**NOTE** There are other user visible registers in some implementations of the architecture, but these are explicitly not part of this processor supplement. A program that uses these registers is not *ABI* compliant and its behavior is undefined.

## The Stack Frame

Each called function in a program allocates a stack frame on the run-time stack, if necessary. A frame is allocated for each non-leaf function and for each leaf function that requires stack storage. A non-leaf function is one that calls other function(s); a leaf function is one that does not itself make any function calls. Stack frames are allocated on the run-time stack; the stack grows downward from high addresses to low addresses.

Each stack frame has sufficient space allocated for:

- local variables and temporaries.
- saved general registers. Space is allocated only for those registers that need to be saved. For non-leaf function, *$31* must be saved. If any of *$16..$23* or *$29..$31* is changed within the called function, it must be saved in the stack frame before use and restored from the stack frame before return from the function. Registers are saved in numerical order, with higher numbered registers saved in higher memory addresses. The register save area must be doubleword (8 byte) aligned.
- saved floating-point registers. Space is allocated only for those registers that need to be saved. If any of *$f20..$f30* is changed within the called function, it must be saved in the stack frame before use and restored from the stack frame before return from the function. Both even- and odd-numbered registers must be saved and restored, even if only single-precision operations are performed since the single-precision operations leave the odd-numbered register contents undefined. The floating-point register save area must be doubleword (8 byte) aligned.
- function call argument area. In a non-leaf function the maximum number of bytes of arguments used to call other functions from the non-leaf function must be allocated. However, at least four words (16 bytes) must always be reserved, even if the maximum number of arguments to any called function is fewer than four words.
- alignment. Although the architecture requires only word alignment, soft-

ware convention and the operating system require every stack frame to be doubleword (8 byte) aligned.

A function allocates a stack frame by subtracting the size of the stack frame from *$sp* on entry to the function. This *$sp* adjustment must occur before *$sp* is used within the function and prior to any jump or branch instructions.

**Figure 3-21: Stack Frame**

| Base | Offset | Contents | Frame |
|---|---|---|---|
| | | unspecified<br>. . .<br>variable size | *High addresses* |
| | +16 | (if present)<br>incoming arguments<br>passed in stack frame | Previous |
| old *$sp* | +0 | space for incoming<br>arguments 1-4 | |
| | | locals and<br>temporaries | |
| | | general register<br>save area | Current |
| | | floating-point<br>register save area | |
| *$sp* | +0 | argument<br>build area | *Low addresses* |

The corresponding restoration of *$sp* at the end of a function must occur after any jump or branch instructions except prior to the jump instruction that returns from the function. It can also occupy the branch delay slot of the jump instruction that returns from the function.

## Standard Called Function Rules

By convention, there is a set of rules that must be followed by every function that allocates a stack frame. Following this set of rules ensures that, given an arbitrary program counter, return address register *$31*, and stack pointer, there is a deterministic way of performing stack backtracing. These rules also make possible programs that translate already compiled absolute code into position-independent code. See Coding Examples in this chapter.

Within a function that allocates a stack frame, the following rules must be observed:

- In position-independent code that calculates a new value for the *gp* register, the calculation must occur in the first three instructions of the function. One possible optimization is the total elimination of this calculation; a local function called from within a position-independent module guarantees that the context pointer *gp* already points to the global offset table. The calculation must occur in the first basic block of the function.

- The stack pointer must be adjusted to allocate the stack frame before any other use of the stack pointer register.

- At most, one frame pointer can be used in the function. Use of a frame pointer is identified if the stack pointer value is moved into another register, after the stack pointer has been adjusted to allocate the stack frame. This use of a frame pointer must occur within the first basic block of the function before any branch or jump instructions, or in the delay slot of the first branch or jump instruction in the function.

- There is only one exit from a function that contains a stack adjustment: a jump register instruction that transfers control to the location in the return address register *$31*. This instruction, including the contents of its branch delay slot, mark the end of function.

- The deallocation of the stack frame, which is done by adjusting the stack pointer value, must occur once and in the last basic block of the function. The last basic block of a function includes all of the non control-transfer instructions immediately prior to the function exit, including the branch delay slot.

## Argument Passing

Arguments are passed to a function in a combination of integer general registers, floating-point registers, and the stack. The number of arguments, their type, and their relative position in the argument list of the calling function determines the mix of registers and memory used to pass arguments. General registers *$4..$7* and floating-point registers *$f12* and *$f14* pass the first few arguments in registers. Double-precision floating-point arguments are passed in the register pairs *$f12*, *$f13* and *$f14*, *$f15*; single-precision floating-point arguments are passed in registers *$f12* and *$f14*.

NOTE

These argument passing rules apply only to languages such as C that do not do dynamic stack allocation of structures and arrays. Ada is an example of a language that does dynamic stack allocation of structures and arrays.

In determining which register, if any, an argument goes into, take into account the following considerations:

- All integer-valued arguments are passed as 32-bit words, with signed or unsigned bytes and halfwords expanded (promoted) as necessary.

- If the called function returns a structure or union, the caller passes the address of an area that is large enough to hold the structure to the function in *$4*. The called function copies the returned structure into this area before it returns. This address becomes the first argument to the function for the purposes of argument register allocation and all user arguments are shifted down by one.

- Despite the fact that some or all of the arguments to a function are passed in registers, always allocate space on the stack for all arguments. This stack space should be a structure large enough to contain all the arguments, aligned according to normal structure rules (after promotion and structure return pointer insertion). The locations within the stack frame used for arguments are called the home locations.

- At the call site to a function defined with an ellipsis in its prototype, the normal calling conventions apply up until the first argument corresponding to where the ellipsis occurs in the parameter list. If, in the absence of the prototype, this argument and any following arguments would have been passed in floating-point registers, they are instead passed in integer registers. Arguments passed in integer registers are not affected by the ellipsis.

  This is the case only for calls to functions which have prototypes containing an ellipsis. A function without a prototype or without an ellipsis in a prototype is called using the normal argument passing conventions.

- When the first argument is integral, the remaining arguments are passed in the integer registers.
- Structures are passed as if they were very wide integers with their size rounded up to an integral number of words. The fill bits necessary for rounding up are undefined.
- A structure can be split so a portion is passed in registers and the remainder passed on the stack. In this case, the first words are passed in *$4*, *$5*, *$6*, and *$7* as needed, with additional words passed on the stack.
- Unions are considered structures.

The rules that determine which arguments go into registers and which ones must be passed on the stack are most easily explained by considering the list of arguments as a structure, aligned according to normal structure rules. Mapping of this structure into the combination of stack and registers is as follows: up to two leading floating-point arguments can be passed in *$f12* and *$f14*; everything else with a structure offset greater than or equal to 16 is passed on the stack. The remainder of the arguments are passed in *$4..$7* based on their structure offset. Holes left in the structure for alignment are unused, whether in registers or in the stack.

The following examples in Figure 3-22 give a representative sampling of the mix of registers and stack used for passing arguments, where d represents double-precision floating-point values, s represents single-precision floating-point values, and n represents integers or pointers. This list is not exhaustive.

See the section “Variable Argument List” later in this section for more information about variable argument lists.

**Figure 3-22: Examples of Argument Passing**

| Argument List | Register and Stack Assignments |
|---|---|
| d1, d2 | *$f12, $f14* |
| s1, s2 | *$f12, $f14* |
| s1, d1 | *$f12, $f14* |
| d1, s1 | *$f12, $f14* |
| n1, n2, n3, n4 | *$4, $5, $6, $7* |
| d1, n1, d2 | *$f12, $6, stack* |
| d1, n1, n2 | *$f12, $6, $7* |
| s1, n1, n2 | *$f12, $5, $6* |
| n1, n2, n3, d1 | *$4, $5, $6, stack* |
| n1, n2, n3, s1 | *$4, $5, $6, $7* |
| n1, n2, d1 | *$4, $5, ($6, $7)* |
| n1, d1 | *$4, ($6, $7)* |
| s1, s2, s3, s4 | *$f12, $f14, $6, $7* |
| s1, n1, s2, n2 | *$f12, $5, $6, $7* |
| d1, s1, s2 | *$f12, $f14, $6* |
| s1, s2, d1 | *$f12, $f14, ($6, $7)* |
| n1, s1, n2, s2 | *$4, $5, $6, $7* |
| n1, s1, n2, n3 | *$4, $5, $6, $7* |
| n1, n2, s1, n3 | *$4, $5, $6, $7* |

In the following examples, an ellipsis appears in the second argument slot.

|  |  |
|---|---|
| n1, d1, d2 | *$4, ($6, $7), stack* |
| *s1, n1* | *$f12,  $5* |
| s1, n1, d1 | *$f12, $5, ($6, $7)* |
| d1, n1 | *$f12, f6* |
| d1, n1, d2 | *$f12,$6, stack* |

## Function Return Values

A function can return no value, an integral or pointer value, a floating-point value (single- or double-precision), or a structure; unions are treated the same as structures.

A function that returns no value (also called procedures or void functions) puts no particular value in any register.

A function that returns an integral or pointer value places its result in register *$2.*

A function that returns a floating-point value places its result in floating-point register *$f0*. Floating-point registers can hold single- or double-precision values.

The caller to a function that returns a structure or a union passes the address of an area large enough to hold the structure in register *$4*. Before the function returns to its caller, it will typically copy the return structure to the area in memory pointed to by *$4*; the function also returns a pointer to the returned structure in register *$2*. Having the caller supply the return object's space allows re-entrancy.

**NOTE** Structures and unions in this context have fixed sizes. The *ABI* does not specify how to handle variable sized objects.

Both the calling and the called function must cooperate to pass the return value successfully:

- The calling function must supply space for the return value and pass its address in the stack frame.
- The called function must use the address from the frame and copy the return value to the object so supplied.

Failure of either side to meet its obligations leads to undefined program behavior.

**NOTE** These rules for function return values apply to languages such as C, but do not necessarily apply to other languages. Ada is one language to which the rules do not apply.