, and four output registers, and the callee allocates four input, six local, and five output registers.

The actual registers to which the stacking registers are physically mapped are not directly addressable by the application software.

### 6.1 Input and Local Registers

The hardware makes no distinction between input and local registers. The caller’s output registers automatically become the callee’s entire register stack frame on a procedure call, with all registers initially allocated as output registers. An \texttt{alloca} instruction may increase or decrease the total size of the register stack frame, and may adjust the boundary between the input and local region and the output region.

The software conventions specify that up to eight registers are used for parameter passing. Any registers in the input and local region beyond those eight may be allocated for use as preserved locals. Floating-point parameters may produce “holes” in the parameter list that is passed in the general registers; those unused input registers may also be used for preserved locals.

The caller’s output registers do not need to be preserved for the caller. Once an input parameter is no longer needed, or has been copied elsewhere, that register may be reused for any other purpose within the procedure.

### 6.2 Output Registers

Up to eight output registers are used for passing parameters. If a procedure call requires fewer than eight general registers for its parameters, the calling procedure does not need to allocate more than are needed. If the called procedure expects more parameters, it will allocate extra input registers; these registers will be uninitialized.
A procedure may also allocate more than eight registers in the output region. While the extra registers may not be used for passing parameters, they can be used as extra scratch registers. On a procedure call, they will show up in the called procedure’s output area as excess registers, and may be modified by that procedure. The called procedure may also allocate few enough total registers in its stack frame that the top of the called procedure’s frame is lower than the caller’s top of frame, but those registers will become available again when control returns to the caller.

### 6.3 Rotating Registers

A subset of the registers in the procedure frame may be designated as rotating registers. The rotating register region always starts with r32, and may be any multiple of eight registers in number, up to a maximum of 96 rotating registers. The renaming is under control of the Rotating Register Base (RRB).

If the rotating registers include any or all of the output registers, software must be careful when using the output registers for passing parameters, since a non-zero RRB will change the virtual register numbers that are part of the output region. In general, software should either ensure that the rotating region does not overlap the output region, or that the RRB is cleared to zero before setting output parameter registers.

### 6.4 Frame Markers

The current application-visible state of the stack frame is stored in an architecturally inaccessible register called the current frame marker. On a procedure call, this register is automatically saved by copying it to an application register, the previous function state (ar,pfs). The current frame marker is modified to describe a new stack frame whose input and local area is initially zero size, and whose output area is equal in size to the previous output area. On return, the previous frame state register is used to restore the current frame marker to its earlier value, and the base of the register stack is adjusted accordingly.

It is the responsibility of a procedure to save the previous function state register before issuing any procedure calls of its own, and to restore it before returning.

### 6.5 Backing Store for Register Stack

When the depth of the procedure call stack exceeds the capacity of the physical register file, the hardware frees physical registers by saving them into a memory stack. This backing store is distinct from the memory stack described in the next chapter.

As returns unwind the procedure call stack, the hardware also restores previously-saved physical registers from the backing store.

The operation of this register stack engine (RSE) is mostly transparent to application software. While the RSE is running, application software may not examine the contents of the backing store, and may not make any assumptions about how much of the register stack is still in physical registers or in the backing store. In order to examine previous stack frames, application software must synchronize the RSE with the flushrs instruction. Synchronizing the RSE forces all stack frames up to, but not including, the current frame to be saved in backing store, allowing the software to examine the contents of the backing store without asynchronous operations modifying the memory. Modifications to the backing store require setting the RSE to “enforced lazy mode”
after synchronizing it, which prevents the RSE from doing any operations other than those required by calls and returns. The procedure for synchronizing the RSE and setting the mode is described in Section 10.2, “User-level Thread Switch, Coroutines” on page 10-2.

**Figure 6-1. Operation of the Register Stack**

The backing store grows towards higher addresses. When the RSE is synchronized and in enforced lazy mode, the top of the stack corresponding to the top of the previous procedure frame is available in the Backing Store Pointer (bsp) application register.

Even when the RSE is in enforced lazy mode, the bsp must always point to a valid backing store address, since the operating system may need to start the RSE to process an exception.

A NaT collection register is stored into the backing store after each group of 63 physical registers. For each register stored, its NaT bit is shifted into the collection register. When the bsp reaches the doubleword just before a 64 doubleword boundary, the RSE stores the collection register. Software can determine the position of the NaT collection registers in the backing store by examining the memory address. This process is described in greater detail in the *Intel® IA-64 Architecture Software Developer’s Manual*. 
Memory Stack

The memory stack is used for local dynamic storage, spilled registers, and parameter passing. It is organized as a stack of procedure frames, beginning with the main program’s frame at the base of the stack, and continuing towards the top of the stack with nested procedure calls. At the top of the stack is the frame for the currently active procedure. (There may be some system-dependent frames at the base of the stack, prior to the main program’s frame, but an application program may not make any assumptions about them.)

The memory stack begins at an address determined by the operating system, and grows towards lower addresses in memory. The stack pointer register, sp, always points to the lowest address in the current, top-most, frame on the stack.

Each procedure creates its frame on entry by subtracting its frame size from the stack pointer, and removes its frame from the stack on exit by restoring the previous value of sp (usually by adding its frame size, but a procedure may save the original value of sp when its frame size may vary).

Because the register stack is also used for the same purposes, not all procedures will need a stack frame. Every non-leaf procedure, however, needs to save at least its return link and the previous frame marker either on the register stack or in the memory stack, so there is an activation record for every non-leaf procedure on one or both of the stacks.

7.1 Procedure Frames

A procedure frame consists of five regions, as illustrated in Figure 7-1.

**Figure 7-1. Procedure Frame**

These regions are:

- **Local storage.** A procedure may store local variables, temporaries, and spilled registers in this region. For conventions affecting the layout of this area for spilled register (see Section 11.3, “Coding Conventions for Reliable Unwinding” on page 11-5).

- **Dynamically-allocated stack storage.** This is a variable-sized region (initially zero length), that can be created by the C library alloca routine and similar routines.

- **Frame marker.** This optional region may contain information required for unwinding through the stack (for example, a copy of the previous stack pointer).
• **Outgoing parameters.** Parameters in excess of those passed in registers are stored in this region of the stack frame. A procedure accesses its incoming parameters in the outgoing parameter region of its caller’s stack frame.

• **Scratch area.** This 16-byte region is provided as scratch storage for procedures that are called by the current procedure. Leaf procedures do not need to allocate this region. A procedure may use the 16 bytes at the top of its own frame as scratch memory, but the contents of this area are not preserved by a procedure call.

The stack pointer must always be aligned at a 16-byte boundary. This implies that all stack frames must be a multiple of 16 bytes in size.

An application may not write to memory below the stack pointer, since this memory area may be written to asynchronously (for example, as a result of exception processing).

Most procedures are expected to have a fixed size frame, and the conventions are biased in favor of this. A procedure with a fixed size frame may reference all regions of the frame with a compile-time constant offset relative to the stack pointer. Compilers should determine the total size required for each region, and pad the local storage area to make the total frame size a multiple of 16 bytes. The procedure may then create the frame by subtracting an immediate constant from the stack pointer in the prologue, and remove the frame by adding the same immediate to the stack pointer in the epilogue.

If a procedure has a variable-size frame (for example, it contains a call to `malloc`), it should make a copy of `sp` to serve as a frame pointer before subtracting the initial frame size from the stack pointer. It may then restore the previous value of the stack pointer in the epilogue without regard for how much dynamic storage has been allocated within the frame. It may also use the frame pointer to access the local storage region, since offsets from `sp` will vary.

A frame pointer, as described above, is not required, however, provided that the compiler uses an equivalent method of addressing the local storage region correctly before and after dynamic allocation, and provided that the code satisfies conditions imposed by the stack unwind mechanism.

To expand a stack frame dynamically, the scratch area, outgoing parameters, and frame marker regions, which are always located relative to the current stack pointer must be relocated to the new top of stack. If the scratch area and outgoing parameter area are both clear of any live values, there is no actual work involved in relocating these areas. For procedures with dynamically-sized frames, it is recommended that the previous stack pointer value be stored in a local stacked general register instead of the frame marker, so that the frame marker is also empty. If the previous stack pointer is stored in the frame marker, the code must take care to ensure that the stack is always unwindable while the stack is being expanded (see Chapter 11, “Stack Unwinding and Exception Handling”).

Other issues 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 any stack frame region beyond those purposes described here. For example, the outgoing parameter region may be used as scratch storage whenever it is not needed for passing parameters.
8.1 External Naming Conventions

The standard naming convention, referred to as the “C” convention, specifies that all external symbols have linkage names identical to the source language identifier. There are no leading or trailing underscores. Other languages may establish other conventions, but they should provide a mechanism to define and reference symbols with “C” linkage.

8.2 The gp Register

Every procedure that references statically-allocated data or calls another procedure requires a pointer to its data segment in the gp register, so that it can access its static data and its linkage tables. Each load module has its own data segment, and the gp register must be set correctly prior to calling any entry point within that load module.

The linkage conventions require that each load module define exactly one gp value to refer to a location within its short data segment. It is expected that this location will be chosen to maximize the usefulness of short-displacement immediate instructions for addressing scalars and linkage table entries. The DLL loader will determine the absolute value of the gp register for each load module after loading its data segment into memory.

For calls within a load module, the gp register will remain unchanged, so calls known to be local can be optimized accordingly.

For calls between load modules, the gp register must be initialized with the correct gp value for the new load module, and the calling function must ensure that its own gp value is saved and restored.

8.3 Types of Calls

The following types of procedure calls are defined:

- **Direct calls.** Direct calls within the same load module may be made directly to the entry point of the target procedure. In this case, the gp register does not need to be changed.

- **Direct dynamically-linked calls.** These calls are routed through an import stub (which may be inlined at compile time if the call is known or suspected to be to another load module). The import stub obtains the address of the main entry point and the gp register value from the linkage table. Although coded in source as a direct call, dynamically-linked calls become indirect.

- **Indirect calls.** A function pointer must point to a descriptor that contains both the address of the function entry point and the gp register value for the target function. The compiler must generate code for an indirect call that sets the new gp value before transferring control to the target procedure.

- **Special calls.** Other special calling conventions are allowed to the extent that the compiler and the runtime library agree on convention, and provided that the stack may be unwound through such a call. Such calls are outside the scope of this document. See Section 8.7 for a discussion of stack unwind requirements.
8.4 Calling Sequence

Direct and indirect procedure calls are described in the following sections. Since the compiler is not required to know whether any given call is local or to another load module, the two types of direct calls are described together in the first section.

8.4.1 Direct Calls

Direct procedure calls follow the sequence of steps shown in Figure 8-1. The following paragraphs describe these steps in detail.

**Figure 8-1. Direct Procedure Calls**

- **Preparation for call.** Values in scratch registers that must be kept live across the call must be saved. They can be saved by copying them into local dynamic registers, or by saving them on the memory stack. If the NaT bits associated with any live scratch registers must be saved, the compiler should use `st8.spill` or `stf.spill` instructions. The User NaT collection register itself is preserved by the call, so the NaT bits need no further treatment at this point.

- If the call is not known (at compile time) to be within the same load module, the `gp` register must be saved.

- The parameters must be set up in registers and memory as described in Section 8.5.

- **Procedure call.** All direct calls are made with a `br.call` instruction, specifying BR 0 (also known as `rp`) for the return link.

- For direct local calls, the pc-relative displacement to the target is computed at link time. Compilers may assume that the standard displacement field in the `br.call` instruction is sufficiently wide to reach the target of the call. If the displacement is too large, the linker must supply a branch stub at
some convenient point in the code; compilers must guarantee the existence of such a point by ensuring that code sections in the relocatable object files are no larger than the maximum reach of the `br.call` instruction. With a 25-bit displacement, the maximum reach is 16 megabytes in either direction from the point of call.

Direct calls to other load modules cannot be statically bound at link time, so the linker must supply an import stub for the target procedure; the import stub obtains the address of the target procedure from the linkage table. The `br.call` instruction can then be statically bound using the pc-relative displacement to the import stub.

The `br.call` instruction saves the return link in the return BR, saves the current frame marker in the `ar.pfs` register, and sets the base of the new register stack frame to the beginning of the output region of the old frame.

**Import stub (direct external calls only).** The import stub is allocated in the load module of the caller, so that the `br.call` instruction may be statically bound to the address of the import stub. It must access the linkage table via the current `gp` (which means that `gp` must be valid at the point of call), and obtain the address of the target procedure’s entry point and its `gp` value. The import stub then establishes the new `gp` value and branches to the target entry point.

If the compiler knows or suspects that the target of a call is in a separate load module, it may wish to generate calling code that performs the functions of the import stub, saving an extra branch. The detailed operation of an import stub, however, is ABI specific.

When the target of a call is in the same load module, an import stub is not used (which also means that `gp` must be valid at the point of call).

**Procedure entry.** The prologue code in the target procedure is responsible for allocating the register stack frame, and a frame on the memory stack, if necessary. It may use the 16 bytes at the top of its caller’s memory stack frame as scratch area.

A non-leaf procedure must save the return BR and previous function state, either in the memory stack frame or in a local dynamic GR.

The prologue must also save any preserved registers that will be used in this procedure. The NaT bits for those registers must be preserved as well, by copying to local stacked general registers, or by using `st8.spill` or `stf.spill` instructions. The User NaT collection register (`ar.unat`) must be saved first, however, since it is guaranteed to be preserved by the call.

**Procedure exit.** The epilogue code is responsible for restoring the return BR and previous function state, if necessary, and any preserved registers that were saved. The NaT bits must be restored using the `ld8.fill` or `ldf.fill` instructions. The User NaT collection register must also be restored if it was saved.

If a memory stack frame was allocated, the epilogue code must deallocate it.

Finally, the procedure exits by branching through the return BR with the `br.ret` instruction.

**After the call.** Any saved values (including `gp`) should be restored.
### 8.4.2 Indirect Calls

Indirect procedure calls follow nearly the same sequence, except that the branch target is established indirectly. This sequence is illustrated in Figure 8-2.

**Figure 8-2. Indirect Procedure Calls**

#### Function Pointers

A function pointer is always the address of a function descriptor for the target procedure. The function descriptor must be allocated in the data segment of the target procedure, because it contains pointers that must be relocated by the DLL loader.

The function descriptor contains at least two 64-bit double-words: the first is the entry point address, and the second is the gp value for the target procedure. An indirect call will load the gp value into the gp register before branching to the entry point address.

In order to guarantee the uniqueness of a function pointer, and because its value is determined at program invocation time, code must materialize function pointers only by loading a pointer from the data segment. The object file format will provide appropriate relocations for this pointer.

#### Preparation for call

Indirect calls are made by first loading the function pointer into a general register, loading the entry point address and the new gp value, then using the Move to Branch Register operation to move the address of the procedure entry point into the BR to be used for the call.

Values in scratch registers that must be kept live across the call must be saved. They can be saved by copying them into local dynamic registers, or by saving them on the memory stack. If the NaT bits associated with any live scratch registers must be saved, the compiler should use st8.spill or stf.spill instructions. The User NaT collection register itself is preserved by the call, so the NaT bits need no further treatment at this point.
Unless the call is known (at compile time) to be within the same load module, the gp register must be saved before the new gp value is loaded.

The parameters must be set up in registers and memory as described in Section 8.5.

**Figure 8-3. Parameter Passing in General Registers and Memory**

![Parameter Passing Diagram]

**Procedure call.** All indirect calls are made with the indirect form of the br.call instruction, specifying BR 0 (also known as rp) for the return link.

The br.call instruction saves the return link in the return BR, saves the current frame marker in the ar.pfs register, and sets the base of the new register stack frame to the beginning of the output region of the old frame. Because the indirect call sequence obtains the entry point address and new gp value from the function descriptor, control flows directly to the target procedure, without the need for any intervening stubs.

**Procedure entry, exit, and return.** The remainder of the calling sequence is the same as for direct calls.

### 8.5 Parameter Passing

Parameters are passed in a combination of general registers, floating-point registers, and memory, as described below, and as illustrated in Figure 8-3.

The parameter list is formed by placing each individual parameter into fixed-size elements of the parameter list, referred to as parameter slots. Each parameter slot is 64 bits wide; parameters larger than 64 bits are placed in as many consecutive parameter slots as are needed to contain the entire parameter. The rules for allocation and alignment of parameter slots are given later in this section.

The contents of the first eight parameter slots are always passed in registers, while the remaining parameters are always passed on the memory stack, beginning at the caller’s stack pointer plus 16 bytes. The caller uses up to eight of the registers in the output region of its register stack for integer parameters, and up to eight floating-point registers for floating-point parameters.
To accommodate variable argument lists in the C language, there is a fixed correspondence between parameter slots and output registers used for general register arguments. This allows a procedure to spill its register parameters easily to memory before stepping through the parameter list with a pointer. Also because of variable argument lists, floating-point parameters are sometimes passed in both general output registers and in floating-point registers.

There is no fixed correspondence between parameter slots and floating-point parameter registers. Parameters passed in floating-point registers always use the next available floating-point parameter register, starting with f8.

A procedure may assume that the NaT bits on its incoming general register arguments are clear, and that the incoming floating-point register arguments are not NaTVals. A procedure making a call must ensure only that registers containing actual parameters are clear of NaT bits or NaTVals; registers not used for actual parameters may contain garbage.

### 8.5.1 Allocation of Parameter Slots

Parameters slots are allocated for each parameter, based on the parameter type and size, treating each parameter in sequence, from left to right. The rules for allocating parameter slots and placing the contents within the slot are given in Table 8-1.

#### Table 8-1. Rules for Allocating Parameter Slots

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (Bits)</th>
<th>Allocation</th>
<th>Number of Slots</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer/Pointer</td>
<td>1–64</td>
<td>Next Available</td>
<td>1</td>
<td>LSB</td>
</tr>
<tr>
<td>Integer</td>
<td>65–128</td>
<td>Next Even</td>
<td>2</td>
<td>LSB</td>
</tr>
<tr>
<td>Single-Precision Floating-Point</td>
<td>32</td>
<td>Next Available</td>
<td>1</td>
<td>LSB</td>
</tr>
<tr>
<td>Double-Precision Floating-Point</td>
<td>64</td>
<td>Next Available</td>
<td>1</td>
<td>LSB</td>
</tr>
<tr>
<td>Double-Extended Floating-Point</td>
<td>80</td>
<td>Next Even</td>
<td>2</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Quad-Precision Floating-Point</td>
<td>128</td>
<td>Next Even</td>
<td>2</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Aggregates</td>
<td>(size+63)/64</td>
<td>Next Aligned</td>
<td>(size+63)/64</td>
<td>Byte 0</td>
</tr>
</tbody>
</table>

**NOTE:** These rules are applied based on the type of the parameter after any type promotion rules specified by the language have been applied. For example, a short integer passed without a function prototype in C would be promoted to the int type, and would be passed according to the rules for the int type.

The allocation column of the table indicates how parameter slots are allocated for each type of parameter.

- “Next Available” means that the parameter is placed in the slot immediately following the last slot used.
- “Next Even” means that the parameter is placed in the next available even-numbered slot, skipping an odd-numbered slot if necessary. If an odd-numbered slot is skipped, it will not be used for any subsequent parameters.
- “Next Aligned” means that the allocation is dependent on the external alignment of the aggregate; that is, on the alignment boundary required for the aggregate as a whole. For aggregates with an external alignment of 1–8 bytes, the “Next Available” policy is used; for aggregates with an external alignment of 16 bytes, the “Next Even” policy is used.

This placement policy ensures that parameters will fall on a natural alignment boundary if passed in memory.
The alignment column of the table indicates how parameters are aligned within a parameter slot. There are two kinds of alignment, “LSB” and “Byte 0.”

- “LSB” alignment specifies that the least-significant bit of the parameter is aligned with the least-significant bit of the argument slot or slots (i.e., right aligned). Parameters shorter than 64 or 128 bits are padded on the left; the padding is undefined (unless specified otherwise). When a pair of parameter slots is required, the even-numbered parameter slot contains the most-significant bits in big-endian environments, and the least-significant bits in little-endian environments. See Figure 8-4 for examples.

- “Byte 0” alignment specifies that byte 0 of the parameter is aligned with byte 0 of the parameter slot. Parameters that are not a multiple of 64 bits in length are padded at the end; the padding is undefined. In big-endian environments, the padding will be at the right end of the final parameter slot; in little-endian environments, the padding will be at the left end of the final parameter slot. See Figure 8-5 for an example.

### 8.5.2 Register Parameters

The first eight parameter slots (64 bytes) are passed in registers, according to the rules in this section.

- These eight argument slots are associated, one-to-one, with the stacked output GRs, as shown in Figure 8-3.

- Integral scalar parameters, quad-precision (128-bit) floating-point parameters, and aggregate parameters in these slots are passed only in the corresponding output GRs. Aggregates consisting solely of floats, of doubles, or of double-extended values are an exception; see below.

- If an aggregate parameter straddles the boundary between slot 7 and slot 8, the part that lies within the first eight slots is passed in GRs, and the remainder is passed in memory; as described in the next section.

Single-precision, double-precision, and double-extended-precision floating-point scalar parameters in these slots are passed according to the available formal parameter information at the point of call (for example, from a function prototype).

If an actual parameter is known to correspond to a floating-point formal parameter, the following rules apply:

- The actual parameter is passed in the next available floating-point parameter register, if one is available. Floating-point parameter registers are allocated as needed from the range f8–f15, starting with f8.

- If all available floating-point parameter registers have been used, the actual parameter is passed in the appropriate general register(s). (This case can occur only as a result of homogeneous floating-point aggregates, described below.)

If a floating-point actual parameter is known to correspond to a variable-argument specification in the formal parameter list, the following rule applies:

- The actual parameter is passed in the appropriate general register(s).

If the compiler cannot determine, at the point of call, whether the corresponding formal parameter is a varargs parameter, it must generate code that satisfies both of the above conditions. (The compiler’s determination may be based on prototype declarations, language standard assumptions, analysis, or other user options or information.)
When floating-point parameters are passed in floating-point registers, they are passed in the register format, rounded to the appropriate precision. When passed in general registers, floating-point values are passed in their memory format.

Parameters allocated beyond the eighth parameter slot are never passed in registers, even when floating-point parameter registers remain unused.

**Figure 8-4. Examples of “LSB” Alignment**
Aggregates whose elements are all single-precision, all double-precision, or all double-extended-precision values (but not quad-precision), are treated specially. These “homogeneous floating-point aggregates” (HFAs) may be arrays of one of these types, structures whose only members are all one of these types, or structures that contain other structures, provided that all lowest-level members are one of these types, and all are the same type. (This definition includes Fortran COMPLEX data, except COMPLEX*32.)

The following additional rules apply to these types of parameters (but only to the portion of an aggregate that lies within the first eight argument slots):

- If an actual parameter is known to correspond to an HFA formal parameter, each element is passed in the next available floating-point argument register, until the eight argument registers are exhausted. The remaining elements of the aggregate are passed in output GRs, according to the normal conventions.

- If an actual parameter is known to correspond to a variable-argument specification, the aggregate is passed as any other aggregate.

If the compiler cannot determine, at the point of call, whether the corresponding formal parameter is a varargs parameter, the elements of the aggregate must be passed in both the corresponding output GRs and in floating-point argument registers.

**Note:** Because HFAs are mapped to parameter slots as aggregates, single-precision HFAs will be allocated with two floating-point values in each parameter slot, but only one value per register. Thus, the available floating-point parameter registers may become exhausted before the end of the first eight parameter slots, and additional members of the HFA must be passed in general registers.

It is possible for the first of two values in a parameter slot to occupy the last available floating-point parameter register. In this case, the second value is passed in its designated GR, but the half of the GR that would have contained the first value is undefined.
8.5.3 Memory Stack Parameters

The remainder of the parameter list, beginning with slot 8, is passed in the outgoing parameter area of the memory stack frame, as described in Section 7.1, “Procedure Frames” on page 7-1. Parameters are mapped directly to memory, with slot 8 placed at location sp+16, slot 9 at sp+24, and so on. Each argument slot is stored in memory as a 64-bit storage unit according to the byte order of the current environment.

8.5.4 Variable Argument Lists

The rules above support variable-argument list functions in both the K&R and the ANSI dialects of the C language. When an ANSI prototype is in scope, any register parameters corresponding to a variable-argument specification are passed in GRs. When no prototype is in scope, a strict ANSI compilation may pass parameters as if a non-variable argument prototype were in scope, while a K&R (or more relaxed ANSI) compilation may pass floating-point parameters in both GRs and FRs to deal with the possibility that the callee may be expecting either a variable or a non-variable argument list.

Thus, a function with variable arguments may assume that the variable arguments that lie within the first eight argument slots can all be found in the stacked input GRs, in0–in7. It may then store these registers to memory, using the 16-byte scratch area for in6 and in7, and using up to 48 bytes at the base of its own stack frame for in0–in5, as necessary. This arrangement places all the variable parameters in one contiguous block of memory.

When storing registers to memory for this purpose, the code must use the st8.spill instruction, since the registers are not guaranteed to contain valid values.

In a big-endian environment, the alignment and padding rules require the code that steps through the argument list to distinguish between aggregates and integers smaller than 8 bytes. Aggregates will be left-aligned within an 8-byte slot, while integers will be right-aligned.

Examples of the macros from the <stdarg.h> header file are given in Appendix A.

8.5.5 Pointers to Formal Parameters

Whenever the address is formed of a formal parameter that is passed in a register, the compiler must store the parameter to the stack, as it would for a variable argument list.

8.5.6 Languages Other than C

Most languages other than C can usually be treated as if prototypes are always in scope, avoiding the need to pass floating-point parameters in both GRs and FRs. For example, because Fortran passes floating-point parameters by value only when calling an intrinsic function, it may safely assume that the callee is expecting the parameter in an FR.

A compiler for another language may need to honor the variable-argument list conventions, however, if it provides a mechanism for calling C procedures that may have variable-argument lists.
8.5.7 Rounding Floating-point Values

Floating-point parameters passed in floating-point registers should always be explicitly rounded to the proper precision expected by the language. There should be no difference in behavior between a floating-point parameter passed directly in registers and a floating-point parameter that has been stored to memory and reloaded.

8.5.8 Examples

The following examples illustrate the parameter passing conventions.

Scalar integers and floats, with prototype:

extern int func(int, double, double, int);
func(i, a, b, j);

The parameters are passed as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>a</td>
<td>f8</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>f9</td>
</tr>
<tr>
<td>j</td>
<td></td>
<td>out3</td>
</tr>
</tbody>
</table>

Scalar integers and floats, without prototype:

extern int func();
func(i, a, b, j);

The parameters are passed as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>a</td>
<td>out1 and f8</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>out2 and f9</td>
</tr>
<tr>
<td>j</td>
<td></td>
<td>out3</td>
</tr>
</tbody>
</table>

Aggregates passed by value:

extern int func();
struct { int array[20]; } a;
func(i, a);

The structure’s external alignment is only 4 bytes, so no padding is required in the parameter list. The parameters are passed as follows:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>a.array[0–13]</td>
<td>out1–out7</td>
</tr>
<tr>
<td></td>
<td>a.array[14–19]</td>
<td>In memory, at sp+16 through sp+39</td>
</tr>
</tbody>
</table>

Aggregates passed by value:

extern int func();
struct { __float128 x; int array[20]; } a;
func(i, a);
The structure’s external alignment is 16 bytes, so parameter slot 1 is skipped. The parameters are passed as follows:

```
i out0
a.x out2–out3
a.array[0–7] out4–out7
a.array[8–19] In memory, at sp+16 through sp+63
```

**Floating-point aggregates, without prototype:**

```c
struct s { float a, b, c; } x;
extern func();
func(x);
```

The parameters are passed as follows:

```
x.a out0 and f8
x.b out0 and f9
x.c out1 and f10
```

In little-endian environments, x.a and x.c are in the least-significant bits of out0 and out1, respectively, while x.b is in the most-significant bits of out0. In big-endian environments, x.a and x.c are in the most-significant bits of out0 and out1, respectively, while x.b is in the least-significant bits of out0. The figure below illustrates this.

![Diagram showing structure alignment in little-endian and big-endian environments](image_url)

**Floating-point aggregates, with prototype:**

```c
struct s { float a, b, c; } x;
extern void func(struct s);
func(x);
```

The parameters are passed as follows:

```
x.a f8
x.b f9
x.c f10
```
8.6 Return Values

Values up to 256 bits and certain aggregates are returned directly in registers, according to the rules in Table 8-2.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (Bits)</th>
<th>Location of Return Value</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer/Pointer</td>
<td>1–64</td>
<td>r8</td>
<td>LSB</td>
</tr>
<tr>
<td>Integer</td>
<td>65–128</td>
<td>r8, r9</td>
<td>LSB</td>
</tr>
<tr>
<td>Single-Precision Floating-Point</td>
<td>32</td>
<td>f8</td>
<td>N/A</td>
</tr>
<tr>
<td>Double-Precision Floating-Point</td>
<td>64</td>
<td>f8</td>
<td>N/A</td>
</tr>
<tr>
<td>Double-Extended Floating-Point</td>
<td>80</td>
<td>f8</td>
<td>N/A</td>
</tr>
<tr>
<td>Quad-Precision Floating-Point</td>
<td>128</td>
<td>r8, r9</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Single-Precision HFA</td>
<td>32–256</td>
<td>f8–f15</td>
<td>N/A</td>
</tr>
<tr>
<td>Double-Precision HFA</td>
<td>64–512</td>
<td>f8–f15</td>
<td>N/A</td>
</tr>
<tr>
<td>Double-Extended HFA</td>
<td>128–1024</td>
<td>f8–f15</td>
<td>N/A</td>
</tr>
<tr>
<td>Aggregates</td>
<td>1–64</td>
<td>r8</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Aggregates</td>
<td>65–256</td>
<td>r8–r11</td>
<td>Byte 0</td>
</tr>
<tr>
<td>Aggregates</td>
<td>&gt;256</td>
<td>Memory</td>
<td>Byte 0</td>
</tr>
</tbody>
</table>

When multiple registers are used to return a numeric value, the lowest-numbered register contains the most-significant bits in big-endian environments, and the least-significant bits in little-endian environments. When multiple registers are used to return an aggregate, the lowest-numbered register contains the first eight bytes of the aggregate. In big-endian environments, the padding will be at the right end of the final register used; in little-endian environments, the padding will be at the left end of the final register used.

Integral return values smaller than 32 bits must be zero-filled (if unsigned) or sign-extended (if signed) to at least 32 bits.

When floating-point parameters are returned in floating-point registers, they are returned in the register format, rounded to the appropriate precision. When they are returned in general registers (e.g., as part of an aggregate), they are returned in their memory format.

Homogeneous floating-point aggregates, as defined in Section 8.5, are returned in floating-point registers, provided the array or structure contains no more than eight individual values. The elements of the aggregate are placed in successive floating-point registers, beginning with f8. If the array or structure contains more than eight elements, it is returned according to the rule below for aggregates larger than 256 bits.

Return values larger than 256 bits (except HFAs of up to 8 elements) are returned in a buffer allocated by the caller. A pointer to the buffer is passed to the called procedure in r8. This register is not guaranteed to be preserved by the called procedure (that is, the caller must preserve the address of the buffer through some other means). The return buffer must be aligned at a 16-byte boundary. A procedure may assume that the return buffer does not overlap any data that is visible to it through any other names.

A procedure may assume that any procedure it calls will return a valid value (i.e., the NaT bits are clear if the return is in general registers, and floating-point values returned are not NaTVals).
8.7 Requirements for Unwinding the Stack

Certain constraints must be met in order to unwind the stack successfully at any time, both by standard procedure calls as described here, and by special-purpose calling conventions. Chapter 11, “Stack Unwinding and Exception Handling,” describes how the unwind process works and the format of the unwind data structures. To meet the needs of the stack unwind mechanism, the following rules must be followed at all times:

- The previous function state register (ar.pfs) must be preserved prior to any call. The compiler must record, in the unwind data structures, where this register is stored, and over what range of code the saved value is valid.

- For special calls using a return BR other than b0, the compiler must record the BR number used for the return link.

- The return BR must be preserved prior to any call involving the same BR. The compiler must record where the return BR is stored and over what range of code the saved value is valid.

- If a procedure has a memory stack frame, the compiler must record either: (1) how large the frame is, or (2) that a previous frame pointer is stored on the stack or in a general register.

- The return BR must contain an address that can be used to determine the unwind state of the calling procedure. For example, a compiler may choose to optimize calls to procedures that do not return. If it does so, however, it must ensure that the unwind information for the procedure properly describes the unwind state at the return point, even though the return pointer will never be used. This may require the insertion of an otherwise unnecessary nop or break instruction.