*Chapter 2*

# Language Implementation Differences

This chapter describes where the 32-bit and 64-bit compilers differ with respect to calling conventions and language implementations. The first section describes the 64-bit subprogram interface. The next two sections identify differences in the 32-bit and 64-bit implementations of the Fortran 77 and C programming languages, respectively.

## Native 64-Bit Subprogram Interface for MIPS Architectures

This section describes the internal subprogram interface for native 64-bit programs. It assumes some familiarity with the current 32-bit interface conventions as specified in the MIPS application binary interface (ABI). The transition to native 64-bit code on the MIPS R8000 requires subprogram interface changes due to the changes in register and address size.

The principal interface for 64-bit native code is similar to the 32-bit ABI standard, with all 32-bit objects replaced by 64-bit objects. In particular, this implies:

- All integer parameters are promoted (that is, sign- or zero-extended to 64-bit integers and passed in a single register). Normally, no code is required for the promotion.
- All pointers and addresses are 64-bit objects.
- Floating point parameters are passed as single- or double-precision according to the ANSI C rules.
- All stack parameter slots become 64-bit doublewords, even for parameters that are smaller (for example, floats and 32-bit integers).

In more detail, the 64-bit native calling sequence has the following characteristics. Square brackets are used to indicate different 32-bit ABI conventions.

- All stack regions are quadword-aligned. [The 32-bit ABI specifies only doubleword alignment.]

- Up to eight integer registers **($4 .. $11 )** may be used to pass integer arguments. [The 32-bit ABI uses only the four registers *$4 .. $7*.]
- Up to eight floating point registers ( *$f12 .. $f19* ) may be used to pass floating point arguments. [The 32-bit ABI uses only the four registers *$f12 .. $f15*, with the odd registers used only for halves of double-precision arguments.]
- The argument registers may be viewed as an image of the initial eight doublewords of a structure containing all of the arguments, where each of the argument fields is a multiple of 64 bits in size with doubleword alignment. The integer and floating point registers are distinct images, that is, the first doubleword is passed in either *$4* or *$f1*, depending on its type; the second in either *$5* or *$f1*; and so on. [The 32-bit ABI associates each floating point argument with an even/odd pair of integer or floating point argument registers.]
- Within each of the 64-bit save area slots, smaller scalar parameters are right-justified, that is, they are placed at the highest possible address (for big-endian targets). This is relevant to float parameters and to integer parameters of 32 or fewer bits. Of these, only **int** parameters arise in C except for prototyped cases – **floats** are promoted to **doubles**, and smaller integers are promoted to **int**. [This is true for the 32-bit ABI, but is relevant only to prototyped small integers since all the other types were at least register-sized.]
- 32-bit integer (*int*) parameters are always sign-extended when passed in registers, whether of signed or unsigned type. [This issue does not arise in the 32-bit ABI.]
- Quad-precision floating point parameters (C **long double** or Fortran **REAL*16**) are always 16-byte aligned. This requires that they be passed in even-odd floating point register pairs, even if doing so requires skipping a register parameter and/or a 64-bit save area slot. [The 32-bit ABI does not consider long double parameters, since they were not supported.]
- **Structs**, **unions**, or other composite types are treated as a sequence of doublewords, and are passed in integer or floating point registers as though they were simple scalar parameters to the extent that they fit, with any excess on the stack packed according to the normal memory layout of the object. More specifically:
  - Regardless of the **struct** field structure, it is treated as a sequence of 64-bit chunks. If a chunk consists solely of a double float field (but not a **double**, which is part of a **union**), it is passed in a floating point register. Any other chunk is passed in an integer register .

  - A **union**, either as the parameter itself or as a **struct** parameter field, is treated as a sequence of integer doublewords for purposes of assignment to integer parameter registers. No attempt is made to identify floating point components for passing in floating point registers.
  - Array fields of **structs** are passed like **unions**. Array parameters are passed by reference (unless the relevant language standard requires otherwise).
  - All **struct** parameters are always left-justified. This applies both to the case of a **struct** smaller than 64 bits, and to the final chunk of a **struct** which is not an integral multiple of 64 bits in size. The implication of this rule is that the address of the first chunk's save area slot is the address of the **struct**, and the **struct** is laid out in the save area memory exactly as if it were allocated normally (once any part in registers has been stored to the save area). [These rules are analogous to the 32-bit ABI treatment – only the chunk size and the ability to pass double fields in floating point registers are different.]

- Whenever possible, floating point arguments are passed in floating point registers regardless of whether they are preceded by integer parameters. [The 32-bit ABI allows only leading floating point (FP) arguments to be passed in FP registers; those coming after integer registers must be moved to integer registers.]
- Variable argument routines require an exception to the previous rule. Any floating point parameters in the variable part of the argument list (leading or otherwise) are passed in integer registers. There are several important cases involved:
  - If a **varargs** prototype (or the actual definition of the callee) is available to the caller, it places floating point parameters directly in the integer register required, and there are no problems.
  - If no prototype is available to the caller for a direct call, the caller's parameter profile is provided in the object file (as are all global subprogram formal parameter profiles), and the linker (*ld/rld*) generates an error message if the linked entry point turns out to be a *varargs* routine.

    **Note:** The linker could insert a stub if required to move floating point parameters to the corresponding integer registers, but we do not intend to do so currently, since ANSI requires prototypes for *varargs* routines.
  - If no prototype is available to the caller for an indirect call (that is, via a function pointer), the caller assumes that the callee is not a *varargs* routine and places floating point parameters in floating point registers (if the callee is *varargs*, it is not ANSI-conformant).

- The portion of the argument structure beyond the initial eight doublewords is passed in memory on the stack and pointed to by the stack pointer at the time of call. The caller does not reserve space for the register arguments; the callee is responsible for reserving it if required (either adjacent to any caller-saved stack arguments if required, or elsewhere as appropriate.) No requirement is placed on the callee either to allocate space and save the register parameters, or to save them in any particular place. [The 32-bit ABI requires the caller to reserve space for the register arguments as well.]
- Function results are returned in **$2** (and **$3** if needed), or **$f0** (and **$f2** if needed), as appropriate for the type. Composite results (**struct**, **union**, or **array**) are returned in **$2**/**$f0** and **$3**/**$f2** according to the following rules:
  - A **struct** with only one or two floating point fields is returned in **$f0** (and **$f2** if necessary). This is a generalization of the Fortran **COMPLEX** case.
  - Any other *struct* or *union* results of at most 128 bits are returned in **$2** (first 64 bits) and **$3** (remainder, if necessary).
  - Larger composite results are handled by converting the function to a procedure with an implicit first parameter, which is a pointer to an area reserved by the caller to receive the result. [The 32-bit ABI requires that all composite results be handled by conversion to implicit first parameters. The MIPS/SGI Fortran implementation has always made a specific exception to return *COMPLEX* results in the floating point registers.]
- There are eight callee-saved floating point registers, **$f24..$f31**. [The 32-bit ABI specifies the six even registers, or even/odd pairs, **$f20..$f31**.]
- Routines are not be restricted to a single exit block. [The 32-bit ABI makes this restriction, though it is not observed under all optimization options.]

There is no restriction on which register must be used to hold the return address in exit blocks. The **.mdebug** format was unable to cope with return addresses in different places, but the DWARF format can. [The 32-bit ABI specifies **$3**, but the implementation supports **.mask** as an alternative.]

PIC (position-independent code, for DSO support) is generated from the compiler directly, rather than converting it later with a separate tool. This allows better compiler control for instruction scheduling and other optimizations, and provides greater robustness.

In the 64-bit interface, **gp** becomes a callee-saved register. [The 32-bit ABI makes **gp** a caller-saved register.]

Table 2-1 specifies the use of registers in native 64-bit mode. Note that "caller-saved" means only that the caller may not assume that the value in the register is preserved across the call.

**Table 2-1** Native 64-Bit Interface Register Conventions

| Register Name | Software Name | Use | Saver |
|---|---|---|---|
| $0 | zero | Hardware zero | |
| $1 or $at | at | Assembler temporary | Caller-saved |
| $2..$3 | v0..v1 | Function results | Caller-saved |
| $4..$11 | a0..a7 | Subprogram arguments | Caller-saved |
| $12..$15 | t4..t7 | Temporaries | Caller-saved |
| $16..$23 | s0..s7 | Saved | Callee-saved |
| $24 | t8 | Temporary | Caller-saved |
| $25 | t9 | Temporary | Caller-saved |
| $26..$27 | kt0..kt1 | Reserved for kernel | |
| $28 or $gp | gp | Global pointer | Callee-saved |
| $29 or $sp | sp | Stack pointer | Callee-saved |
| $30 | s8 | Frame pointer (if needed) | Callee-saved |
| $31 | ra | Return address | Caller-saved |
| hi, lo | | Multiply/divide special registers | Caller-saved |
| $f0, $f2 | | Floating point function results | Caller-saved |
| $f1, $f3 | | Floating point temporaries | Caller-saved |
| $f4..$f11 | | Floating point temporaries | Caller-saved |

**Table 2-1 (continued)** Native 64-Bit Interface Register Conventions

| Register Name | Software Name | Use | Saver |
|---|---|---|---|
| $f12..$f19 | | Floating point arguments | Caller-saved |
| $f20..$f23 | | Floating point temporaries | Caller-saved |
| $f24..$f31 | | Floating point | Callee-saved |

Table 2-2gives several examples of parameter passing. It illustrates that at most eight values can be passed through registers. In the table note that:

- **d1..d5** are double precision floating point arguments
- **s1..s4** are single precision floating point arguments
- **n1..n3** are integer arguments

**Table 2-2** Native 64-Bit C Parameter Passing

| Argument List | Register and Stack Assignments |
|---|---|
| d1,d2 | $f12, $f13 |
| s1,s2 | $f12, $f13 |
| s1,d1 | $f12, $f13 |
| d1,s1 | $f12, $f13 |
| n1,d1 | $4,$f13 |
| d1,n1,d1 | $f12, $5,$f14 |
| n1,n2,d1 | $4, $5,$f14 |
| d1,n1,n2 | $f12, $5,$6 |
| s1,n1,n2 | $f12, $5,$6 |
| d1,s1,s2 | $f12, $f13, $f14 |
| s1,s2,d1 | $f12, $f13, $f14 |
| n1,n2,n3,n4 | $4,$5,$6,$7 |

**Table 2-2 (continued)** Native 64-Bit C Parameter Passing

| Argument List | Register and Stack Assignments |
|---|---|
| n1,n2,n3,d1 | $4,$5,$6,$f15 |
| n1,n2,n3,s1 | $4,$5,$6, $f15 |
| s1,s2,s3,s4 | $f12, $f13,$f14,$f15 |
| s1,n1,s2,n2 | $f12, $5,$f14,$7 |
| n1,s1,n2,s2 | $4,$f13,$6,$f15 |
| n1,s1,n2,n3 | $4,$f13,$6,$7 |
| d1,d2,d3,d4,d5 | $f12, $f13, $f14, $f15, $f16 |
| d1,d2,d3,d4,d5,s1,s2,s3,s4 | $f12, $f13, $f14, $f15, $f16, $f17, $f18,$f19,stack |
| d1,d2,d3,s1,s2,s3,n1,n2,n3 | $f12, $f13, $f14, $f15, $f16, $f17, $10,$11, stack |

## Fortran Implementation Differences

This section lists differences between the 32-bit and the 64-bit Fortran implementations. Command line argument compatibility is described in Chapter 4. The 32-bit Fortran front end is called *fcom* and the 64-bit front end is called *fef77*.

### New Fortran Features

*fef77* implements **REAL*16** and **COMPLEX*32** and all associated intrinsics as 16-byte floating point entities. *fcom* recognizes them, but converts them to **REAL*8** and **COMPLEX*16** respectively.

### Incompatibilities and Differences

- *fcom* and *fef77* are incompatible in the way they fold **REAL*4** constants. *fcom* promotes them internally to **REAL*8**.
- *fcom* allows more constant expressions in parameter statements than *fef77*.

- *fcom* allows lines longer than 256 characters by default*; fef77* currently has a hard limit of 256.
- *fef77* allows parameters (which are ignored) to the program statement.
- *fef77* allows empty parameters in subroutine/function calls, which result in zeroes being passed by value.
- *fef77* implements the Fortran 8*x* array syntax. (Fortran 8*x* is a subset of Fortran 90 that was implemented by various vendors before the Fortran 90 standard was complete).
- *fef77* accepts PCF-style parallel directives in addition to the directives such as **C$DOACROSS,** which *fcom* accepts. PCF-style directives are documented in the *MIPSpro Fortran 77 Programmer's Guide*.

## C Implementation Differences

This section lists differences between the 32-bit and the 64-bit C implementations. Because both compilers adhere to the ANSI standard, and because C is a rigorously defined language designed to be portable, there are not many differences between the 32-bit and 64-bit compiler implementations. The only areas where differences can occur are in data types (by definition) and in areas where ANSI does not define the precise behavior of the language.

Table 2-3 summarizes the differences in data types under the 32-bit and 64-bit data type models.

**Table 2-3** Differences in Data Type Sizes

| C type | 32-bit | 64-bit |
|---|---|---|
| char | 8 | 8 |
| short int | 16 | 16 |
| int | 32 | 32 |
| long int | 32 | 64 |
| long long int | 64 | 64 |
| pointer | 32 | 64 |

**Table 2-3 (continued)** Differences in Data Type Sizes

| C type | 32-bit | 64-bit |
|---|---|---|
| float | 32 | 32 |
| double | 64 | 64 |
| long double[a] | 64 | 128 |

a. On 32-bit compiles the *long double* data type generates a warning message indicating that the *long* qualifier is not supported.

Table 2-3 shows that **long ints**, **pointers** and **long doubles** are different under the two models.

## Structure and Union Layout Examples

Simple examples illustrate the alignment and size issues of C structures and unions.

### Example 1: Structure Smaller Than a Word

```
struct c {
   char c;
} c1;
```

Byte-aligned, **sizeof** struct is 1.

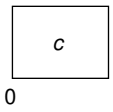


**Figure 2-1** Structure Smaller Than a Word

### Example 2: Structure With No Padding

```
struct s {
   char c;
   char d;
   short s;
   int   i;
} s1;
```

Word-aligned, **sizeof** struct is 8.

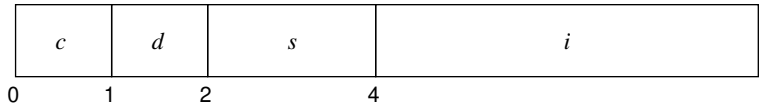


**Figure 2-2** Structure With No Padding

### Example 3: Structure With Internal Padding

```
struct t {
   char c;
   char d;
   short s;
   long  l;
} t1;
```

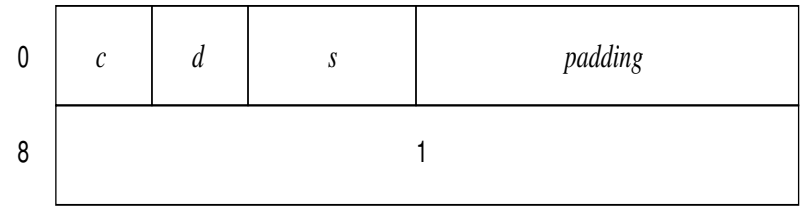


**Figure 2-3** Structure With Internal Padding

### Example 4: Structure With Internal and Tail Padding

```
struct l {
   char c;
   long  l;
   short s;
} l1;
```

**Figure 2-4** Structure With Internal and Tail Padding

**Example 5: Union Allocation**

```
union u {
   char c;
   short s;
   int  i;
   long  l;
} u1;
```

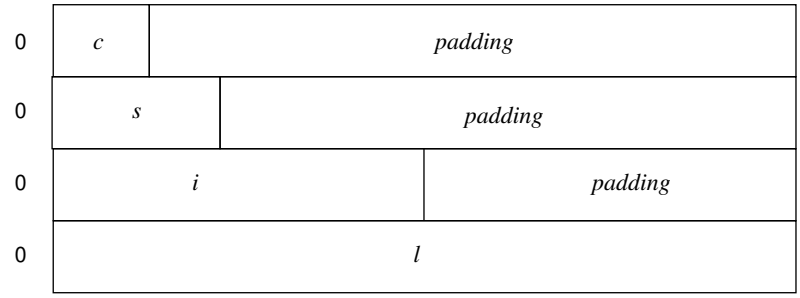


**Figure 2-5** Union Allocation

## Portability Issues

If code was originally written with portability in mind, the type size differences should not be difficult to reconcile. However, production code is often written without regard for portability.When porting code written without regard to portability the following areas should be handled carefully:

- Equivalence of **pointers** and **ints**
- Equivalence of **long ints** and **ints**

- Code without prototypes

These areas a covered in depth in Chapter 3, “Source Code Porting”: