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.

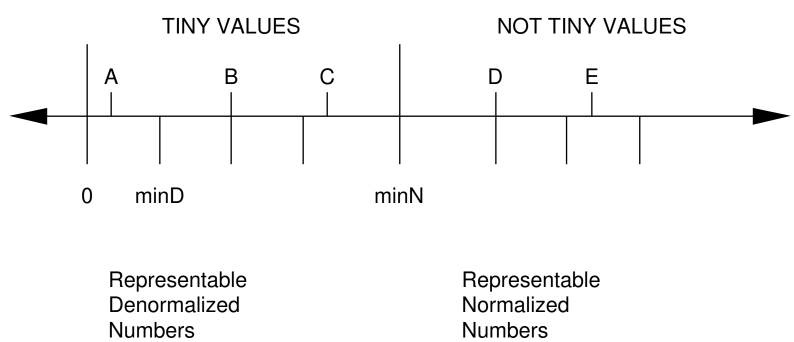

Figure 5-1 Floating Point Numbers

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.