Function Calling Sequence

This section describes 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.

CPU Registers

The MIPS I ISA specifies 32 general purpose 32-bit registers; two special 32-bit registers that hold the results of multiplication and division instructions; and a 32-bit program counter register. The general registers have the names *$0..$31*. By convention, there is also a set of software names for some of the general registers. Figure 3-18 describes the conventions that constrain register usage. Figure 3-19 describes special CPU registers.

NOTE Not all register usage conventions are described. In particular, register usage conventions in languages other than C are not included, nor are the effects of high optimization levels. These conventions do not affect the interface to the system libraries described in Chapter 6.

Figure 3-18: General CPU Registers

Register Name	Software Name	Use
$0	`zero`	always has the value 0.
$at	`AT`	temporary generally used by assembler.
$2..$3	`v0—v1`	used for expression evaluations and to hold the integer and pointer type function return values.
$4..$7	`a0—a3`	used for passing arguments to functions; values are not preserved across function calls. Additional arguments are passed on the stack, as described below.
$8-$15	`t0—t7`	temporary registers used for expression evaluation; values are not preserved across function calls.
$16-$23	`s0—s7`	saved registers; values are preserved across function calls.
$24..$25	`t8—t9`	temporary registers used for expression evaluations; values are not preserved across function calls. When calling position independent functions *$25* must contain the address of the called function.
$26-$27	`kt0—kt1`	used only by the operating system.
$28 or $gp	`gp`	global pointer and context pointer.
$29 or $sp	`sp`	stack pointer.
$30	`s8`	saved register (like `s0-s7`).
$31	`ra`	return address. The return address is the location to which a function should return control.

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, software 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 . . . variable size	*High addresses*
	+16	(if present) incoming arguments passed in stack frame	Previous
old *$sp*	+0	space for incoming arguments 1-4	
		locals and temporaries	
		general register save area	Current
		floating-point register save area	
$sp	+0	argument 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