# MIPSpro™ 64-Bit Porting and Transition Guide

Document Number 007-2391-006

CONTRIBUTORS

Written by George Pirocanac
Illustrated by Dany Galgani
Production by Carlos Miqueo
Updated by Jean Wilson
Engineering contributions by Dave Anderson, Bean Anderson, Dave Babcock, Jack Carter, Ann Chang, Wei-Chau Chang, Steve Cobb, Rune Dahl, Jim Dehnert, David Frederick, Jay Gischer, Bob Green, W. Wilson Ho, Peter Hsu, Bill Johnson, Dror Maydan, Ash Munshi, Michael Murphy, Bron Nelson, Paul Rodman, John Ruttenberg, Ross Towle, Chris Wagner
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image courtesy of Xavier Berenguer, Animatica.



MIPSpro™ 64-Bit Porting and Transition Guide
Document Number 007-2391-006

# Contents

# List of Figures

# List of Tables

*Chapter 1*

# 64-bit ABI and Compiler Overview

This chapter gives a brief overview of the 64-bit application binary interface (ABI) and describes the MIPSpro 7.3 32-bit, 64-bit and high performance 32-bit (N32) compilers. It contains six sections:

- The first section introduces the 64-bit ABI and describes the compatibility of old IRIX 5.*x* programs with IRIX 6.x (see Figure 1-1).
- The second section describes the various components of both the 64-bit compiler system and the 32-bit compiler.
- The third section gives an overview of interprocedural analysis (IPA).
- The fourth section gives an overview of the loop nest optimizer (LNO).
- The fifth section describes the Auto-Parallelizing Option which replaces the Kuck and Associates Preprocessor (KAP), as a means of converting programs to parallel code.
- The last section introduces the multiprocessing support provided by the MIPSpro compilers.

## 64-bit ABI Overview

Three different ABIs are currently supported on IRIX platforms:

| | |
|---|---|
| o32 | The old 32-bit ABI generated by the *ucode* compiler. |
| n32 | The 32-bit ABI generated by the MIPSpro 64-bit compiler. N32 is described in the *MIPSpro N32 ABI Handbook*. |
| n64 | The 64-bit ABI generated by the MIPSpro 64-bit compiler. |

The 64-bit ABI was introduced in IRIX 6.0; it was designed to exploit the high performance capabilities and 64-bit virtual addressing provided by the MIPS R8000 processor. These capabilities include:

- The ability to execute MIPS1 user code, compatible with the R3000.
- The ability to execute MIPS2 instruction set extensions introduced in the R4000.
- The ability to execute MIPS3 64-bit addressing and instructions introduced in the R4400.
- The ability to execute new instructions which improved floating point and integer performance (MIPS4 instructions).

The MIPS3 and MIPS4 64-bit capabilities provide both 64-bit virtual addressing and instructions which manipulate 64-bit integer data. Processor registers are 64 bits in size. Also provided is the ability to use 32 64-bit floating point registers.

Table 1-1 compares the various ABIs.

**Table 1-1** ABI Comparison Summary

| | o32 | n32 | n64 |
|---|---|---|---|
| Compiler Used | ucode | MIPSpro | MIPSpro |
| Integer Model | ILP32 | ILP32 | LP64 |
| Calling Convention | mips | new | new |
| Number of FP Registers | 16 (FR=0) | 32 (FR=1) | 32 (FR=1) |

**Table 1-1 (continued)** ABI Comparison Summary

| | o32 | n32 | n64 |
|---|---|---|---|
| Number of Argument Registers | 4 | 8 | 8 |
| Debug Format | mdbug | dwarf | dwarf |
| ISAs Supported | mips1/2 | mips3/4 | mips3/4 |
| 32/64 Mode | 32 (UX=0) | 64 (UX=1) * | 64 (UX=1) |

* UX=1 implies 64-bit registers and also indicates that MIPS3 and MIPS4 instructions are legal. N32 uses 64-bit registers but restricts addresses to 32 bits.

## Compatibility and Supported ABIs

All versions of IRIX 6.x support development for o32, n32 and n64 programs. All IRIX 6.x systems also support execution of o32 and n32 programs. However, in order to execute 64-bit programs you must be running on IRIX 6.4 or a 64-bit version of IRIX 6.2 or IRIX 6.5. IRIX 6.3 and the 32-bit version of IRIX 6.2 or IRIX 6.5 do not support execution of 64-bit programs. You can tell if you are running on a system capable of executing 64-bit programs by running the *uname* command. If it returns *IRIX64*, you are on a 64-bit version of IRIX. If it returns *IRIX*, you are on a 32-bit version.

On 64-bit versions of IRIX you can execute programs conforming to any of the following Application Binary Interfaces (ABIs):

- An o32 program built under IRIX 5.x or IRIX 6.x (32-bit MIPS1 or MIPS2 ABI). COFF is no longer supported as of IRIX 6.2.
- A 64-bit program (64-bit MIPS3 or MIPS4 ABI).
- An N32 program (N32 MIPS3 or MIPS4 ABI).

Figure 1-1 illustrates the ABIs supported by IRIX 6.x.

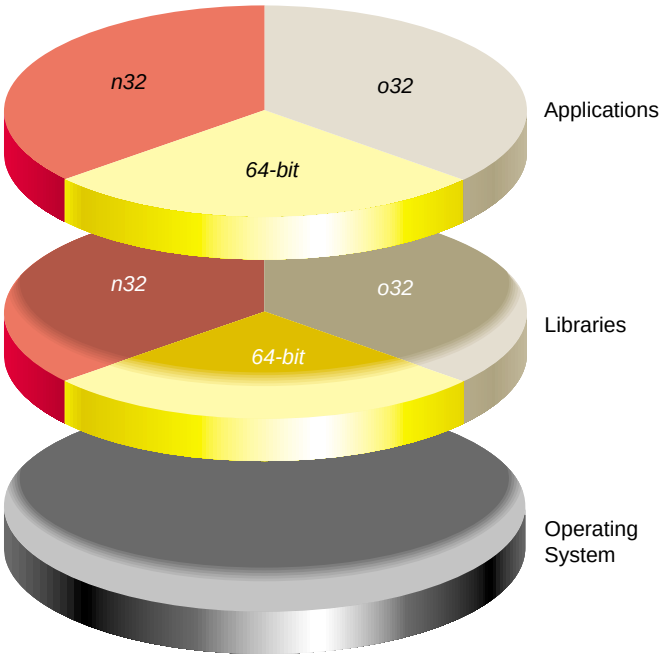


**Figure 1-1** ABIs supported by IRIX 6.x

More specifically, the execution and development environments under IRIX 6.x provide the following functionality:

- 32-bit IRIX 5.*x* binaries and Dynamic Shared Objects (DSOs) execute under IRIX 6.x
- IRIX 6.x has a set of compilers (32-bit Native Development Environment) that generate 32-bit code. You can mix objects and link with objects produced on IRIX 5.*x*. (There is no guarantee, however, that this code runs on IRIX 5.*x* systems.)
- IRIX 6.x also has a set of compilers (64-bit Native Development Environment) that generates either 64-bit or N32 code. This code **cannot** run on IRIX 5.x systems.
- You can specify which compiler you want to run by using the **-64**, **-n32** or **-32** (**-o32**) flags on the compiler command line.

  The compiler driver then executes the appropriate compiler binaries and links with the correct libraries. This also applies to the assembler, linker, and archiver. If these switches are not present, the driver checks for an */etc/compiler.defaults* file and an environment variable, *SGI_ABI*, for these values. See the compiler man pages for more details.

- All of the compiler-related tools (*dbx*, *nm*, *dis*) can work with either 32-bit, N32 or 64-bit binaries. *Prof* functionality is rolled into the *SpeedShop* product line.
- You **cannot** mix objects and DSOs produced by the 32-bit compilers with objects and DSOs produced by the 64-bit compilers. In Figure 1-1, this is illustrated by the lines separating the 32-bit, N32 and 64-bit libraries. Therefore, the following rules apply:
  - You **cannot** link 32-bit objects with 64-bit or N32 objects and shared libraries
  - You **cannot** link 64-bit objects with 32-bit or N32 objects and shared libraries
  - You **cannot** link N32 objects with 64-bit or 32-bit objects and shared libraries
- The */usr/lib* directory on IRIX 6.x systems contains the 32-bit libraries and *.so*s. The 64-bit *.so*s are located in */usr/lib64*. The N32 *.so*s are located in */usr/lib32*. The complete layout looks like this:

  **32-bit**: This is the IRIX 5.x */usr/lib*, including compiler components:

```
/usr/lib/
          *.so
          mips2/
          *.so
/usr/lib/
          cfe
          fcom
          ugen
          uopt
          as
```

  **64-bit**: These are the 64-bit-specific libraries:

```
/usr/lib64/
          *.so
          mips3/
          *.so
          mips4/
          *.so
```

  **N32**: These are the N32-specific libraries and components:

```
/usr/lib32/
          *.so
          mips3/
          *.so
          mips4/
          *.so
```

```
/usr/lib32/cmplrs
          be
          fec
          mfef77
          as
```

### Known Compatibility Issues

The following issues are known to cause errors for 32-bit programs running on IRIX 6.x:

- Any access to kernel data structures, for example, through */dev/kmem*. Many of these structures have changed in size. Programs making these kinds of accesses must be ported to 64-bit. 32-bit programs cannot access all of kernel memory and should probably also be ported to 64-bit.
- Use of **nlist()** does not work on any 64-bit *.o* or *a.out*. A new **nlist64()** is supplied for 64-bit ELF.
- Any assumption that the page size is 4Kbytes (for example, using **mmap()** and specifying the address). The page size is no longer 4Kbytes. Programs must use **getpagesize()**.
- Ada programs which catch floating point exceptions do not work.
- Any program using */proc* must have some interfaces changed.

It is possible for a program to determine if it is running on a 64-bit capable kernel in order to work around the issues listed above. Use **sysconf(_SC_KERN_POINTERS)**, which returns 32 or 64.

## Compiler System Components

As explained earlier, the MIPSpro compiler system on IRIX 6.x consists of two independent compiler systems. One system supports the 64-bit and high performance 32-bit (N32) ABIs. The other supports the old 32-bit ABI. This section describes and compares them.

### Fortran

The MIPSpro Fortran 77 compiler and the MIPSpro 7 Fortran 90 compiler support 32-bit, 64-bit and N32 compiler modes. The following section describe the components of these compiling systems. For more details about using the compilers, see the manuals provided with each compiling system.

#### Fortran 64-Bit and N32 System

The 64-bit Fortran compilers have the following components:

| | |
|---|---|
| *Fortran driver* | Fortran driver or command line: Executes the components below. |
| *front end (fe)* | Parses the source file into an intermediate representation. It also performs scalar optimization and automatic parallelization. |
| *back end (be)* | Generates code and assembles it into an object file. It also performs a variety of optimizations. It also automatically performs scalar optimizations and inter procedural optimizations. When used with the MIPSpro Auto-Parallelizing Option product it automatically converts programs to parallel code. |
| *dsm_prelink* | Prelinker for routines that use distributed shared memory. If a reshaped array is passed as a parameter to another subroutine, *dsm_prelink* automatically propagates the *distribute_reshape* directive to the called subroutine. |
| *linker* | Links the object file(s) with any libraries. |

When you run 64-bit compilations for single processor applications, the following components are executed by the compiler driver:

```
%f77 -64 foo.f
%f77 -64 -O foo.f
       fe --> be --> linker
```

When you run 64-bit compilations for multiprocessor applications an additional step invoking *dsm_prelink* is done just before the final linking step:

```
%f77 -64 -mp foo.f
%f77 -64 -pfa foo.f
       fe --> be --> dsm_prelink --> linker
```

With the MIPSpro 64-bit compiler, optimizations are performed in the back end. Note that **-O3** is available with **-c**. Unlike the older *ucode* compilers, **-O3** does not result in interprocedural optimizations being performed. Use the **-IPA:...** control group to perform interprocedural optimizations with the 64-bit compiler. See the ipa(5) man page for more details.

The **-sopt** switch is NOT supported on the 64-bit compiler. Use the **-LNO: ...** control group flags to perform the desired scalar optimizations. See the lno(5) man page for details.

The **-mp** switch is supported on the 64-bit compiler and causes the front end to recognize inserted parallelization directives.

### Fortran 32-Bit System

The 32-bit (*ucode)* Fortran compiler systems contain the following components:

| | |
|---|---|
| *Fortran driver* | Fortran driver or command line: Executes the components below. |
| *cpp* | C preprocessor: Handles **#include** statements and other *cpp* constructs such as **#define**, **#ifdef**, and so on, in the source file. |
| *fopt* | Special scalar optimizer: Performs scalar optimization on the Fortran source. |
| *fcom* | Fortran front end: Parses the source file into intermediate code (*ucode*). |
| *uopt* | Optimizer: Performs optimizations on the intermediate file. |

| | |
|---|---|
| *ugen* | Code generator: Generates binary assembly code from the intermediate file. |
| *as1* | Binary assembler: Assembles the *binasm* code into an object file. |
| *linker* | Linker: Links the object file(s) with any libraries. |

When you run simple examples through the *ucode* Fortran compilers, the following components are executed by the compiler driver:

```
%f77 -32 foo.f
    cpp --> fcom --> ugen --> as1 --> linker
```

The command

```
%f77 -32 -O foo.f
    cpp --> fcom --> uopt --> ugen --> as1 --> linker
```

also invokes the *ucode* optimizer, *uopt*. The command

```
%f77 -32 -sopt foo.f
    cpp --> fopt --> fcom --> ugen --> as1 --> linker
```

invokes the scalar optimizer but does not invoke the *ucode* optimizer.

The **-mp** option signals *fcom* to recognize inserted parallelization directives:

```
%f77 -32 -mp foo.f
    cpp --> fcom --> ugen --> as1 --> linker
```

## C

For C, the respective compiler systems are similar to their Fortran counterparts. The front ends, of course, are different in each system.

### C 64-Bit and N32 System

The MIPSpro (64-bit) C compiler systems contain the following components:

| | |
|---|---|
| *cc* | C driver: Executes the appropriate components below. |
| *fec* | C front end: Preprocesses the C file, and then parses the source file into an intermediate representation. |

| | |
|---|---|
| *be* | Back end: Generates code and assembles it into an object file. It also performs a variety of optimizations which are described in Chapter Four of this book, *Compilation Issues*. It also automatically performs scalar optimizations and inter procedural optimizations. Available with the MIPSpro AutoParallelizing Option product, is the ability to automatically convert programs to parallel code. |
| *dsm_prelink* | Prelinker for routines that use distributed shared memory. If a reshaped array is passed as a parameter to another subroutine, *dsm_prelink* automatically propagates the *distribute_reshape* directive to the called subroutine. |
| *linker* | Links the object file(s) with any libraries. |

When you run simple examples through the 64-bit C compilers, the following components are executed by the compiler driver:

```
%cc -64 foo.c
%cc -64 -O foo.c
        fec --> be --> linker
```

When you run 64-bit compilations for multiprocessor applications an additional step invoking *dsm_prelink* is done just before the final linking step:

```
%cc -64 -mp foo.c
%cc -64 -pca foo.c
        fec --> be --> dsm_prelink --> linker
```

### C 32-Bit System

The 32-bit (*ucode*) C compiler systems contain the following components:

| | |
|---|---|
| *cc* | C driver: Executes the appropriate components below. |
| *acpp* | ANSI C preprocessor: Handles **#include** statements and other *cpp* constructs such as **#define**, **#ifdef**, and so on, in the source file. |
| *cfe* | C front end: Preprocesses the C file, and then parses the source file into intermediate code (*ucode*). |
| *mpc* | Interprets parallel directives. |
| *copt* | C scalar optimizer: Performs scalar optimization. |

| | |
|---|---|
| *uopt* | Optimizer: Performs optimizations on the intermediate file. |
| *ugen* | Code Generator: Generates binary assembly code from the intermediate file. |
| *as1* | Binary assembler: Assembles the *binasm* code into an object file. |
| *linker* | Linker: Links the object file(s) with any libraries. |

When you run simple examples through the *ucode* C compiler, the following components are executed by the compiler driver:

```
%cc -32 foo.c
    cfe --> ugen --> as1 --> linker
```

**Note:** *cfe* has a built-in C preprocessor.

The command

```
%cc -32 -O foo.c
    cfe --> uopt --> ugen --> as1 --> linker
```

also invokes the *ucode* optimizer, *uopt*.

The command

```
%cc -32 -sopt foo.c
    acpp --> copt --> cfe --> ugen --> as1 --> linker
```

invokes the scalar optimizer but does not invoke the *ucode* optimizer.

The C preprocessor has to be run before *copt* can do its source-to-source translation:

```
%cc -32 -mp foo.c
    acpp --> mpc --> cfe --> ugen --> as1 --> linker
```

-**mp** signals *mpc* to recognize inserted parallelization directives.

## Interprocedural Analysis (IPA)

As of version 7.0, the MIPSpro 64-bit or N32 compilers can perform interprocedural analysis and optimization when invoked with the **-IPA** command line option. Current IPA optimizations include: inlining, interprocedural constant propagation, dead function, dead call and dead variable elimination and others. For more information about IPA and its optimization options, see the ipa(5) man page.

An important difference between the 64-bit compiler's use of **-IPA** and **-c** and the 32-bit compilers use of **-O3** and **-j** is that the intermediate files generated by the 64-bit compiler have the *.o* suffix. This can greatly simplify *Makefiles*. For example:

```
% cc -n32 -O -IPA -c main.c
% cc -n32 -O -IPA -c foo.c
% ls
foo.c   foo.o   main.c  main.o
% cc -n32 -IPA main.o foo.o
```

An analogous 32-bit compilation would look like:

```
% cc -32 -O3 -j main.c
% cc -32 -O3 -j foo.c
% ls
foo.c   foo.u   main.c  main.u
% cc -32 -O3 main.u foo.u
```

**Note:** Use of the non-standard **-j** option and non-standard *.u* (*ucode*) files leads to more complicated *Makefiles*.

## Loop Nest Optimizer (LNO)

The loop nest optimizer performs high-level optimizations that can greatly improve program performance by exploiting instruction level parallelism and caches. LNO is run by default at the **-O3** optimization level. LNO is integrated into the compiler back end (*be*) and is not a source-to-source preprocessor. As a result, LNO optimizes C++, C and Fortran programs, although C and C++ often include features that make them inherently more difficult to optimize. For more information about LNO and its optimization options, refer to the lno(5) man page.

In order to view the transformations that LNO performs, you can use the **-CLIST:=ON** or **-FLIST:=ON** options to generate C or Fortran listing files respectively. The listing files are generated with the *.w2.f* (or *.w2.c*) suffix. For example:

```
%cat bar.f

subroutine bar(a,b,c,d,j)
real*4 a(1024),b(1024),c(1024)
real*4 d,e,f

sum = 0
do m= 1,j
do i=1,1024
b(i) = b(i) * d
enddo
enddo

call foo(a,b,c)
end


%f77 -64 -O3 -FLIST:=ON foo.f
%cat foo.w2.f
C ************************************************************
C Fortran file translated from WHIRL Fri May 17 12:07:56 1997
C ************************************************************


        SUBROUTINE bar(a, b, c, d, j)
        IMPLICIT NONE
        REAL*4 a(1024_8)
        REAL*4 b(1024_8)
        REAL*4 c(1024_8)
        REAL*4 d
        INTEGER*4 j
C
C**** Variables and functions ****
C
        INTEGER*4 m
        INTEGER*4 i
        EXTERNAL foo
C
C**** Temporary variables ****
C
        INTEGER*4 wd_m
        INTEGER*4 i0
C
```

```
C**** statements ****
C

        DO m = 1, j + -1, 2
          DO i = 1, 1024, 1
            b(i) = (b(i) * d)
            b(i) = (b(i) * d)
          END DO
        END DO
        DO wd_m = m, j, 1
          DO i0 = 1, 1024, 1
            b(i0) = (b(i0) * d)
          END DO
        END DO
        CALL foo(a, b, c)
        RETURN
        END ! bar
```

## MIPSpro Auto-Parallelizing Option

The MIPSpro Auto-Parallelizing Option analyzes data dependence to guide automatic parallelization. For the 7.2 compiler release this functionality is implemented in the 64-bit and N32 compiler back end (*be*). It replaces KAP (Kuck and Associates Preprocessor) which was implemented as a separate preprocessor. An advantage to being built into the backend is that automatic parallelization is now available for C++ as well the previously supported C, Fortran 77 and Fortran 90. Another advantage to this design, is that a separate (and orthogonal) set of optimization options is no longer necessary.

### Compiling with Automatic Parallelization

To compile with automatic parallelization you must obtain the *MIPSpro Auto-Parallelizing Option* and install its license. The syntax for compiling programs with automatic parallelization is as follows:

For Fortran 77, C, and Fortran 90 compilations use **-apo** on your compilation command line. For example:

```
%f77 -apo foo.f
```

If you link separately, you must also add **-mp** to the link line. See the apo(5) man page for details.

## Automatic Parallelization Listings

The auto-parallelizer provides a listing mechanism when you use the **-apo list** option. This causes the compiler to generate a *.l* file. The *.l* file lists the original loops in the program along with messages telling if the loops were parallelized. For loops that were not parallelized, an explanation is given. For example:

```
%cat test.f
       program test
       real*8 a, x(100000),y(100000)
       do i = 1,2000
         y(i) =  y(i-1) + x(i)
       enddo
       do i = 1,2000
         call daxpy(3.7,x,y,100000)
        enddo
        stop
        end


        subroutine daxpy( a, x, y, nn)
        real*8 a, x(*), y(*)
        do i = 1, nn,1
          y(i) =  y(i) + a*x(i)
        end do
        return
        end
%f77 -64 -mp list test.f
%cat test.l
Parallelization Log for Subprogram MAIN__
    3: Not Parallel
        Array dependence from y on line 4 to y on line 4.
    6: Not Parallel
        Call daxpy on line 7.
Parallelization Log for Subprogram daxpy_
    14: PARALLEL (Auto) __mpdo_daxpy_1
```

The **-mplist** option will, in addition to compiling your program, generate a *.w2f.f* file (for Fortran, *.w2c.c* file for C) that represents the program after the automatic parallelization phase. These programs should be readable and in most cases should be valid code suitable for recompilation. The **-mplist** option can be used to see what portions of your code were parallelized. Continuing our example from above:

```
%f77 -64 -apo -mplist test.f
%cat test.w2f.f
C ************************************************************
C Fortran file translated from WHIRL Sat Jul 26 12:05:52 1997
C ************************************************************


        PROGRAM MAIN
        IMPLICIT NONE
C
C          **** Variables and functions ****
C
        REAL*8 x(100000_8)
        REAL*8 y(100000_8)
        INTEGER*4 i

C
C       **** statements ****
C
        DO i = 1, 2000, 1
          y(i) = (x(i) + y(i + -1))
        END DO
        DO i = 1, 2000, 1
          CALL daxpy(3.7000000477, x, y, 100000)
        END DO
        STOP
        END ! MAIN


        SUBROUTINE daxpy(a, x, y, nn)
        IMPLICIT NONE
        REAL*8 a
        REAL*8 x(*)
        REAL*8 y(*)
        INTEGER*4 nn
C
C       **** Variables and functions ****
C
        INTEGER*4 i
        INTEGER*4 __mp_sug_numthreads_func$
        EXTERNAL  __mp_sug_numthreads_func$
C
C       **** statements ****
C
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_daxpy_1
```

```
C$OMP PARALLEL DO if(((DBLE(__mp_sug_numthreads_func$()) *((DBLE(
C$&(__mp_sug_numthreads_func$())*1.23D+02)+2.6D+03)).LT. (DBLE((
C$&(__mp_sug_numthreads_func$()+ -1))*(DBLE(nn)*7.0D00)))), private
C$&(i), shared(y, x, a, nn)

        DO i = 1, nn, 1
          y(i) = (y(i) +(x(i) * a))
        END DO
        RETURN
        END ! daxpy
```

The **-pfa keep** option generates a *.l* file, a *.anl* file that used by the *Workshop ProMPF* tool, and a *.m* file. The *.m* file is similar to the *.w2f.f* or *.w2c.c* file except that the file is annotated with some information used by *Workshop ProMPF*.

For Fortran 90 and C++, automatic parallelization happens after the source program has been converted into an internal representation. It is not possible to regenerate Fortran 90 or C++ after parallelization.

## Multiprocessing Support

IRIX 6.x and the MIPSpro compilers support multiprocessing primitives for 32-bit, N32 and 64-bit applications. The 64-bit (and N32) multiprocessor programming environment is a superset of the 32-bit one. It also contains enhancements.

### MP Compatibility

This section describes 64-bit and 32-bit Fortran MP compiler compatibility:

- The 64-bit Fortran compiler supports all of the parallelization directives (such as *C$DOACROSS, C$&, C$MP_SCHEDTYPE, C$CHUNK, C$COPYIN*) supported by the 32-bit Fortran compiler. In addition, the Fortran front end supports PCF style parallel directives, which are documented in the *MIPSpro Fortran 77 Programmer's Guide*.
- The 64-bit Fortran compiler supports the same set of multiprocessing utility subroutine calls (such as **mp_block** and **mp_unblock**) as the 32-bit compiler.

  The 64-bit Fortran compiler supports the same set of environment variables (such as *MP_SET_NUMTHREADS* and *MP_BLOCKTIME*) as the 32-bit compiler.

- The **-mp** option is supported on both the 32-bit compilers and the 64-bit compilers.
  - **-mp** allows LNO to recognize manually-inserted parallelization directives in the 64-bit compiler.
  - **-apo** enables automatic parallelization by the MIPSpro Auto-Parallelizing Option (64-bit and N32).

## MP Enhancements

The MIPSpro 64-bit Fortran MP I/O library has been enhanced to allow I/O from parallel regions. In other words, multiple threads can read and write to different files as well as read and write to the same file. The latter case, of course, encounters normal overhead due to file locking.

The MIPSpro 64-bit compilers also have been enhanced to allow parallel C and parallel Fortran programs to share a common runtime. This allows you to link parallel C routines with parallel Fortran routines and have a single master. Figure 1-2 illustrates this.

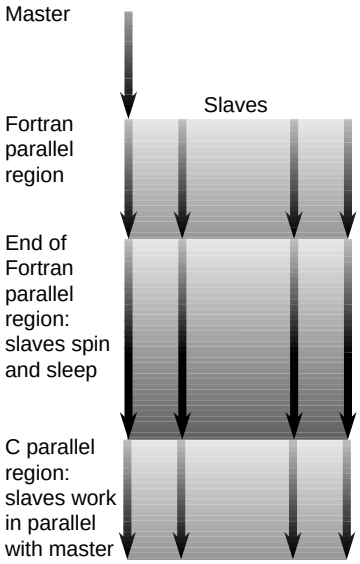


**Figure 1-2** Running Parallel C and Parallel Fortran Programs Together

The MIPSpro 64-bit compilers are also enhanced to provide a variety of primitive synchronization operations. The operations are guaranteed to be atomic (typically achieved by implementing the operation using a sequence of load-linked/store-conditional instructions in a loop).

Associated with each operation are certain memory barrier properties that restrict the movement of memory references to *visible* data across the intrinsic operation (by either the compiler or the processor). For more information, see the *MIPSpro Fortran77 Programmer's Guide* and the sync(3i) and the sync(3c) man pages.

## New Directives for Tuning on Origin2000

The Origin2000 provides cache-coherent, shared memory in the hardware. Memory is physically distributed across processors. Consequently, references to locations in the remote memory of another processor take substantially longer (by a factor of two or more) to complete than references to locations in local memory. This can severely affect the performance of programs that suffer from a large number of cache misses.

The new programming support consists of extensions to the existing multiprocessing Fortran and C directives (pragmas) as well as support for C++. Also provided are intrinsic functions that can be used to manage and query the distribution of shared memory. For more information, see the *MIPSpro Fortran77 Programmer's Guide,* and *MIPSPro C and C++ Pragmas.*

## OpenMP Support

Starting with the MIPSpro 7.2.1 release, the Fortran77 and Fortran90 64-bit and N32 compilers support the OpenMP application programming interface (API) when used in conjunction with the **-mp** flag.The **-mp** flag enables the processing of the original SGI/PCF directives as well as the OpenMP directives. To selectively disable one or the other set of directives, add the following **-MP** option group flag to the **-mp** flag:

**-MP:old_mp=off**

disable processing of the original SGI/PCF directives, but retain the processing of OpenMP directives.

**-MP:open_mp=off**

disable processing of the OpenMP directives, but retain processing of the original SGI/PCF directives.

To run OpenMP programs you must install the appropriate version of *libmp.so*. Please refer to your *IRIX Development Foundation Release Notes* for more information about this. For more information about the OpenMP directives, see the *MIPSPro Fortran 77 Programmer's Guide* or the *MIPSpro Fortran 90 Commands and Directives Reference Manual*.

## MP Application Testing

In general, to test 64-bit MP applications, follow these guidelines:

- First, get the application to run with no parallelization at the highest optimization level.
- When testing the parallel version, first run it with only one thread (either on a single CPU machine or by setting the environment variable *MP_SET_NUMTHREADS* to 1).
- Go down to the **-g** optimization level for the first MP test, and run that version with one thread, then with multiple threads. Then go up the optimization scale, testing both single and multi-threaded versions.

You can, of course, skip as many steps as you like. In case of failure, however, this method of incremental iterations can help you narrow down and identify the problem.

*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).
  - Right-justifying small scalar parameters in their save area slots notwithstanding, 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:** If you add **–TENV:varargs_prototypes=off** to the compilation command line, the floating point parameters appear in both floating point registers and integer registers. This decreases the performance of not only **varargs** routines with floating point parameters, but also of any unprototyped routines that pass floating point parameters. The program compiles and executes correctly; however, a warning message about unprototyped **varargs** routines still is present.

  - 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 |

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

| Register Name | Software Name | Use | Saver |
|---|---|---|---|
| $f1, $f3 | | Floating point temporaries | Caller-saved |
| $f4..$f11 | | Floating point temporaries | Caller-saved |
| $f12..$f19 | | Floating point arguments | Caller-saved |
| $f20..$f23 | | Floating point temporaries | Caller-saved |
| $f24..$f31 | | Floating point | Callee-saved |

Table 2-2 gives 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 |

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

| Argument List | Register and Stack Assignments |
|---|---|
| d1,s1,s2 | $f12, $f13, $f14 |
| s1,s2,d1 | $f12, $f13, $f14 |
| n1,n2,n3,n4 | $4,$5,$6,$7 |
| 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,s4 | $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 *mfef*.

### Fortran Features

*mfef* 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 *mfef* are incompatible in the way they fold **REAL*4** constants. *fcom* promotes them internally to **REAL*8**. *mfef* however, provides the **-r8const** flag to simulate the *fcom* behavior.
- *mfef* allows more constant expressions in parameter statements than *fcom*.
- *mfef* allows parameters (which are ignored with a warning message) to the program statement.
- *mfef* 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 |
| float | 32 | 32 |

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

| C type | 32-bit | 64-bit |
|---|---|---|
| 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.

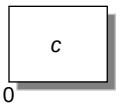


**Figure 2-1** Structure Smaller Than a Word

**Example 2: Structure With No Padding**

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

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

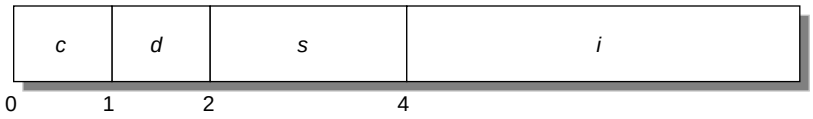


**Figure 2-2** Structure With No Padding

**Example 3: Structure With Internal Padding**

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

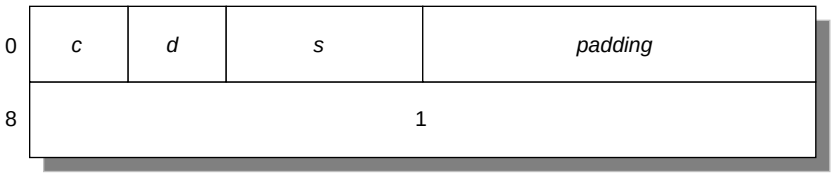


**Figure 2-3** Structure With Internal Padding

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

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

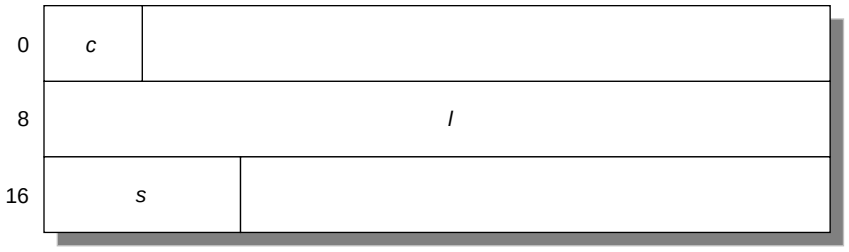


**Figure 2-4** Structure With Internal and Tail Padding

**Example 5: Union Allocation**

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

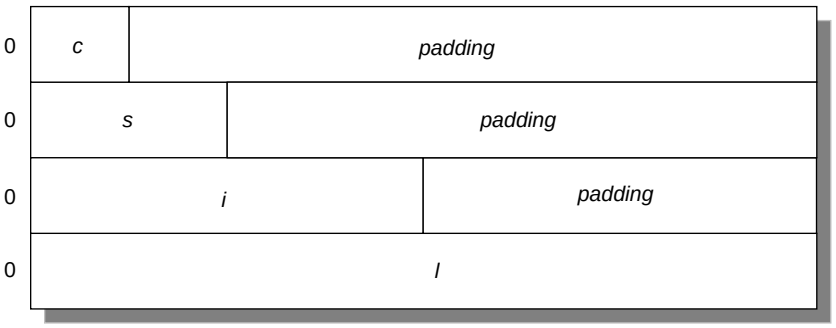


**Figure 2-5** Union Allocation

## 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 are covered in depth in Chapter 3, “Source Code Porting.”

*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*8 iptr, asize

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 numptrs
integer*8 tableptr, asize

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 MIPS2executables:**

```
-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.

*Chapter 4*

# Compilation Issues

This chapter outlines the issues dealt with at compile and link times. It covers environment variable usage, compile-time switches, error messages and how to link correctly.

## Environment Variables

The *SGI_ABI* environment variable can be used to specify whether you want an old 32-bit, high performance 32-bit (N32) or 64-bit compilation environment. It is overridden by using the **-32** (or **-o32**), **-64** or **-n32** option on the command line.

Because there are three distinct ABIs used by programs with accompanying libraries, there are three distinct sets of *rld* search paths and three sets of environment variables.

The default library search path for o32 (old 32-bit ABI) programs is (*/usr/lib:/usr/lib/internal:/lib*), which can be overridden by either of the environment variables *_RLD_ROOT* or *LD_LIBRARY_PATH*.

The default library search path for 64 (64-bit ABI) programs is (*/usr/lib64:/usr/lib64/internal:/lib64*), which can be overridden by either of the environment variables *_RLD64_ROOT* or *LD_LIBRARY64_PATH*.

The default library search path for n32 (new 32-bit ABI) programs is (*/usr/lib32:/usr/lib32/internal:/lib32*), which can be overridden by either of the environment variables *_RLDN32_ROOT* or *LD_LIBRARYN32_PATH*.

For n32 or 64-bit programs, if *LD_LIBRARYN32_PATH* or *LD_LIBRARY64_PATH* is not specified, then *rld* will honor *LD_LIBRARY_PATH* if specified.

## Command Line Switches

This section lists the differences between the supported switches of the 32-bit and 64-bit compilers.

If you use the **-32** switch on the compiler command line, the 32-bit (*ucode)* compiler is invoked. If you use **-64** or **-n32**, the 64-bit compiler is run.

The default switches on all R8000 based platforms are **-64 -mips4**. On all other platforms the default switches are **-32 -mips2** unless the */etc/compiler.defaults* file is present. If it is, its contents will specify what the default values are.

Although they are two separate and different compiler systems, the 32-bit and 64-bit compilers have similar command line interfaces. The following list summarizes the differences between the compilers in the switches that they support. A full list can be found in the cc(1), f77(1) and f90(1) man pages:

### Fortran Switch Differences

The 32-bit compiler supports the following switches which the 64-bit compiler does not:

**-32, -o32** By definition.

**-mips1** Generate code using the MIPS1 instruction set.

**-mips2** Generate code using the MIPS2 instruction set (the default for the 32-bit compiler).

**-66** Suppress extensions that enhance FORTRAN 66 compatibility.

**-[no]kpicopt** Asks *uopt* to perform the special treatment for global variables to optimize their program accesses regardless of the shared or non-shared compilation mode.

**-j** Compile the specified source program and leave the *ucode* object file in a *.u* file.

**-w66** Suppress only FORTRAN 66 compatibility warnings messages.

**-usefpidx** Force the compiler to use the floating point **DO** loop variable as the loop counter.

**-vms_cc** Use VMS Fortran carriage control interpretation on unit 6.

**-vms_endfile** Write a VMS endfile record to the output file when **ENDFILE** statement is executed and allow subsequent reading from an input file after an endfile record is encountered.

**-vms_library** Treat subroutines/functions starting with **LIB$**, **OTS$**, and **SMG$** as VMS runtime library routines which take a variable number of arguments.

**-vms_stdin** Allow rereading from *stdin* after **EOF** has been encountered.

The 64-bit compiler supports the following switches which the 32-bit compiler does not:

**-64** By definition.

**-n32** Generate code for the high performance 32-bit (N32) ABI.

**-mips3** Generate code using the MIPS3 instruction set.

**-mips4** Generate code using the MIPS4 instruction set (the default for the 64-bit compiler on Power Challenge).

## C Switch Differences

The 32-bit compiler supports the following switches which the 64-bit compiler does not:

**-32, -o32, -mips1,-mips2**
As in 32-bit Fortran above.

**-[no]kpicopt** Asks *uopt* to perform the special treatment for global variables to optimize their program accesses regardless of the shared or non-shared compilation mode.

**-j** As in 32-bit Fortran above.

**-hpath** Use *path* rather than the directory where the name is normally found.

**-Bstring** Append *string* to all names specified by the **-t** option.

The 64-bit compiler supports the following switches which the 32-bit compiler does not:

**-64, -n32,-mips3 -mips4**
As in 64-bit Fortran above.

**-help** Print a list of possible options.

**-keep** Keep intermediate files.

**-show1** Show phases but don't have phases print additional info.

**-showt** Show time taken by each phase.

**-unsigned** Characters are unsigned by default.

**-woffall** Turn off all warnings.

The following switches are accepted by both compilers but have different semantics:

**-O3** On the 32-bit compiler, a *ucode* object file is created and left in a *.u* file. Inter-procedural optimization is done on the resulting *ucode* linked file and an *a.out* file is created. On the 64-bit compiler interprocedural optimization is carried out when the **-IPA** flag is present and can be done at optimization levels less than **-O3**.

**-v** On the 32-bit compiler, it's the same as **-show**, whereas it means verbose warnings on the 64-bit compiler.

**-woff** Turn off named warnings, but the warning numbers are different between the compilers.

**-Wc,arg1**,[,**arg2**...]
Designate to which pass, **c**, of the compiler the arguments are to be passed.

## Optimization Switches of the 64-Bit Compilers

In addition to the switches listed above, both the 64-bit Fortran and 64-bit C compilers support many more switches. These switches are used to control the types of optimizations the compilers perform. This section outlines the various optimizations that the 64-bit compilers can perform, and lists the switches used to control them. To get a summary of the flags and their values set the **-LIST:all_options=ON** flags on the command line. For example:

```
%f77 -64 -O3 -mips4 -LIST:all_options=ON foo.f
```

This creates a *.l* file which contains a summary of all the flags used in the compilation including the default values of flags not explicitly set on the command line.

The optimizations and switches fall into the following categories:

- General optimizations which are described on the individual compiler man pages (f77(1), f90(1) and cc(1)). These include options to perform local and global optimizations, aggressive optimizations, and options that maximize performance.
- Floating point optimizations, which include a number of options which trade off source language expression evaluation rules and IEEE 754 conformance against better performance of generated code. These options allow transformations of calculations specified by the source code that may not produce precisely the same floating point result, although they involve a mathematically equivalent calculation. See the opt(5) man page for details about these options.
- Processor-specific tuning options, which are used to instruct the compiler to schedule code for the given processor and to use processor-specific math libraries (e.g. */usr/lib64/mips4/r10000/libm.so*) when linking. The resulting code may be optimized for a particular platform, but may not be optimal for use across all platforms. These options are described on the individual compiler man pages (f77(1), f90(1) and cc(1)).
- Inter-Procedural Analysis (IPA) optimizations, which control subroutine inlining. See the ipa(5) man page for details.
- Loop Nest Optimizations (LNO), which allow the user to specify that command-line options override pragmas in a file. See the lno(5) man page for details.
- Target environment options, which tell the compiler about the target software environment. See the individual compiler man pages (f77(1), f90(1) and cc(1)) for details about the **-TENV** options.

## Data Alignment Options

Two compiler options are concerned with the alignment of data:

#### -align32 | -align64

The MIPS architectures perform memory references much more efficiently if the data referenced is naturally aligned, that is, if 4-byte objects are at 4-byte-aligned address, etc. By default, the compilers allocate well-aligned data, and that is a requirement of the ABI for C. However, code ported from other architectures without alignment constraints may require less restricted alignment.The ANSI Fortran standard essentially requires maximum alignment of 4 bytes (32 bits), although it is unusual for code to actually

depend on this. These options specify a maximum alignment (in bits) to be forced in allocating data. The MIPSpro 64-bit compilers default to **-align64** for MIPS3 or MIPS4 Fortran, and to ABI alignment (up to 128 bits for long double) for C.

#### -TENV:align_aggregates=n

The ABI specifies that aggregates (that is, **structs** and **arrays**) be aligned according to the strictest requirements of their components (that is, fields or elements). Thus, an array of **short ints** (or a **struct** containing only **short ints** or **chars**) is normally 2-byte aligned. However, some non-ANSI-conforming code may reference such objects assuming greater alignment, and some code (for example, **struct** assignments) may be more efficient if the objects are better aligned). This option specifies that any aggregate of size at least *n* is at least n-byte aligned. It does not affect the alignment of aggregates which are themselves components of larger aggregates.

## Compilation Messages

Because they are two separate and different compiler systems, the 32-bit and 64-bit MIPSpro compilers have different warning and error messages.

The 32-bit C compiler emits errors as follows: the phase of the compiler which detects an error or warning, identifies itself first, then it prints whether it found an error or warning. The next printed items are the file name, line number and description of the error.

Fortran error messages are similar except the compiler phase name is omitted.

Two examples illustrate this:

C:

```
%cc -32 test.c
cfe: Error: test.c, line 4: Syntax Error
```

Fortran:

```
% f77 -32 err.f
Error on line 6 of err.f: syntax error
```

Under the 64-bit compiler, back end warning messages start with the string “!!!”. Error messages start with the string “###”. This allows easier searches in log files. C error messages are similar to the 32-bit compiler’s error messages, although they are usually more descriptive.

Here the same files compiled with the 64-bit compiler result in the following error messages:

C:

```
%cat test.c
#include <stdio.h>
main ()
{
    printf (“help”)
}
%cc -64 test.c
“test.c”, line 4: error(1065): expected a “;”
```

Fortran:

```
%cat err.f
        program test
c
c
2       write(6,8)
8       formatr('This is a test')
        end

% f77 -64 err.f
“err.f”, line 6: error(2080): expected a “(“
8       formatr('This is a test')
                    ^
1 error detected in the compilation of “err.f”
Important Warning Messages
```

When porting a 32-bit application to 64-bits use the **-fullwarn** option to expose variable size related issues. Two warnings in particular emitted by the 64-bit C compiler are helpful in this respect. They allow you to locate faulty assumptions that integers and pointers are the same size.

**cc-3187 cc: WARNING File = foo2.c, Line =4**

**A pointer is converted to a smaller integer.**

For example, the following code fragment generates this warning when compiled **-64**.

```
i = (unsigned) p;
```

In this example, the unsigned integer variable *i* is set to only the low order 32 bits of *p*. If *p* contains an address greater than 32 bits, that address is truncated in *i*.

## Linking Your Application

The *cc* driver automatically picks the correct compiler, linker, and library paths if you use the **-32, -n32** or **-64** switches on the command line. If you compile and link separately, you must use these switches on both the *cc* driver and *ld* command lines. If you link with libraries that are not included with IRIX or the compiler, you must make sure that they are of the proper type. Remember, you can't link 32-bit applications with N32 or 64-bit *.so's*.

To create your own N32, 64-bit or 32-bit libraries, you must supply the correct switch on the archiver command line.

### Libraries

The 32-bit and 64-bit compilers include some different libraries. Whereas, the 32-bit compiler includes *libftn.so, libF77.a, libI77.a, libU77.a* and *libisam.a* under */usr/lib*, the 64-bit compiler has one library, *libftn.so*, under */usr/lib64* and */usr/lib32.*

The 64-bit compiler also introduces routines which do QUAD precision (128-bit) floating point calculations into existing libraries.

The 64-bit compiler provides a united runtime library for parallel C and parallel Fortran programs (*/usr/lib64/libmp.so*), as opposed to two (*/usr/lib/libc_mp.a, /usr/lib/libkapio.a*) libraries under the 32 code compiler. This united library allows you to mix parallel C with parallel fortran programs.

The 64-bit compiler also does not include *libmld.a*, but provides *libdwarf.a*.

*Chapter 5*

# Runtime Issues

This chapter outlines why your 32-bit and 64-bit applications may run differently, due both to compiler differences and to architectural modes. It describes the Performance and Precise Exception Modes of the R8000 microprocessor architecture and how they affect the calculations of applications. This chapter also briefly outlines a methodology to bring up and debug applications.

## Runtime Differences

Your 64-bit application may produce slightly different floating point calculations on the R8000 than on its 32-bit counterpart. There can be a variety of causes for this. These include reassociation of operations by optimizations, algorithm changes in libraries and hardware changes.

### Reassociation of Operations by Optimizations

The order in which equivalent floating point expressions are executed can cause differences in their results. The 32-bit and 64-bit compiler systems perform different optimizations which can cause reordering of instructions leading to slightly different results. The compilers may also perform operation reductions which can affect the results.

### Algorithm Changes in Libraries

The 64-bit compiler comes with new math libraries which use different algorithms than those used with the 32-bit compiler to improve their performance. The values which they return can cause potentially noticeable differences in application results.

## Hardware Changes

The R8000 microprocessor includes four floating point multiply/add / subtract instructions which allow two floating point computations to be performed with one instruction. The intermediate result is calculated to infinite precision and is not rounded prior to the addition or subtraction. The result is then rounded according to the rounding mode specified by the instruction. This can yield slightly different calculations than a multiply instruction (which is rounded) and an add instruction (which is rounded again).

The R8000 microprocessor architecture also defines two execution environments which can affect your application if it generates floating point exceptions such as underflow. Performance Mode enhances the execution speed of floating point applications, by rounding denormalized numbers to zero and allowing the hardware to trap exceptions imprecisely. Precise Exception Mode, on the other hand, is fully compatible to the existing MIPS floating point architecture.

It should be emphasized that running in Performance Mode does not affect those applications which don't cause floating point exceptions.

A program, *fpmode*, allows you to run your application in either Performance (imprecise) or Precise Mode. Its usage is as follows:

```
%fpmode precise commandargs
```

or

```
%fpmode imprecise commandargs
```

A full discussion of the Extended MIPS Floating Point Architecture is provided as a reference.

## Extended MIPS Floating-Point Architecture

The MIPS architecture fully complies with the ANSI/IEEE Standard 754-1985, *IEEE Standard for Binary Floating-Point Arithmetic*. Most application programs utilize only a fraction of all the features required by the Standard. These applications can gain additional performance if executed in an environment that supports only those features of the Standard that are actually necessary for the correct execution of the application. The Extended MIPS Floating-Point Architecture defines two execution environments:

- Performance Mode enhances the execution speed of most applications by rounding denormalized numbers to zero and by allowing the hardware to trap exceptions imprecisely. This mode requires compiler and library support to fully comply with the Standard.

  In Performance Mode, the hardware and operating system are relieved of the requirements to precisely trap floating-point exceptions and to compute using denormalized operands. This mode is defined in such a way that it is adequate for a majority of application programs in use today, yet it can also be used in conjunction with compiler and library support to fully implement the Standard in the future.

  Performance Mode improves the floating-point execution speed of processors. On the R4000, Performance Mode enables flushing operands to zero, thus avoiding the software emulation overhead of denormalized computations. On the R8000, Performance Mode enables floating-point instructions to execute out-of-order with respect to integer instructions, improving performance by a factor of two or more.

  Performance Mode is the standard execution environment on R8000 based Power Challenge systems.

- Precise Exception Mode fully complies with the Standard and is compatible in every way to the preexisting MIPS floating-point architecture.

  In Precise Exception Mode the responsibility for compliance lies entirely with the hardware and operating system software; no compiler support is assumed. Since there is no information about the application, the hardware must assume the most restrictive features of the Standard applies at all times. The result is lost performance opportunities on applications that utilize only a subset of the features called for by the Standard.

## Performance Mode

The purpose of this section is to define Performance Mode and explain why it is necessary and desirable.

## Background

The IEEE Standard defines floating-point numbers to include both normalized and denormalized numbers. A denormalized number is a floating-point number with a minimum exponent and a nonzero mantissa which has a leading bit of zero. The vast majority of representable numbers in both single and double precision are normalized numbers. An additional small set of very tiny numbers (less than $2^{-126}$ ($\sim 10^{-38}$) in single precision, less than $2^{-1022}$ ($10^{-308}$) in double precision are represented by denormalized numbers. The importance of approximating tiny real values by denormalized numbers, as opposed to rounding them to zero, is controversial. It makes no perceptible difference to many applications, but some algorithms need them to guarantee correctness.

Figure 5-1 shows pictorially the IEEE definition of floating-point numbers. Only the positive side of the real number line is shown, but there is a corresponding negative side also. The tick marks under the real number line denote example values that can be precisely represented by a single or double precision binary number. The smallest representable value larger than zero is *minD*, a denormalized number. The smallest normalized number is *minN*. The region between zero and just less than *minN* contains tiny values. Larger values starting with *minN* are not tiny.

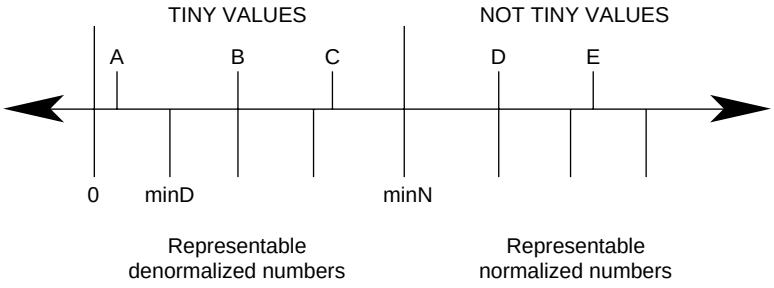


**Figure 5-1** Floating Point Numbers

The different cases that must be considered are represented by the values A-E. According to the IEEE Standard, the behavior of an operation that produces these result values is defined as shown in Table 5-1.

**Table 5-1** Operation Results According to IEEE Standard

| Value | Result | Flags |
|---|---|---|
| A:TooSmall | rnd(A) | U=1,I=1 |
| B:ExactDenorm | B | U=1, I=0 if Enable U=U=0, I=0 if EnableU=0 |
| C:InexactDenorm | rnd(C) | U=1,I=1 |
| D:ExactNorm | D | U=0,I=0 |
| E:InexactNorm | rnd(E) | U=0, I=1 |

The flags *U* and *I* abbreviate Underflow and Inexact, respectively. The function *rnd()* rounds the operand to the nearest representable floating point number based on the current rounding mode, which can be round-to-zero, round-to-nearest, round-to-plus-infinity, and round-to-minus-infinity. For example, *rnd(A)* is either zero or *minD*. A trap occurs if a flag is set and the corresponding enable is on. For example, if an operation sets *I*=1 and *EnableI*=1, then a trap should occur. Note that there is a special case for representable tiny values: the setting of the *U* flag depends on the setting of its enable.

Supporting denormalized numbers in hardware is undesirable because many high performance hardware algorithms are designed to work only with normalized numbers, and so a special case using additional hardware and usually additional execution time is needed to handle denormalized numbers. This special case hardware increases the complexity of the floating-point unit and slows down the main data path for normalized numbers, but is only rarely used by a few applications. Therefore most processor designers have generally deemed it not cost effective to support computations using denormalized numbers in hardware. To date no implementation of the MIPS architecture supports denormalized number in hardware.

Computations using denormalized numbers can also be supported by software emulation. Whenever a floating-point operation detects that it is about to either generate a denormalized result or begin calculating using a denormalized operand, it can abort the operation and trap to the operating system. A routine in the kernel, called *softfp*, emulates the computation using an algorithm that works correctly for denormalized numbers and deposits the result in the destination register. The operating system then

resumes the application program, which is completely unaware that a floating-point operation has been emulated in software rather than executed in hardware. Emulation via *softfp* is the normal execution environment on all IRIX platforms today.

The problem with the software emulation approach is two-fold. Firstly, emulation is slow. Computations using denormalized operands frequently generate denormalized results. So, once an application program creates a denormalized intermediate value, the execution speed of the application drastically slows down as it propagates more and more denormalized intermediate results by software emulation. If the application truly requires representation of denormalized numbers in order to perform correctly, then the slowdown is worthwhile. But in many cases the application also performs correctly if all the denormalized intermediate results were rounded to zero. For these applications software emulation of denormalized computations is just a waste of time.

The second problem with software emulation is that it demands precise floating-point exceptions. In order for *softfp* to substitute the result of an arbitrary floating-point instruction, the hardware must be capable of aborting an already-executing floating-point instruction based on the value of the input operand or result, aborting any subsequent floating-point instruction that may already be in progress, and trapping to the operating system in such a way that the program can be resumed. Providing precise exceptions on floating-point operations is always difficult since they take multiple cycles to execute and should be overlapped with other operations. It becomes much more difficult when, to achieve higher performance, operations are executed in a different order than that specified in the program. In this case instructions logically after a floating-point operation that needs to be emulated may have already completed execution! While there are known techniques to allow *softfp* to emulate the denormalized operation, all these techniques require considerable additional hardware.

## Performance Mode Definition

In defining a new floating-point execution environment there are several goals:

- Give sufficient latitude to facilitate the design of all conceivable future high performance processors.
- Fully comply with the IEEE Standard via a combination of compiler, library, operating system and hardware.
- Preserve the correct operation of a broad subset of existing applications compiled under the preexisting floating-point environment (which we now call Precise Exception Mode).
- Provide a software-only solution to retrofit the new mode on existing hardware.

The first goal is important because we do not want to be changing floating-point architectures with every implementation. The second goal is important because we want to continue to say we have "IEEE arithmetic" machines. The third goal gives our customers a smooth transition path. The fourth goal lets our customers upgrade their old machines.

Performance mode is defined by omitting denormalized numbers from the IEEE Standard and by deleting the requirement to precisely trap floating-point exceptions. Referring to Table 5-2, the behavior of an operation that produces result values A-E in Performance Mode is defined as follows.

**Table 5-2** Operation Results Using Performance Mode

| Value | Input | Result | Flags |
|---|---|---|---|
| A: TooSmall | - | 0 or minN | U=1, I=1 |
| B: ExactDenorm | 0 or min | 0 or minN | U=1, I=1 |
| C: InexactDenorm | - | 0 or minN | U=1, I=1 |
| D: ExactNorm | D | D | U=0, I=0 |
| E: InexactNorm | - | rnd(E) | U=0, I=1 |

Tiny results are mapped to either zero or the minimum normalized number, depending on the current Rounding Mode. Note that the inexact flag *I* is set in case *B* because although there is an exact denormalized representation for that value, it is not being used. Denormalized input operands, *B*, are similarly mapped to zero or *minN*. Note that there are no inexact inputs since they cannot be represented. The normalized cases are identical to those in Precise Exception mode.

All IEEE Standard floating-point exceptions are trapped imprecisely in Performance Mode. Regardless of whether the exceptions are enabled or disabled, the result register specified by the offending instruction is unconditionally updated as if all the exceptions are disabled, and the exception conditions are accumulated into the flag bits of the *FSR*, the floating point control and status register.

There are two classes of exceptions in Performance Mode. If any flag bit (invalid operation, division by zero, overflow, underflow, inexact) and its corresponding enable bit are both set, then an imprecise trap occurs at or after the offending instruction up to the next trap barrier. In addition, if *FS*=0 (*FS* is a special control bit in the *FSR*) then an imprecise trap occurs when a tiny result that would be represented as a denormalized number gets mapped into zero or minN. *FS*=0 also causes an imprecise trap if an input operand is a denormalized number that gets trapped into zero or *minN*.

A floating-point trap barrier is defined by a code sequence that begins with an instruction moving the *FSR* to an integer register and concludes with an instruction that uses the integer register containing the *FSR* contents. Any number of other instructions are allowed in between as long as they are not floating-point computation instructions (that is, they cannot set flag bits). All imprecise floating-point traps that occur on behalf of an instruction before the barrier are guaranteed to have occurred before the conclusion of the barrier. At the conclusion of the barrier the flag bits accurately reflect the accumulated results of all floating point instructions before the barrier. The floating-point barrier is defined in this way to give implementations maximum flexibility in overlapping integer and floating-point operations serialization of the two units is deferred as late as possible to avoid performance loss.

The cause bits of the *FSR* present a serious problem in Performance Mode. Ideally they should contain the result of the latest floating-point operation. However, this may be very difficult or expensive to implement when floating-point instructions are issued or even completed out of order. In order to maximize the opportunity for correctly running existing binaries and yet retain full flexibility in future out-of-order implementations, the cause bits of the *FSR* are defined to be cleared by each floating-point operation. Future applications, however, should avoid looking at the cause bits, and instead should use the flag bits.

The selection of Performance or Precise Exception Mode is defined as a protected or kernel-only operation. This is necessary for several reasons. When executing existing binaries that operate correctly in Performance Mode, we do not want the program to accidently go into Precise Exception Mode. Since existing programs regularly clear the entire *FSR* when they want to clear just the rounding mode bits, Performance Mode cannot be indicated by setting a bit in the *FSR*. On the other hand, existing programs that must run in Precise Exception Mode must not accidently go into Performance Mode. Thus Performance Mode cannot be indicated by clearing a bit in the *FSR* either. We cannot use a new user-accessible floating-point control register to indicate Performance Mode because when a new program running on an existing processor that does not understand Performance Mode writes to this nonexisting control register, it is undefined what happens to the floating-point unit. Finally, on the R8000 there are implementation restrictions on what instructions may proceed and follow a mode change, so such changes can only be done safely by the kernel.

## R8000 and R4400 Implementations

The R4000 already made a step in the direction of Performance Mode by defining the *FS* bit in the *FSR*, the floating-point control and status register. When *FS* is set, denormalized results are flushed to zero or *minN* depending on current Rounding Mode instead of causing an unimplemented operation exception. This feature eliminates the most objectionable aspect of Precise Exception Mode, namely the slow propagation of denormalized intermediate results via *softfp*. However, it does not eliminate the need to precisely trap floating-point exceptions because denormalized input operands must still be handled by *softfp*.

The R8000 extends the R4000 floating-point architecture to include another control bit whose states are labeled *PERF* and *PREX*, for Performance Mode and Precise Exception Mode, respectively. In Performance Mode the R8000 hardware (see Table 5-3) does the following:

**Table 5-3** R8000 Performance Mode

| Value | Input | Result | Flags |
|---|---|---|---|
| A:TooSmall | - | 0 or minN | U=1, I=1 E=1 if FS=0 |
| B:ExactDenorm | 0 | 0 or minN | U=1, I=1 E=1 if FS=0 |
| C:InexactDenorm | - | 0 or minN | U=1, I=1 E=1 if FS=0 |
| D:ExactNorm | D | D | U=0, I=0 |
| E:InexactNorm | - | rnd(E) | U=0, I=1 |

The *E* bit, which becomes sticky in Performance Mode, signifies that a denormalized number was mapped to 0 or *minN*. Note that the R8000 can only flush denormalized input operands to zero, as opposed to either zero or *minN*. This deviation is unfortunate but unlikely to be noticeable and is not scheduled to be fixed.

In Precise Exception Mode the R8000 hardware (see Table 5-4) does the following:

**Table 5-4** R8000 Precise Exception Mode

| Value | Input | Result | Flags |
|---|---|---|---|
| A:TooSmall | - | trap | U=1, I=1 |
| B:ExactDenorm | trap | trap | U=1, I=1 |
| C:InexactDenorm | - | trap | U=1, I=1 |
| D:ExactNorm | D | D | U=0, I=0 |
| E: InexactNorm | - | rnd(E) | U=0, I=1 |

Unlike the R4400, the R8000 totally ignores the *FS* bit in this case and relies on **softfp** to emulate the result. This simplification degrades the performance of Precise Exception Mode but does not alter the results.

Performance Mode is retrofitted on the R4400 by enhancing the kernel and **softfp**. The emulation of Performance Mode deviates from the definition in that the cause bits of the *FSR* are not cleared by every floating-point operation, but instead continue to be updated based on the result of the operation. This deviation is necessary to achieve acceptable performance.

## Full IEEE Compliance in Performance Mode

Full IEEE Standard compliance including precise exceptions and support for denormalized computations is possible in Performance Mode with help from the compiler. Although most applications never need it, some programming languages (for example, Ada) require more precision in exception reporting than what the hardware provides in Performance Mode. Also, a small number of algorithms really do need to perform computations on denormalized numbers to get accurate results.

The concept behind achieving precise exceptions in Performance Mode relies on two observations. Firstly, a program can be divided into a sequence of blocks, each block containing a computation section followed by an update section. Computation sections can read memory, calculate intermediate results, and store to temporary locations which are not program variables, but they cannot modify program visible state. Update sections

store to program visible variables, but they do not compute. Floating-point exceptions can only occur on behalf of instructions in computation sections, and can be confined to computation sections by putting a floating-point exception barrier at the end of computation sections.

Secondly, it is always possible to partition the computation and update sections in such a way that the computation sections are infinitely reexecutable. We call such computation sections Idempotent Code Units.

Intuitive, an ICU corresponds to the right hand side of an assignment statement, containing only load and compute instructions without cyclic dependencies. In practice ICUs can also contain stores if they spill to temporary locations. As long as the input operands remain the same, the result generated by an ICU remains the same no matter how many times the ICU is executed. We achieve precise exceptions in Performance Mode by compiling the program (or parts of the program) into blocks of the following form (registers are marked with %):

```
restart = current pc      #restart point
. . .                     #Idempotent Code Unit
%temp = FSR               #trap barrier
%add %r0 = %r0 + %temp    #end of trap barrier
    Fixup:nop             #break point inserted here
    . . .                 #update section
```

A map specifying the locations of all ICU's and their Fixup point is included in the binary file, and the program is linked with a special floating-point exception handler.

When a floating-point exception trap occurs, the handler switches the processor to Precise Exception mode, inserts a break point at location Fixup, and re-executes the ICU by returning to the program counter in *%restart*. This time the exception(s) are trapped precisely, and denormalized computations can be emulated. When the program reaches the break point inserted at **Fixup**, another trap occurs to allow the handler to remove the break point, reinsert the nop, and return to Performance Mode.

## Application Bringup and Debugging

The first step in bringing up applications is to compile and run them at the lowest optimization level. Once a stable baseline of code is established, you can compare it with code that does not run, to isolate problems. This methodology is expanded upon as follows:

- Use the source base of a 32-bit working application which is compiled *-mips2* (compiled with the *ucode* compilers).
- Compile **-64 -mips4 -g** and fix source problems. Refer to other sections in this Guide for help on general porting issues.
- Run the **-64 -mips4 -g** binaries on a PowerChallenge, R8000 IRIX 6 system. At this point the ucode 32-bit binaries, should also be run, side by side, on the PowerChallenge system to help isolate where the problems are creeping in.
- Repeat the previous step, going up the optimization scale.
- Compile **-64**, **-mips4**, **-O3** and tune applications on Power Challenge to get the best perfromance. Refer to Chapter 6 in this Guide for help on tuning for the R8000.

The good news is that if you can get everything working **-64 -mips4 -g**, then you have a 64-bit baseline with which to compare non-working code. The MIPSpro compiler allows you to link object files compiled at different optimizations. By repeatedly linking objects from the working baseline, with those of the non-working set and testing the resulting application, you should be able to identify which objects from the non-working set are bad.

Once you have narrowed down your search to a small area of code, use the *dbx* debugger. You can then compare the variables and program behavior of your working application against the variables and program behavior of the code that fails. By isolating the problem to a line of code, you can then find the cause of the failure.

*Chapter 6*

# Performance Tuning for the R8000 and R10000

This chapter outlines techniques for tuning the performance of your R8000 and R10000 applications. It contains five sections:

- The first section provides an architectural overview of the R8000 and R10000. This will serve as background for the software pipelining discussion.
- The second section presents the compiler optimization technique of software pipelining, which is crucial to getting optimal performance on the R8000. It shows you how to read your software pipelined code and how to understand what it does.
- The third section uses matrix multiplies as a case study on loop unrolling.
- The fourth section describes how the *IVDEP* directive can be used in Fortran to gain performance.
- The final section describes how using vector intrinsic functions can improve the performance of your program.

## Architectural Overview

Table 6-1 illustrates the main architectural features of the R8000 and R10000 microprocessors. Both can execute the MIPS IV instruction set and both can issue up to four instructions per cycle, of which two can be integer instructions and two can be floating point instructions. The R8000 can execute up to two *madd* instructions per cycle, while the R10000 can only execute one per cycle.The R8000 can issue two memory instructions per cycle while the R10000 can issue one per cycle. On the other hand, the R10000 operates at a much faster clock frequency and is capable of out-of-order execution. This makes it a better candidate for executing programs that were not explicitly tuned for it.

**Table 6-1** Architectural Features of the R8000 and R10000

| Feature | R8000 | R10000 |
|---|---|---|
| ISA | MIPS IV | MIPS IV |
| Frequency | 90 Mhz | 200 Mhz |
| Peak MFLOPS | 360 | 400 |
| Total Number of Instructions per cycle | 4 | 4 |
| Number of Integer Instructions per cycle | 2 | 2 |
| Number of Floating Point Instructions per cycle | 2 | 2 |
| Number of Multiply-Add Instructions per cycle. | 2 | 1 |
| Number of Load/Store Instructions per cycle | 2 | 1 |
| Out-of-order Instruction execution | No | Yes |

## Software Pipelining

The purpose of this section is to give you an overview of software pipelining and to answer these questions:

- Why software pipelining delivers better performance
- How software pipelined code looks
- How to diagnose what happened when things go wrong

### Why Software Pipelining Delivers Better Performance

To introduce the topic of software pipelining, let's consider the simple DAXPY loop (double precision *a* times *x* plus *y*) shown below.

```
DO i = 1, n
    v(i) = v(i) + X * w(i)
ENDDO
```

On the MIPS IV architecture, this can be coded as two load instructions followed by a *madd* instruction and a store. See Figure 6-1.

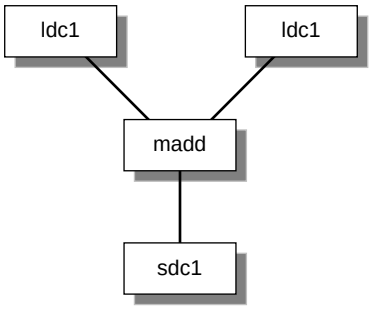


**Figure 6-1** A Simple DAXPY Implementation

This simplest schedule achieves the desired result after five cycles. Since the R8000 architecture can allow up to two memory operations and two floating point operations in the same cycle, this simple example uses only one tenth of the R8000's peak megaflops. There is also a delay of three cycles before the results of the *madd* can be stored.

```
0: ldc1 ldc1 madd
1:
2:
3:
4: sdc1
```

A loop unrolling by four schedule improves the performance to one quarter of the R8000's peak megaflops.

```
0: ldc1    ldc1    madd
1: ldc1    ldc1    madd
2: ldc1    ldc1    madd
3: ldc1    ldc1    madd
4: sdc1
5: sdc1
6: sdc1
7: sdc1
```

But this schedule does not take advantage of the R8000's ability to do two stores in one cycle.The best schedule that could be achieved would look like the following:

```
0: ldc1    ldc1
1: ldc1    ldc1    madd    madd
2: sdc1    sdc1
```

It uses 1/3 of the R8000's peak megaflops. But there still is a problem with the *madd* **sdc1** interlock delay. Software pipelining addresses this problem.

Software pipelining allows you to mix operations from different loop iterations in each iteration of the hardware loop. Thus the store instructions (which would interlock) in the above example could be used to store the results of different iterations. This can look like the following:

```
L1:
0: t1 = ldc1    t2 = ldc1   t3 = madd t1 X t2
1: t4 = ldc1    t5 = ldc1   t6 = madd t4 X t5
2: sdc1 t7      sdc1 t8     beq DONE

3: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
4: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
5: sdc1 t3      sdc1 t6     bne L1

DONE:
```

The stores in this loop are storing the *madd* results from previous iterations. But, in general, you could mix any operations from any number of different iterations. Also, note that every loop replication completes two loop iterations in 3 cycles.

In order to properly prepare for entry into such a loop, a windup section of code is added. The windup section sets up registers for the first stores in the main loop. In order to exit the loop properly, a winddown section is added. The winddown section performs the final stores. Any preparation of registers needed for the winddown is done in the compensation section. The winddown section also prevents speculative operations.

```
windup:
0: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
1: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
L1:
0: t1 = ldc1    t2 = ldc1   t3 = madd t1 X t2
1: t4 = ldc1    t5 = ldc1   t6 = madd t4 X t5
2: sdc1 t7      sdc1 t8     beq compensation1

3: t1 = ldc1    t2 = ldc1   t7 = madd t1 X t2
4: t4 = ldc1    t5 = ldc1   t8 = madd t4 X t5
5: sdc1 t3      sdc1 t6     bne L1

winddown:
0: sdc1 t7      sdc1 t8     br ALLDONE
```

```
compensation1:
0: t7 = t3      t8 = t6
1: br winddown

ALLDONE:
```

Our example loop always does loads from at least 4 iterations, so we don't want to start it if we don't want speculative operations and if the trip count is less than 4. The following generalizes our example into a map of a software pipelined loop:

```
/* Precondition for unroll by 2 */
   do i = 1, n mod 2
      original loop
   enddo

   if ( n-i < 4 ) goto simple_loop
windup:
      ...
/* fp - fence post */
      fp = fp - peel_amount
      ...

swp replication 0:
      ...
      if ( i == fp ) goto compensation 0
      ...
swp replication n - 1:
      ...
      if ( i != fp ) goto swp replication 0

compensation n-1:

winddown:
   ...
   goto done

compensation 0:
/* Move registers to set up winddown */
   rx = ry
   ...
   goto winddown
   ...
```

```
compensation n - 2:
   ...
/* Move registers to set up winddown */
   rx = ry
   goto winddown

simple_loop:

   do i = i,n
      original_loop
   enddo

done:
```

In practice, software pipelining can make a huge difference. Sample benchmark performances see more than 100% improvement when compiled with software pipelining. This makes it clear that in order to get the best performance on the R8000 (**-mips4**), you should compile your application to use software pipelining (**-O3 -r8000**).

## Software Pipelining on the R10000

Scheduling for the R10000 (**-r10000**) is somewhat different. First of all, since the R10000 can execute instructions out-of-order, static scheduling techniques are not as critical to its performance. On the other hand, the R10000 supports the prefetch (*pref*) instruction. This instruction is used to load data into the caches before it is needed, reducing memory delays for programs that access memory predictably. In the schedule below, the compiler generates prefetch instructions.

Since the R10000 can execute only one memory instruction and only one *madd* instruction per cycle, there are two open slots available for integer instructions. The R10000 has a delay of three cycles before the results of the *madd* can be stored. It also has delays of three cycles and one cycle before the *madd* can use the result of a load for multiplication and addition, respectively.

The following schedule shows four replications from a daxpy inner loop scheduled for the R10000. Notice how the operands of the *madd* instruction are loaded well ahead of the actual execution of the instruction itself. Notice also that the final replication contains two prefetch instructions. Use of the prefetch instruction is enabled by default with the **-r10000** flag. The other replications each have a hole where the first prefetch instruction

is placed. Had prefetch been turned off through the **-LNO:prefetch=0** option, each replication could have been scheduled in three cycles.

```
L1:

0: t0 = ldc1
1:
2: t7 = ldc1
3: sdc1 t2
4: t2 = madd t4 X t5    beq compensation0

0: t4 = ldc1
1:
2: t3 = ldc1
3: sdc1 t2
4: t2 = madd t0 X t1    beq compensation1

0: t0 = ldc1
1:
2: t5 = ldc1
3: sdc1 t2
4: t2 = madd t4 X t7    beq compensation2

0: t4 = ldc1
1: pref
2: t1 = ldc1
3: sdc1 t2
4: t2 = madd t0 X t3    bne L1   pref
```

## Looking at the Code Produced by Software Pipelining

The proper way to look at the assembly code generated by software pipelining is to use the **-S** compiler switch. This is vastly superior to using the disassembler (*dis*) because the **-S** switch adds annotations to the assembly code which name out the sections described above.

The annotations also provide useful statistics about the software pipelining process as well as reasons why certain code did not pipeline. To get a summary of these annotations do the following:

```
%f77 -64 -S -O3 -mips4 foo.f
```

This creates an annotated *.s* file

```
%grep '#<swp' foo.s
```

**#<swpf** is printed for loops that failed to software pipeline. #<*swps* is printed for statistics and other info about the loops that did software pipeline.

**Example 1: Output from Using the -S Compiler Switch**

```
%cat test.f
        program test
        real*8 a x(100000),y(100000)
        do i = 1, 2000
           call daxpy(3.7, x, y, 100000)
        enddo
        stop
        end

        subroutine daxpy(a, x, y, nn)
        real*8 a x(*),y(*)
        do i = 1, nn, 1
           y(i) = y(i) + a * x(i)
        enddo
        return
        end
%f77 -64 -r8000 -mips4 -O3 -S test.f
%grep swps test.s
#<swps>
#<swps> Pipelined loop line 12 steady state
#<swps>
#<swps> 25 estimated iterations before pipelining
#<swps>  4 unrollings before pipelining
#<swps>  6 cycles per 4 iterations
#<swps>  8 flops       ( 33% of peak) (madds count as 2)
#<swps>  4 flops       ( 33% of peak) (madds count as 1)
#<swps>  4 madds       ( 33% of peak)
#<swps> 12 mem refs    (100% of peak)
#<swps>  3 integer ops ( 25% of peak)
#<swps> 19 instructions( 79% of peak)
#<swps>  2 short trip threshold
#<swps>  7 integer registers used
#<swps> 14 float registers used
#<swps>
```

This example was compiled with scheduling for the R8000. It shows that the inner loop starting at line 12 was software pipelined. The loop was unrolled four times before pipelining. It used 6 cycles for every four loop iterations and calculated the statistics as follows:

- If each *madd* counts as two floating point operations, the R8000 can do four floating point operations per cycle (two *madds*), so its peak for this loop is 24. Eight floating point references are 8/24 or 33% of peak. The figure for *madds* is likewise calculated.
- If each *madd* counts as one floating point operation, the R8000 can do two floating point operations per cycle, so its peak for this loop is 12. Four floating point operations are 4/12 or 33% of peak.
- The R8000 can do two memory operations per cycle, so its peak for this loop is 12. Twelve memory references are 12/12 or 100% of peak.
- The R8000 can do two integer operations per cycle, so its peak for this loop is 12. Three integer references are 3/12 or 25% of peak.
- The R8000 can do four instructions per cycle, so its peak for this loop is 24. Nineteen instructions are 19/24 or 79% of peak. The statistics also point out that loops of less than 2 iterations would not go through the software pipeline replication area, but would be executed in the **simple_loop** section shown above and that a total of seven integer and eight floating point registers were used in generating the code.

If the example would have been compiled with scheduling for the R10000, the following results would have been obtained.

```
%f77 -64 -r10000 -mips4 -O3 -S test.f
%grep swps test.s
#<swps>
#<swps> Pipelined loop line 12 steady state
#<swps>
#<swps>    25 estimated iterations before pipelining
#<swps>     4 unrollings before pipelining
#<swps>    14 cycles per 4 iterations
#<swps>     8 flops       ( 28% of peak) (madds count as 2)
#<swps>     4 flops       ( 14% of peak) (madds count as 1)
#<swps>     4 madds       ( 28% of peak)
#<swps>    12 mem refs    ( 85% of peak)
#<swps>     3 integer ops ( 10% of peak)
#<swps>    19 instructions( 33% of peak)
#<swps>     2 short trip threshold
#<swps>     7 integer registers used
#<swps>    15 float registers used
#<swps>
```

The statistics are tailored to the R10000 architectural characteristics. They show that the inner loop starting at line 12 was unrolled four times before being pipelined. It used 14 cycles for every four loop iterations and the percentages were calculated as follows:

- The R10000 can do two floating point operations per cycle (one *multiply* and one *add*), so its floating point operations peak for this loop is 28. If each *madd* instruction counts as a *multiply* and an *add*, the number of operations in this loop are 8/28 or 28% of peak.
- If each *madd* counts as one floating point instruction, the R10000 can do two floating point operations per cycle (one *multiply* and one *add*), so its peak for this loop is 28. Four floating point operations (four *madds*) are 4/28 or 14% of peak.
- The R10000 can do one *madd* operation per cycle, so its peak for this loop is 14. Four *madd* operations are 4/14 or 28% of peak.
- The R10000 can do one memory operation per cycle, so its peak for this loop is 14. Three memory references are 12/14 or 85% of peak.

**Note:** prefetch operations are sometimes not needed in every replication. The statistics will miss them if they are not in replication 0 and they will understate the number of memory references per cycle while overstating the number of cycles per iteration.

- The R10000 can do two integer operations per cycle, so its peak for this loop is 28. Three integer operations are 3/28 or 10% of peak.
- The R10000 can do four instructions per cycle, so its peak for this loop is 56. Nineteen instructions are 19/56 or 33% of peak. The statistics also point out that loops of less than 2 iterations would not go through the software pipeline replication area, but would be executed in the **simple_loop** section shown above and that a total of seven integer and fifteen floating point registers were used in generating the code.

## How to Diagnose What Went Wrong

When you don't get much improvement in your application's performance after compiling with software pipelining, you should ask the following questions and consider some possible answers:

1. Did it software pipeline at all?

   Software pipelining works only on inner loops. What is more, inner loops with subroutine calls or complicated conditional branches do not software pipeline.

2. How well did it pipeline?

   Look at statistics in the *.s* file.

3. What can go wrong in code that was software pipelined?

   Your generated code may not have the operations you expected.

4. Think about how you would hand code it.

   What operations did it need?

   Look at the loop in the *.s* file.

   Is it very different? Why?

   Sometimes this is human error. (Improper code, or typo.)

   Sometimes this is a compiler error.

Perhaps the compiler didn't schedule tightly. This can happen because there are unhandled recurrence divides (Divides in general, are a problem) and because there are register allocation problems (running out of registers).

## Matrix Multiply – A Tuning Case Study

Matrix multiplication illustrates some of the issues in compiling for the R8000 and R10000. Consider the simple implementation.

```
do j = 1,n
   do i = 1,m
      do k = 1 , p
         c(i,j) = c(i,j) - a(i,k)*b(k,j)
```

As mentioned before, the R8000 is capable of issuing two *madds* and two memory references per cycle. This simple version of matrix multiply requires 2 loads in the inner loop and one *madd*. Thus at best, it can run at half of peak speed. Note, though, that the same locations are loaded on multiple iterations. By unrolling the loops, we can eliminate some of these redundant loads. Consider for example, unrolling the outer loop by 2.

```
do j = 1,n,2
   do i = 1,m
      do k = 1 , p
         c(i,j) = c(i,j) - a(i,k)*b(k,j)
         c(i,j+1) = c(i,j+1) - a(i,k)*b(k,j+1)
```

We now have 3 loads for two *madds*. Further unrolling can bring the ratio down further. On the other hand, heavily unrolled loops require many registers. Unrolling too much can lead to register spills. The loop nest optimizer (LNO) tries to balance these trade-offs.

Below is a good compromise for the matrix multiply example on the R8000. It is generated by LNO using the **-O3** and **-r8000** flags. The listing file is generated using **-FLIST:=ON** option.

```
%f77 -64 -O3 -r8000 -FLIST:=ON mmul.f
%cat mmul.w2f.f
C ************************************************************
C Fortran file translated from WHIRL Mon Mar 22 09:31:25 1999
C ************************************************************

        PROGRAM MAIN
        IMPLICIT NONE
C
C       **** Variables and functions ****
C
        REAL*8 a(100_8, 100_8)
        REAL*8 b(100_8, 100_8)
        REAL*8 c(100_8, 100_8)
        INTEGER*4 j
        INTEGER*4 i
        INTEGER*4 k
C
C       **** Temporary variables ****
C
        REAL*8 mi0
        REAL*8 mi1
        REAL*8 mi2
        REAL*8 mi3
        REAL*8 mi4
        REAL*8 mi5
        REAL*8 mi6
        REAL*8 mi7
        REAL*8 mi8
        REAL*8 mi9
        REAL*8 mi10
        REAL*8 mi11
        REAL*8 c_1
        REAL*8 c_
        REAL*8 c_0
        REAL*8 c_2
        REAL*8 c_3
```

```
      REAL*8 c_4
      REAL*8 c_5
      REAL*8 c_6
      REAL*8 c_7
      REAL*8 c_8
      REAL*8 c_9
      REAL*8 c_10
      INTEGER*4 i0
      REAL*8 mi12
      REAL*8 mi13
      REAL*8 mi14
      REAL*8 mi15
      INTEGER*4 k0
      REAL*8 c_11
      REAL*8 c_12
      REAL*8 c_13
      REAL*8 c_14
C
C     **** statements ****
C
      DO j = 1, 8, 3
        DO i = 1, 20, 4
          mi0 = c(i, j)
          mi1 = c((i + 3), (j + 2))
          mi2 = c((i + 3), (j + 1))
          mi3 = c(i, (j + 1))
          mi4 = c((i + 3), j)
          mi5 = c((i + 2), (j + 2))
          mi6 = c((i + 2), (j + 1))
          mi7 = c(i, (j + 2))
          mi8 = c((i + 2), j)
          mi9 = c((i + 1), (j + 2))
          mi10 = c((i + 1), (j + 1))
          mi11 = c((i + 1), j)
          DO k = 1, 20, 1
            c_1 = mi0
            mi0 = (c_1 -(a(i, k) * b(k, j)))
            c_ = mi3
            mi3 = (c_ -(a(i, k) * b(k, (j + 1))))
            c_0 = mi7
            mi7 = (c_0 -(a(i, k) * b(k, (j + 2))))
            c_2 = mi11
            mi11 = (c_2 -(a((i + 1), k) * b(k, j)))
            c_3 = mi10
            mi10 = (c_3 -(a((i + 1), k) * b(k, (j + 1))))
```

```
      c_4 = mi9
      mi9 = (c_4 -(a((i + 1), k) * b(k, (j + 2))))
      c_5 = mi8
      mi8 = (c_5 -(a((i + 2), k) * b(k, j)))
      c_6 = mi6
      mi6 = (c_6 -(a((i + 2), k) * b(k, (j + 1))))
      c_7 = mi5
      mi5 = (c_7 -(a((i + 2), k) * b(k, (j + 2))))
      c_8 = mi4
      mi4 = (c_8 -(a((i + 3), k) * b(k, j)))
      c_9 = mi2
      mi2 = (c_9 -(a((i + 3), k) * b(k, (j + 1))))
      c_10 = mi1
      mi1 = (c_10 -(a((i + 3), k) * b(k, (j + 2))))
    END DO
    c((i + 1), j) = mi11
    c((i + 1), (j + 1)) = mi10
    c((i + 1), (j + 2)) = mi9
    c((i + 2), j) = mi8
    c(i, (j + 2)) = mi7
    c((i + 2), (j + 1)) = mi6
    c((i + 2), (j + 2)) = mi5
    c((i + 3), j) = mi4
    c(i, (j + 1)) = mi3
    c((i + 3), (j + 1)) = mi2
    c((i + 3), (j + 2)) = mi1
    c(i, j) = mi0
  END DO
END DO
DO i0 = 1, 20, 4
  mi12 = c(i0, 10_8)
  mi13 = c((i0 + 1), 10_8)
  mi14 = c((i0 + 3), 10_8)
  mi15 = c((i0 + 2), 10_8)
  DO k0 = 1, 20, 1
    c_11 = mi12
    mi12 = (c_11 -(a(i0, k0) * b(k0, 10_8)))
    c_12 = mi13
    mi13 = (c_12 -(a((i0 + 1), k0) * b(k0, 10_8)))
    c_13 = mi15
    mi15 = (c_13 -(a((i0 + 2), k0) * b(k0, 10_8)))
    c_14 = mi14
    mi14 = (c_14 -(a((i0 + 3), k0) * b(k0, 10_8)))
  END DO
  c((i0 + 2), 10_8) = mi15
```

```
          c((i0 + 3), 10_8) = mi14
          c((i0 + 1), 10_8) = mi13
          c(i0, 10_8) = mi12
        END DO
        WRITE(6, '(F18.10)') c(9_8, 8_8)
        STOP
        END ! MAIN
```

The outermost loop is unrolled by three as suggested above and the second loop is unrolled by a factor of four. Note that we have not unrolled the inner loop. The code generation phase of the compiler back end can effectively unroll inner loops automatically to eliminate redundant loads.

In optimizing for the R10000, LNO also unrolls the outermost loop by three, but unrolls the second loop by 5. LNO can also automatically tile the code to improve cache behavior. You can use the **-LNO:** option group flags to describe your cache characteristics to LNO. For more information, please consult the *MIPSpro Compiling, Debugging and Performance Tuning Guide*.

```
%f77 -64 -O3 -r10000 -FLIST:=ON mmul.f
%cat mmul.w2f.f

C ******************************************************
C Fortran file translated from WHIRL Mon Mar 22 09:41:39 1999
C ******************************************************

        PROGRAM MAIN
        IMPLICIT NONE
C
C       **** Variables and functions ****
C
        REAL*8 a(100_8, 100_8)
        REAL*8 b(100_8, 100_8)
        REAL*8 c(100_8, 100_8)
        INTEGER*4 j
        INTEGER*4 i
        INTEGER*4 k
C
C       **** Temporary variables ****
C
        REAL*8 mi0
        REAL*8 mi1
        REAL*8 mi2
        REAL*8 mi3
```

```
      REAL*8 mi4
      REAL*8 mi5
      REAL*8 mi6
      REAL*8 mi7
      REAL*8 mi8
      REAL*8 mi9
      REAL*8 mi10
      REAL*8 mi11
      REAL*8 mi12
      REAL*8 mi13
      REAL*8 mi14
      INTEGER*4 i0
      REAL*8 mi15
      REAL*8 mi16
      REAL*8 mi17
      REAL*8 mi18
      REAL*8 mi19
      INTEGER*4 k0
      REAL*8 mi20
      REAL*8 mi21
      REAL*8 mi22
      REAL*8 mi23
      REAL*8 mi24
C
C     **** statements ****
C
      DO j = 1, 8, 3
        DO i = 1, 20, 5
          mi0 = c(i, j)
          mi1 = c((i + 4), (j + 2))
          mi2 = c((i + 4), (j + 1))
          mi3 = c(i, (j + 1))
          mi4 = c((i + 4), j)
          mi5 = c((i + 3), (j + 2))
          mi6 = c((i + 3), (j + 1))
          mi7 = c(i, (j + 2))
          mi8 = c((i + 3), j)
          mi9 = c((i + 2), (j + 2))
          mi10 = c((i + 2), (j + 1))
          mi11 = c((i + 1), j)
          mi12 = c((i + 2), j)
          mi13 = c((i + 1), (j + 2))
          mi14 = c((i + 1), (j + 1))
          DO k = 1, 20, 1
            mi0 = (mi0 -(a(i, k) * b(k, j)))
```

```
      mi3 = (mi3 -(a(i, k) * b(k, (j + 1))))
      mi7 = (mi7 -(a(i, k) * b(k, (j + 2))))
      mi11 = (mi11 -(a((i + 1), k) * b(k, j)))
      mi14 = (mi14 -(a((i + 1), k) * b(k, (j + 1))))
      mi13 = (mi13 -(a((i + 1), k) * b(k, (j + 2))))
      mi12 = (mi12 -(a((i + 2), k) * b(k, j)))
      mi10 = (mi10 -(a((i + 2), k) * b(k, (j + 1))))
      mi9 = (mi9 -(a((i + 2), k) * b(k, (j + 2))))
      mi8 = (mi8 -(a((i + 3), k) * b(k, j)))
      mi6 = (mi6 -(a((i + 3), k) * b(k, (j + 1))))
      mi5 = (mi5 -(a((i + 3), k) * b(k, (j + 2))))
      mi4 = (mi4 -(a((i + 4), k) * b(k, j)))
      mi2 = (mi2 -(a((i + 4), k) * b(k, (j + 1))))
      mi1 = (mi1 -(a((i + 4), k) * b(k, (j + 2))))
    END DO
    c((i + 1), (j + 1)) = mi14
    c((i + 1), (j + 2)) = mi13
    c((i + 2), j) = mi12
    c((i + 1), j) = mi11
    c((i + 2), (j + 1)) = mi10
    c((i + 2), (j + 2)) = mi9
    c((i + 3), j) = mi8
    c(i, (j + 2)) = mi7
    c((i + 3), (j + 1)) = mi6
    c((i + 3), (j + 2)) = mi5
    c((i + 4), j) = mi4
    c(i, (j + 1)) = mi3
    c((i + 4), (j + 1)) = mi2
    c((i + 4), (j + 2)) = mi1
    c(i, j) = mi0
  END DO
END DO
DO i0 = 1, 20, 5
  IF(IAND(i0, 31) .EQ. 16) THEN
    mi15 = c(i0, 10_8)
    mi16 = c((i0 + 1), 10_8)
    mi17 = c((i0 + 4), 10_8)
    mi18 = c((i0 + 2), 10_8)
    mi19 = c((i0 + 3), 10_8)
    DO k0 = 1, 20, 1
      mi15 = (mi15 -(a(i0, k0) * b(k0, 10_8)))
      mi16 = (mi16 -(a((i0 + 1), k0) * b(k0, 10_8)))
      mi18 = (mi18 -(a((i0 + 2), k0) * b(k0, 10_8)))
      mi19 = (mi19 -(a((i0 + 3), k0) * b(k0, 10_8)))
      mi17 = (mi17 -(a((i0 + 4), k0) * b(k0, 10_8)))
```

```
    END DO
    c((i0 + 3), 10_8) = mi19
    c((i0 + 2), 10_8) = mi18
    c((i0 + 4), 10_8) = mi17
    c((i0 + 1), 10_8) = mi16
    c(i0, 10_8) = mi15
  ELSE
    mi20 = c(i0, 10_8)
    mi21 = c((i0 + 1), 10_8)
    mi22 = c((i0 + 4), 10_8)
    mi23 = c((i0 + 2), 10_8)
    mi24 = c((i0 + 3), 10_8)
    DO k0 = 1, 20, 1
      mi20 = (mi20 -(a(i0, k0) * b(k0, 10_8)))
      mi21 = (mi21 -(a((i0 + 1), k0) * b(k0, 10_8)))
      mi23 = (mi23 -(a((i0 + 2), k0) * b(k0, 10_8)))
      mi24 = (mi24 -(a((i0 + 3), k0) * b(k0, 10_8)))
      mi22 = (mi22 -(a((i0 + 4), k0) * b(k0, 10_8)))
    END DO
    c((i0 + 3), 10_8) = mi24
    c((i0 + 2), 10_8) = mi23
    c((i0 + 4), 10_8) = mi22
    c((i0 + 1), 10_8) = mi21
    c(i0, 10_8) = mi20
  ENDIF
END DO
WRITE(6, '(F18.10)') c(9_8, 8_8)
STOP
END ! MAIN
```

## Use of the IVDEP Directive

The *IVDEP (*Ignore Vector Dependencies*)* directive was started in Cray Fortran. It is a Fortran or C *pragma* that tells the compiler to be less strict when it is deciding whether it can get parallelism between loop iterations. By default, the compilers do the safe thing: they try to prove to that there is no possible conflict between two memory references. If they can prove this, then it is safe for one of the references to pass the other.

In particular, you need to be able to perform the load from iteration *i+1* before the store from iteration *i* if you want to be able to overlap the calculation from two consecutive iterations.

Now suppose you have a loop like:

```
do i = 1, n
   a(l(i)) = a(l(i)) + ...
enddo
```

The compiler has no way to know that

```
&a(l(i)) != &a(l(i+1))
```

without knowing something about the vector *l*. For example, if every element of *l* is 5, then

```
&a(l(i)) == &a(l(i+1))
```

for all values of *i*.

But you sometimes know something the compiler doesn't. Perhaps in the example above, *l* is a permutation vector and all its elements are unique. You'd like a way to tell the compiler to be less conservative. The *IVDEP* directive is a way to accomplish this.

Placed above a loop, the statement:

```
cdir$ ivdep
```

tells the compiler to assume there are no dependencies in the code.

The MIPSpro v7.x compilers provide support for three different interpretations of the *IVDEP* directive, because there is no clearly established standard.Under the default interpretation, given two memory references, where at least one is loop variant, the compiler will ignore any loop-carried dependences between the two references. Some examples:

```
do i = 1,n
  b(k) = b(k) + a(i)
enddo
```

Use of *IVDEP* will not break the dependence since *b(k)* is not loop variant.

```
do i=1,n
  a(i) = a(i-1) + 3.
enddo
```

Use of *IVDEP* does break the dependence but the compiler warns the user that it's breaking an obvious dependence.

```
do i=1,n
  a(b(i)) = a(b(i)) + 3.
enddo
```

Use of *IVDEP* does break the dependence.

```
do i = 1,n
  a(i) = b(i)
  c(i) = a(i) + 3.
enddo
```

Use of *IVDEP* does not break the dependence on *a(i)* since it is within an iteration.

The second interpretation of *IVDEP* is used if you use the **-OPT:cray_ivdep=TRUE** command line option. Under this interpretation, the compiler uses Cray semantics. It breaks all lexically backwards dependences. Some examples:

```
do i=1,n
  a(i) = a(i-1) + 3.
enddo
```

Use of IVDEP does break the dependence but the compiler warns the user that it's breaking an obvious dependence.

```
do i=1,n
  a(i) = a(i+1) + 3.
enddo
```

Use of *IVDEP* does not break the dependence since the dependence is from the load to the store, and the load comes lexically before the store.

The third interpretation of *IVDEP* is used if you use the **-OPT:liberal_ivdep=TRUE** command line option. Under this interpretation, the compiler will break all dependences.

**Note:** *IVDEP* IS DANGEROUS! If your code really isn't free of vector dependences, you may be telling the compiler to perform an illegal transformation which will cause your program to get wrong answers. But, *IVDEP* is also powerful and you may very well find yourself in a position where you need to use it. You just have to be very careful when you do this.

## Vector Intrinsic Functions

The MIPSpro 64-bit compilers support both single and double precision versions of the following vector instrinsic functions: **asin()**, **acos()**, **atan()**, **cos()**, **exp()**, **log()**, **sin()**, **tan()**, **sqrt()**. In C they are declared as follows:

```
/* single precision vector routines */
extern  __vacosf( float *x, float *y, int count, int stridex, int stridey );
extern  __vasinf( float *x, float *y, int count, int stridex, int stridey );
extern  __vatanf( float *x, float *y, int count, int stridex, int stridey );
extern  __vcosf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vexpf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vlogf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vsinf(  float *x, float *y, int count, int stridex, int stridey );
extern  __vtanf(  float *x, float *y, int count, int stridex, int stridey );
/* double precision vector routines */
extern  __vacos( double *x, double *y, int count, int stridex, int stridey );
extern  __vasin( double *x, double *y, int count, int stridex, int stridey );
extern  __vatan( double *x, double *y, int count, int stridex, int stridey );
extern  __vcos(  double *x, double *y, int count, int stridex, int stridey );
extern  __vexp(  double *x, double *y, int count, int stridex, int stridey );
extern  __vlog(  double *x, double *y, int count, int stridex, int stridey );
extern  __vsin(  double *x, double *y, int count, int stridex, int stridey );
extern  __vtan(  double *x, double *y, int count, int stridex, int stridey )
```

The variables x and y are assumed to be pointers to non-overlapping arrays. Each routine is functionally equivalent to the following pseudo-code fragment:

```
do i = 1, count-1
    y[i*stridey] = func(x[i*stridex])
enddo
```

where *func()* is the scalar version of the vector intrinsic.

## Performance and Accuracy

The vector intrinsics are optimized and software pipelined to take advantage of the R8000's performance features. Throughput is several times greater than that of repeatedly calling the corresponding scalar function though the result may not necessarily agree to the last bit. For further information about accuracy and restrictions of the vector intrinsics please refer to your *IRIX Compiler_dev Release Notes*.

## Manual vs. Automatic Invocation

All of the vector intrinsics can be called explicitly from your program. In Fortran the following example invokes the vector intrinsic **vexpf()**.

```
real*4 x(N), y(N)
integer*8 N, i
i = 1
call vexpf$( %val(%loc(x(1))),
&            %val(%loc(y(1))),
&            %val(N),%val(i),%val(i))
end
```

The compiler at optimization level **-O3** also recognizes the use of scalar versions of these intrinsics on array elements inside loops and turns them into calls to the vector versions. If you need to turn this feature off to improve the accuracy of your results, add **-LNO:vint=OFF** to your compilation command line or switch to a lower optimization level.

*Chapter 7*

# Miscellaneous FAQ

This chapter summarizes important concepts found throughout this manual in the form of frequently asked questions and their answers.

Q. Why can't I link my 32-bit application with a 64-bit library?

A. There are differences in the subroutine calling interface.

Q. How can I see what LNO does to my code?

A. To view the file , use **-LNO:FLIST=ON**on your compile line
%f**`77 -64 -LNO:FLIST=ON foo.f`**

Q. How can I see what the automatic parallelizer does to my code when I add **-pfa** to my command line.?

A. To generate the listing file, use **-pfa keep** on your compile line
%f**`77 -pfa keep foo.f`**

Q. My */tmp* directory becomes full when I do a compile.
What should I do?

A. %**`setenv TMP_DIR /directory`** on a free partition.

Q. How do I know which compiler is being invoked?

A. *cc -show* will show each of the components as they are being run.

Q. My 64-bit shared application gets mysterious messages from *rld* at runtime.What is the problem?

A. It's possible that you are linking with 32-bit *.so's*. Check and reset your *_RLD_ROOT* environment variable.

Q. How can I avoid always setting **-32, -n32** or **-64** on my compilation command line?

A. *%setenv SGI_ABI -32*
*%setenv SGI_ABI -n32* or
*%setenv SGI_ABI -64*.

Q. How do I know if my code software pipelined OK?

A. Compile your file (say *foo.f*)*-64 -O3 -mips4 -S*. Then *grep #<swp foo.s*# **#<swpf** will be printed for loops that failed to software pipeline. **#<swps** will be printed for statistics and other info about the loops that did software pipeline.

Q. I compiled **-O**, but my code did not software pipeline. Why not?

A. Software pipelining occurs at **-O3**. **-O** implies **-O2**.

Q. Ok, I now compiled **-O3**. Why didn't my code software pipeline?

A. Does it have a call or branch in the inner loop?
Is the inner loop too long?
Does the inner loop execute too few iterations?
Are there divide recurrences in your inner loop?

Q. What predefine should my source code look at to know if its being compiled 32-bit or 64-bit.

A. One way is:
*#if (_MIPS_SZLONG == 64)*

Q. How can I force a large loop to be unrolled by the compiler?

A Add **-OPT:unroll_size=1000** to your command line?

Q. Why does my application gets different floating point numbers in 64-bit mode on an R8000 system?

A. If its compiled with optimization, the order of operations may different resulting in slightly different calculations due to rounding. *madd* instructions also round differently than a multiply and an add. The math libraries use different algorithms.

Q. If I just do %**cc hello.c** which ABI is the default?

A. If no */etc/compiler.defaults* file is present and *SGI_ABI* is not set, the 64-bit ABI is the default on R8000 systems.On all other systems, the 32-bit ABI is the default.

Q. What does the **-r10000** flag do?

A. It produces code scheduled for the R10000 microprocessor.
It enables use of the pre-fetch instruction.
It instructs the linker to use math libraries that are customized for the R10000.

## Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-2391-006.

Thank you!

## Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
  - On the Internet: techpubs@sgi.com
  - For UUCP mail (through any backbone site): *[your_site]*!sgi!techpubs
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 650-932-0801
- To send your comments by **traditional mail**, use this address:

  Technical Publications
  Silicon Graphics, Inc.
  2011 North Shoreline Boulevard, M/S 535
  Mountain View, California  94043-1389