*Chapter 3*

# Source Code Porting

This chapter describes changes you must make to your application source code to port it from a 32-bit to a 64-bit system. The first section outlines changes to Fortran code. The second and third sections deal with C source code issues. The fourth section provide guidelines on writing portable assembly language and C code (with respect to 32-bit and 64-bit systems).

## 64-Bit Fortran Porting Guidelines

This section describes which sections of your Fortran source code you need to modify to port to a 64-bit system.

Standard ANSI Fortran 77 code should have no problems, but the following areas need attention:

- Code that uses **REAL*16** could get different runtime results due to additional accuracy in the QUAD libraries.
- *fcom* and *fef77* are incompatible with regard to real constant folding.
- Integer variables which were used to hold addresses need to be changed to **INTEGER*8**.
- C interface issues (Fortran passes by reference).
- **%LOC** returns 64-bit addresses.
- **%VAL** passes 64-bit values.

Source code modifications are best done in phases. In the first phase, try to locate all of the variables affected by the size issues mentioned above. In the second phase, locate variables that depend on the variables changed in the first phase. Depending on your code, you may need to iterate on this phase as you track down long sets of dependencies.

## Examples of Fortran Portability Issues

The following examples illustrate the variable size issues outlined above:

### Example 1: Changing Integer Variables

Integer variables used to hold addresses must be changed to **INTEGER*8**.

32-bit code:

```
integer iptr, asize

iptr = malloc(asize)
```

64-bit code:

```
integer asize
integer*8 iptr

iptr = malloc(asize)
```

### Example 2: Enlarging Tables

Tables which hold integers used as pointers must be enlarged by a factor of two.

32-bit code:

```
integer tableptr, asize, numptrs
numptrs = 100
asize = 100 * 4

tableptr = malloc(asize)
```

64-bit code:

```
integer asize, numptrs
integer*8 tableptr

numptrs = 100
asize = 100 * 8

tableptr = malloc(asize)
```

#### Example 3: Using #if Directives with Predefined Variables.

You should use *#if* directives so that your source code can be compiled either **-32** or **-64**. The compilers support predefined variables such as *_MIPS_SZPTR* or *_MIPS_SZLONG*, which can be used to differentiate 32-bit and 64-bit source code. A later section provides a more complete list of predefined compiler variables and their values for 32-bit and 64-bit operation. For example, the set of changes in the previous example could be coded:

```
integer asize, numptrs

#if (_MIPS_SZPTR==64)
   integer*8 tablept
   asize = 100 * 8
#else
   integer*4 tableptr
   asize = 100 * 4
#endif

tableptr = malloc(asize)
```

#### Example 4: Storing *%LOC* Return Values

**%LOC** returns 64-bit addresses. You need to use an *INTEGER*8* variable to store the return value of a **%LOC** call.

```
#if (_MIPS_SZLONG==64)
   INTEGER*8 HADDRESS
#else
   INTEGER*4 HADDRESS
#endif


C determine memory location of dummy heap array
     HADDRESS = %LOC(HEAP)
```

#### Example 5: Modifying C Routines Called by Fortran

C routines which are called by Fortran where variables are passed by reference must be modified to hold 64-bit addresses.Typically, these routines used **ints** to contain the addresses in the past. For 64-bit use, at the very least, they should use **long ints**. There are no problems if the original C routines simply define the parameters as **pointers**.

Fortran:

```
call foo(i,j)
```

C:

```
foo_( int *i, int *j) or at least
foo_( long i, long j)
```

#### Example 6: Declaring Fortran Arguments as *long ints*

Fortran arguments passed by *%VAL* calls to C routines should be declared as *long ints* in the C routines.

Fortran:

```
call foo(%VAL(i))
```

C:

```
foo_( long i )
```

#### Example 7: Changing Argument Declarations in Fortran Subprograms

Fortran subprograms called by C where *long int* arguments are passed by address need to change their argument declarations.

C:

```
long l1, l2;
foo_(&l1, &l2);
```

Fortran:

```
subroutine foo(i, j)
#if (_MIPS_SZLONG==64)
   INTEGER*8 i,j
#else
   INTEGER*4 i,j
#endif
```

## 64-Bit C Porting Guidelines

This section details the issues involved in porting 32-bit C application code to 64 bits. It addresses both the porting of existing code and guidelines for writing code to be ported at a later date.

Porting programs written in C to a Silicon Graphics 64-bit MIPS architecture platform, using the native 64-bit C compilers and related tools, should be straightforward. However, depending on assumptions made in writing the original C code, it may require some changes.

The C integer data types historically have been sized based on matching desired functionality with the target architecture's ability to efficiently implement integers of various sizes.

The SGI 64-bit platforms support the LP64 model for native 64-bit programs. In this model, **pointers** and **long** integers are 64 bits.

In the sections below, we discuss what problems to expect in porting existing C code to the LP64 native model, and suggest approaches for avoiding porting problems in new code.

### Porting to the LP64 Model

For code which currently runs on SGI 32-bit platforms, porting to the LP64 model is straightforward. (It may also be unnecessary. Unless the program requires a 64-bit address space, 64-bit data, or other native-only functionality, it may still be run as a 32-bit program.)

Porting requires, at minimum, recompiling and relinking the program. You must specify a 64-bit target if you are doing this on a 32-bit workstation; on a 64-bit workstation this is the default (and you must request the 32-bit compatibility model if you want it). In some cases, the differences between the models imply changes in SGI-provided system header files and/or libraries; in such cases, your selection of the 32-bit or LP64 model selects the correct version automatically.

Within your code, most porting problems arise from assumptions, implicit or explicit, about either absolute or relative sizes of the **int**, **long int**, or **pointer** types. The most common are likely to be:

- sizeof(void *) == 4

  This assumption is analogous to the previous one. But mappings to external data structures should seldom be a problem, since the external definition should also assume 64-bit pointers in the LP64 model.

- constants

  The change in type sizes may yield some surprises related to constants. You should be especially careful about using constants with the high-order (sign) bit set. For instance, the hex constant 0xffffffff yields different results in the expression:

  ```
  long x;
  ... ( (long) ( x + 0xffffffff ) ) ...
  ```

  In both models, the constant is interpreted as a 32-bit *unsigned int*, with value 4,294,967,295. In the 32-bit model, the addition result is a 32-bit **unsigned long**, which is cast to **type long** and has value *x-1* because of the truncation to 32 bits. In the LP64 model, the addition result is a 64-bit *long* with value *x+4,294,967,295*, and the cast is redundant.

- arithmetic assumptions

  Related to some of the above cases, code which does arithmetic (including shifting) which may overflow 32 bits, and assumes particular treatment of the overflow (for example, truncation), may exhibit different behavior in the LP64 model, depending on the mix of types involved (including signedness).

  Similarly, implicit casting in expressions which mix int and long values may behave unexpectedly due to sign/zero extension. In particular, remember that integer constants are sign or zero extended when they occur in expressions with long values.

  Once identified, each of these problems is easy to solve. Change the relevant declaration to one which has the desired characteristics in both target environments, add explicit type casts to force the correct conversions, use function prototypes, or use type suffixes (for example, ‘l' or ‘u') on constants to force the correct type.

## Writing Code Portable to 64-Bit Platforms

The key to writing new code which is compatible with the 32-bit and LP64 data models described is to avoid those problems described above. Since all of the assumptions described sometimes represent legitimate attributes of data objects, this requires some tailoring of declarations to the target machines' data models.

We suggest observing the following guidelines to produce code without the more common portability problems. They can be followed from the beginning in developing new code, or adopted incrementally as portability problems are identified.

In a header file which can be included in each of the program's source files, define (**typedef**) a type for each of the following functions:

- For each specific integer data size required, that is, where exactly the same number of bits is required on each target, define a *signed* and *unsigned* type, for example:
```
typedef signed char int8_t
typedef unsigned char uint8_t
...
typedef unsigned long long uint64_t
```
- If you require a large scaling integer type, that is, one which is as large as possible while remaining efficiently supported by the target, define another pair of types, for example:
```
typedef signed long intscaled_t
typedef unsigned long uintscaled_t
```
- If you require integer types of at least a particular size, but chosen for maximally efficient implementation on the target, define another set of types, similar to the first but defined as larger standard types where appropriate for efficiency.

Having constructed the above header file, use the new **typedef**'ed types instead of the standard C type names. You need (potentially) a distinct copy of this header file (or conditional code) for each target platform supported. As a special case of this, if you are providing libraries or interfaces to be used by others, be particularly careful to use these types (or similar application specific types) chosen to match the specific requirements of the interface. Also in such cases, you should choose the actual names used to avoid name space conflicts with other libraries doing the same thing. If this is done carefully, your clients should be able to use a single set of header files on all targets. However, you generally need to provide distinct libraries (binaries) for the 32-bit compatibility model and the LP64 native model on 64-bit SGI platforms, though the sources may be identical.

Be careful that constants are specified with appropriate type specifiers so that they extend to the size required by the context with the values that you require. Bit masks can be particularly troublesome in this regard:avoid using constants for negative values. For example, 0xffffffff may be equivalent to a -1 on 32-bit systems, but it is interpreted as 4,294,967,295 (*signed* or *unsigned*) on 64-bit systems. The *inttypes.h* header file provides *cpp* macros to facilitate this conversion.

Defining constants which are sensitive to type sizes in a central header file may help in modifying them when a new port is done. Where **printf()**/**scanf()** are used for objects whose types are **typedef**'ed differently among the targets you must support, you may need to define constant format strings for each of the types defined in step (1), for example:

```
#define _fmt32 "%d"
#define _fmt32u "%u"
#define _fmt64 "%ld"
#define _fmt64u "%lu"
```

The *inttypes.h* header file also defines **printf**()/**scanf**() format extensions to standardize these practices.

## Fundamental Types for C

This section discusses 'fundamental types' useful in converting C code from 32-bit to 32- or 64-bit. These take the form of **typedefs**, and are available in the file **<***sgidefs.h***>**. These typedefs are enabled by compiler predefines, which are also described. This discussion is entirely from the C point of view, although the predefines discussed are also emitted by the other compilers.

It is desirable to have source code that can be compiled either in 32-bit mode or 64-bit mode. An example is *libc*, which we provide in both 32-bit and 64-bit form. (In this discussion, 32-bit code means **mips1** or **mips2**, 64-bit code means **mips3** or **mips4**.)

As previously mentioned, the compilation model chosen for 64-bit objects is referred to as LP64, where **longs** and **pointers** are 64 bits, and **ints** remain at 32 bits. Since **ints** and **pointers** are no longer the same size, and **ints** and **longs** are not the same size, a lot of code can break in this compilation model.

The **typedefs** discussed, in their naming convention, explicitly call out certain attributes of the **typedef**. The goal of this, by naming those attributes, is to ease the long term maintenance of code which has to compile in both the 32-bit and 64-bit models.

The **typedefs** are enabled by predefines from the compilers. The predefines that the compilers emit are:

For MIPS1executables:

```
-D_MIPS_FPSET=16
-D_MIPS_ISA=_MIPS_ISA_MIPS1
-D_MIPS_SIM=_MIPS_SIM_ABI32
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=32
-D_MIPS_SZPTR=32
```

For MIPS3 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS3
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

For MIPS4 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS4
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

The explanation of these predefines is as follows:

- MIPS_ISA is Mips Instruction Set Architecture. MIPS_ISA_MIPS1 and MIPS_ISA_MIPS3 would be the most common variants for kernel level assembler code.
- MIPS_ISA_MIPS4 is the ISA for R8000 applications. MIPS_SIM is Mips Subprogram Interface Model -- this describes the subroutine linkage convention and register naming/usage convention.

- _MIPS_FPSET describes the number of floating point registers. The MipsIII compilation model makes use of the extended floating point registers available on the R4000.
- _MIPS_SZINT, _MIPS_SZLONG, and _MIPS_SZPTR describe the size of each of those types.

An example of the use of these predefined variables:

```
#if (_MIPS_SZLONG == 32)
   typedef int    ssize_t;
#endif
#if (_MIPS_SZLONG == 64)
   typedef long    ssize_t;
#endif
```

The **typedefs** following are largely self-explanatory. These are from <*sgidefs.h*>:

```
__int32_t     Signed 32 bit integral type
__uint32_t    Unsigned 32 bit integral type
__int64_t     Signed 64 bit integral type
__uint64_t    Unsigned 64 bit integral type
```

These are “pointer-sized **int**” and “pointer-sized **unsigned int**’ respectively. As such, they are guaranteed to have the same number of bits as a pointer.

```
__psint_t
__psunsigned_t
```

These are 'scaling **int**' and 'scaling **unsigned**' respectively, and are intended for variables that you want to grow as the code is compiled in the 64-bit model.

```
__scint_t
__scunsigned_t
```

The usefulness of these types is that they free the coder from having to know the underlying compilation model -- indeed, that model can change, and the code should still work. In this respect, use of these **typedefs** is better than replacing the assumption, that an **int** and a **pointer** are the same size with the new assumption, that a **long** and a **pointer** are the same size.'

## Assembly Language Coding Guidelines

This section describes techniques for writing assembler code which can be compiled and run as either a 32-bit or 64-bit executable. These techniques are based on using certain predefined variables of the compiler, and on macros defined in *sys/asm.h* and *sys/regdef.h* which rely on those compiler predefines. Together, they enable a fairly easy conversion of existing assembly code to run in either the 32-bit or LP64 compilation model. They also allow retargeted assembler code to look fairly uniform in the way it is converted.

### Overview and Predefined Variables

There are two sets of issues: the LP64 model, and the new calling conventions. Each of these issues is solved by a combination of predefined variables that the compiler emits, and macros in **<***sys/asm.h***>** and **<***sys/regdef.h***>**, that use those predefine variables to define macros appropriately.
The predefines that the assembler emits are:

**For MIPS1/2 executables:**

```
-D_MIPS_FPSET=16
-D_MIPS_ISA=_MIPS_ISA_MIPS1
-D_MIPS_SIM=_MIPS_SIM_ABI32
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=32
-D_MIPS_SZPTR=32
```

**For MIPS3 executables:**

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS3
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

For MIPS4 executables:

```
-D_MIPS_FPSET=32
-D_MIPS_ISA=_MIPS_ISA_MIPS4
-D_MIPS_SIM=_MIPS_SIM_ABI64
-D_MIPS_SZINT=32
-D_MIPS_SZLONG=64
-D_MIPS_SZPTR=64
```

The explanation of these predefined variables is as follows:

- *MIPS_ISA* is MIPS Instruction Set Architecture. *MIPS_ISA_MIPS1* and *MIPS_ISA_MIPS3* would be the most common variants for kernel-level assembler code.
- *MIPS_ISA_MIPS4* is the ISA for R8000 applications. *MIPS_SIM* is MIPS Subprogram Interface Model – this describes the subroutine linkage convention and register naming/usage convention.
- *_MIPS_FPSET* describes the number of floating point registers. The MipsIII compilation model makes use of the extended floating point registers available on the R4000.
- *_MIPS_SZINT*, *_MIPS_SZLONG*, and *_MIPS_SZPTR* describe the size of each of those types.

An example of the use of these macros:

```
#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define SZREG          4
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define SZREG          8
#endif
```

## LP64 Model Implications for Assembly Language Code

Four implications to writing assembly language code for LP64 are:

- The first deals with different register sizes as explained in "Different Register Sizes."
- The second requires you to use a different subroutine linkage convention as explained in "Using a Different Subrouting Linkage."
- The third requires you to use a different convention to save the global pointer register (*$gp*) as explained in "Caller $gp (o32) vs. Callee Saved $gp (LP64)."
- The fourth restricts your use of *lwc1* instructions to access floating point register pairs but allows you to use more floating point registers as described in "Using More Floating Point Registers."

### Different Register Sizes

The MIPSpro 64-bit C compiler generates code in the LP64 model -- that is, **pointers** and **longs** are 64 bits, **ints** are 32 bits. This means that all assembler code which deals with either **pointers** or **longs** needs to be converted to using doubleword instructions for MipsIII/IV code, and must continue to use word instructions for MipsI/II code.

Macros in <*sys/asm.h*>, coupled with the compiler predefines, provide a solution to this problem. These macros look like *PTR_*<*op*> or *LONG_*<*op*>, where *op* is some operation such as *L* for load, or *ADD*, etc.. These *ops* use standard defines such as *_MIPS_SZPTR* to resolve to doubleword opcodes for MIPS3, and word opcodes for MIPS1. There are specific macros for PTR ops, for LONG ops, and for INT ops.

### Using a Different Subrouting Linkage

The second implication of LP64 is that there is a different subroutine linkage convention, and a different register naming convention. The compiler predefine *_MIPS_SIM* enables macros in <*sys/asm.h*> and <*sys/regdef.h*> Some important ramifications of that linkage convention are described below.

In the *_MIPS_SIM_ABI64* model there are 8 argument registers – **$4 .. $11**. These additional 4 argument registers come at the expense of the temp registers in <*sys/regdef.h*>. In this model, there are no registers **t4 .. t7**, so any code using these registers does not compile under this model. Similarly, the register names **a4 .. a7** are not available under the *_MIPS_SIM_ABI32* model. (It should be pointed out that those temporary registers are not lost -- the argument registers can serve as scratch registers also, with certain constraints.)

To make it easier to convert assembler code, the new names **ta0**, **ta1**, **ta2**, and **ta3** are available under both *_MIPS_SIM models*. These alias with **t4 .. t7** in the 32-bit world, and with **a4 ..a7** in the 64-bit world.

Another facet of the linkage convention is that the caller no longer has to reserve space for a called function to store its arguments in. The called routine allocates space for storing its arguments on its own stack, if desired. The *NARGSAVE* define in <*sys/asm.h*> helps with this.

### Caller $gp (o32) vs. Callee Saved $gp (LP64)

The *$gp* register is used to point to the Global Offset Table (GOT). The GOT stores addresses of subroutines and static data for runtime linking. Since each DSO has its own GOT, the *$gp* register must be saved across function calls. Two conventions are used to save the *$gp* register.

Under the first convention, called caller saved *$gp*, each time a function call is made, the calling routine saves the *$gp* and then restores it after the called function returns. To facilitate this two assembly language pseudo instructions are used. The first, *.cpload*, is used at the beginning of a function and sets up the *$gp* with the correct value. The second, *.cprestore*, saves the value of *$gp* on the stack at an offset specified by the user. It also causes the assembler to emit code to restore *$gp* after each call to a subroutine.

The formats for correct usage of the *.cpload* and *.cprestore* instructions are shown below:

*.cpload reg* — *reg* is *t9* by convention

*.cprestore offset* — *offset* refers to the stack offset where *$gp* is saved

Under the second convention, called callee saved *$gp*, the responsibility for saving the *$gp* register is placed on the called function. As a result, the called function needs to save the *$gp* register when it first starts executing. It must also restore it, just before it returns. To accomplish this the *.cpsetup* pseudo assembly language instruction is used. Its usage is shown below:

*.cpsetup reg, offset, proc_name*

*reg* is *t9* by convention
*offset* refers to the stack offset where *$gp* is saved
*proc_name* refers to the name of the subroutine

You must create a stack frame by subtracting the appropriate value from the *$sp* register before using the directives which save the *$gp* on the stack.

In order to facilitate writing assembly language code for both conventions several macros have been defined in <*sys/asm.h*>. The macros **SETUP_GP, SETUP_GPX, SETUP_GP_L,** and **SAVE_GP** are defined under o32 and provide the necessary functionality to support a caller saved *$gp* environment. Under LP64, these macros are null. However, **SETUP_GP64, SETUP_GPX64, SETUP_GPX64_L**, and **RESTORE_GP64** provide the functionality to support a callee saved environment. These same macros are null for o32.

In conclusion, predefines from the compiler enable a body of macros to generate 32/64-bit asm code. Those macros are defined in <*sys/asm.h*>, <*sys/regdef.h*>, and <*sys/fpregdef.h*>

The following example handles assembly language coding issues for LP64 and KPIC (KPIC requires that the *asm* coder deals with PIC issues). It creates a template for the start and end of a generic assembly language routine.

The template is followed by relevant defines and macros from <*sys/asm.h*>.

```
 LOCALSZ=        4               # save a0, a1, ra, gp
 FRAMESZ=        (((NARGSAVE+LOCALSZ)*SZREG)+ALSZ)&ALMASK
 RAOFF=          FRAMESZ-(1*SZREG)
 A0OFF=          FRAMESZ-(2*SZREG)
 A1OFF=          FRAMESZ-(3*SZREG)
 GPOFF=          FRAMESZ-(4*SZREG)

 NESTED(asmfunc,FRAMESZ,ra)
        move t0, gp   # save entering gp
                      # SIM_ABI64 has gp callee save
                      # no harm for SIM_ABI32
        SETUP_GPX(t8)
        PTR_SUBU sp,FRAMESZ
        SETUP_GP64(GPOFF,_sigsetjmp)
        SAVE_GP(GPOFF)
/* Save registers as needed here */
        REG_S ra,RAOFF(sp)
        REG_S a0,A0OFF(sp)
        REG_S a1,A1OFF(sp)
        REG_S t0,T0OFF(sp)

/* do real work here */
/* safe to call other functions */

/* restore saved regsisters as needed here */
        REG_L ra,RAOFF(sp)
        REG_L a0,A0OFF(sp)
        REG_L a1,A1OFF(sp)
        REG_L t0,T0OFF(sp)
```

```
/* setup return address, $gp and stack pointer */
REG_L    ra,RAOFF(sp)
RESTORE_GP64
PTR_ADDU sp,FRAMESZ

        bne      v0,zero,err
        j        ra

        END(asmfunc)
```

The *.cpload/.cprestore* is only used for generating KPIC code -- and tells the assembler to initialize, save, and restore the **gp**.

The following are relevant parts of *asm.h*:

```
#if (_MIPS_SIM == _MIPS_SIM_ABI32)
#define NARGSAVE        4
#define ALSZ            7
#define ALMASK          ~7
#endif
#if (_MIPS_SIM == _MIPS_SIM_ABI64)
#define NARGSAVE        0
#define ALSZ            15
#define ALMASK          ~0xf
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA ==_MIPS_ISA_MIPS2)
#define SZREG           4
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define SZREG           8
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS1 || _MIPS_ISA == _MIPS_ISA_MIPS2)
#define REG_L   lw
#define REG_S   sw
#endif

#if (_MIPS_ISA == _MIPS_ISA_MIPS3 || _MIPS_ISA == _MIPS_ISA_MIPS4)
#define REG_L   ld
#define REG_S   sd
#endif
```

```
#if (_MIPS_SZINT == 32)
#define INT_L   lw
#define INT_S   sw
#define INT_LLEFT       lwl
#define INT_SLEFT       swl
#define INT_LRIGHT      lwr
#define INT_SRIGHT      swr
#define INT_ADD         add
#define INT_ADDI        addi
#define INT_ADDIU       addiu
#define INT_ADDU        addu
#define INT_SUB         sub
#define INT_SUBI        subi
#define INT_SUBIU       subiu
#define INT_SUBU        subu
#define INT_LL          ll
#define INT_SC          sc
#endif

#if (_MIPS_SZINT == 64)
#define INT_L   ld
#define INT_S   sd
#define INT_LLEFT       ldl
#define INT_SLEFT       sdl
#define INT_LRIGHT      ldr
#define INT_SRIGHT      sdr
#define INT_ADD         dadd
#define INT_ADDI        daddi
#define INT_ADDIU       daddiu
#define INT_ADDU        daddu
#define INT_SUB         dsub
#define INT_SUBI        dsubi
#define INT_SUBIU       dsubiu
#define INT_SUBU        dsubu
#define INT_LL          lld
#define INT_SC          scd
#endif

#if (_MIPS_SZLONG == 32)
#define LONG_L  lw
#define LONG_S  sw
#define LONG_LLEFT      lwl
#define LONG_SLEFT      swl
#define LONG_LRIGHT     lwr
#define LONG_SRIGHT     swr
```

```
#define LONG_ADD        add
#define LONG_ADDI       addi
#define LONG_ADDIU      addiu
#define LONG_ADDU       addu
#define LONG_SUB        sub
#define LONG_SUBI       subi
#define LONG_SUBIU      subiu
#define LONG_SUBU       subu
#define LONG_LL         ll
 #define LONG_SC         sc
 #endif

 #if (_MIPS_SZLONG == 64)
 #define LONG_L  ld
 #define LONG_S  sd
 #define LONG_LLEFT      ldl
 #define LONG_SLEFT      sdl
 #define LONG_LRIGHT     ldr
 #define LONG_SRIGHT     sdr
 #define LONG_ADD        dadd
 #define LONG_ADDI       daddi
 #define LONG_ADDIU      daddiu
 #define LONG_ADDU       daddu
 #define LONG_SUB        dsub
 #define LONG_SUBI       dsubi
 #define LONG_SUBIU      dsubiu
 #define LONG_SUBU       dsubu
 #define LONG_LL         lld
 #define LONG_SC         scd
 #endif

 #if (_MIPS_SZPTR == 32)
 #define PTR_L   lw
 #define PTR_S   sw
 #define PTR_LLEFT       lwl
 #define PTR_SLEFT       swl
 #define PTR_LRIGHT     lwr
 #define PTR_SRIGHT      swr
 #define PTR_ADD         add
 #define PTR_ADDI        addi
 #define PTR_ADDIU       addiu
 #define PTR_ADDU        addu
 #define PTR_SUB         sub
 #define PTR_SUBI        subi
 #define PTR_SUBIU       subiu
```

```
#define PTR_SUBU        subu
#define PTR_LL          ll
#define PTR_SC          sc
#endif

#if (_MIPS_SZPTR == 64)
#define PTR_L   ld
#define PTR_S   sd
#define PTR_LLEFT       ldl
#define PTR_SLEFT       sdl
#define PTR_LRIGHT      ldr
#define PTR_SRIGHT      sdr
#define PTR_ADD         dadd
#define PTR_ADDI        daddi
#define PTR_ADDIU       daddiu
#define PTR_ADDU        daddu
#define PTR_SUB         dsub
#define PTR_SUBI        dsubi
#define PTR_SUBIU       dsubiu
#define PTR_SUBU        dsubu
#define PTR_LL          lld
#define PTR_SC          scd
#endif
```

### Using More Floating Point Registers

On the R4000 and later generation MIPS microprocessors, the FPU provides:

- 16 64-bit Floating Point registers (FPRs) each made up of a pair of 32-bit floating point general purpose register when the FR bit in the Status register equals 0, or
- 32 64-bit Floating Point registers (FPRs) each corresponding to a 64-bit floating point general purpose register when the FR bit in the Status register equals 1

For more information about the FPU of the R4000 refer to Chapter 6 of the *MIPS R4000 User's Manual*.

Under o32, the FR bit is set to 0. As a result, o32 provides only 16 registers for double precision calculations. Under o32, double precision instructions must refer to the even numbered floating point general purpose register. A major implication of this is that code written for the MIPS I instruction set treated a double precision floating point register as an odd and even pair of single precision floating point registers. It would typically use sequences of the following instructions to load and store double precision registers.

```
lwc1 $f4, 4(a0)
lwc1 $f5, 0(a0)
...
swc1 $f4, 4(t0)
swc1 $f5, 0(t0)
```

Under LP64, however, the FR bit is set to 1. As a result, LP64 provides all 32 floating point general purpose registers for double precision calculations. Since *$f4* and *$f5* refer to different double precision registers, the code sequence above will not work under LP64. It can be replaced with the following:

```
l.d $f14, 0(a0)
...
s.d $f14, 0(t0)
```

The assembler will automatically generate pairs of LWC1 instructions for MIPS I and use the LDC1 instruction for MIPS II and above.

On the other hand, you can use these additional odd numbered registers to improve performance of double precision code.