CRAY T90 Series IEEE Floating Point Migration Issues and Solutions

Philip G. Garnatz, Cray Research, Inc., Eagan, Minnesota, U.S.A.

ABSTRACT: *Migration to the new CRAY T90 series with IEEE floating-point arithmetic presents a new challenge to applications programmers. More precision in the mantissa and less range in the exponent will likely raise some numerical differences issues. A step-by-step process will be presented to show how to isolate these numerical differences.*

Data files will need to be transferred and converted from a Cray format PVP system to and from a CRAY T90 series system with IEEE. New options to the `assign` *command allow for transparent reading and writing of files from the other types of system.*

Introduction

This paper is intended for the user services personnel and help desk staff who will be asked questions by programmers and users who will be moving code to the CRAY T90 series IEEE from another Cray PVP architecture. The issues and solutions presented here are intended to be a guide to the most frequent problems that programmers may encounter.

What is the numerical model?

New CRAY T90 series IEEE programmers might notice different answers when running on the IEEE machine than on a system with traditional Cray format floating-point arithmetic. The IEEE format has the following characteristics:

- Fewer bits of exponent
- More bits of mantissa

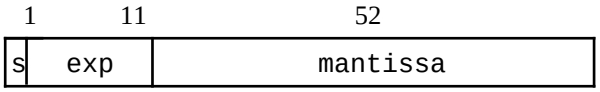

Figure 1: Cray Research single-precision IEEE format

What are the benefits of IEEE floating-point representation?

The IEEE standard floating-point number representation is being adopted to ease the movement of data between CRAY T90 series mainframes and other computational entities such as workstations and graphic displays. Other benefits are as follows:

Figure2: Cray format floating-point

- Greater precision. An IEEE floating-point number provides approximately 16 decimal digits of precision; this is about one and a half digits more precise than Cray format numbers.
- Specific representations for infinity (±∞) and nonnumeric results (such as an attempt to divide by 0) so that a user can be more confident of results (or know that there is a numeric problem), even if exception interrupts are turned off.
- Control of rounding modes. Some specific operations (for example, calculating floor and ceiling functions) benefit from control of rounding mode. Running a job with a different rounding mode can sometimes provide an easy check of numeric stability.

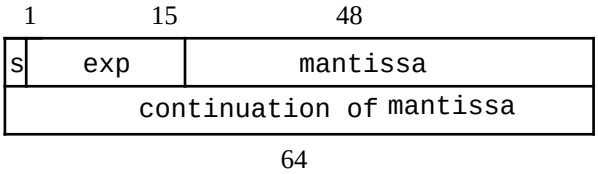

Figure 3. Extended double format on CRAY T90 series

- Consistency in handling end-cases.
- Expanded exceptions. For this implementation, the floating-point functional units generate six kinds of exceptions, which helps users handle numeric problems more easily. The IEEE floating-point standard requires five of these six exceptions.

A reduced exponent range is one disadvantage to IEEE standard floating-point numbers with respect to Cray format numbers. In almost all cases, however, the reduced range will not be noticed.

Conformance with IEEE standard

Table 1 shows features that are required by the standard.

Table 1. Required by the standard

Feature	CRAY T90 IEEE
Single format (32 bits)	No†
Double format (64 bits) (not obligatory)	Yes
Three additional, user-selectable, rounding modes	Yes
NaN (Not a Number value) compares unordered (neither greater than, less than, nor equal)	Yes
Positive and negative infinity operands and result	Yes
Signaling NaN operands (such as for uninitialized variables) supported	Yes
Quiet NaN (containing diagnostic information) preserved in operands and results	Yes
Exceptions	Yes
Signal five types of floating-point exceptions (inexact rounding, underflow, overflow, divide-by-zero, and invalid operation)	Yes
Round to nearest as default	Yes

† Cray Research provides conversion routines that let you read in and write out 32-bit data.

Nonconformance with IEEE standard

For reasons of overall system performance, CRAY T90 series systems do not conform with the following requirements of the standard:

- Single (32-bit) format will be supported only through conversion. When the 32-bit conversion capability is delivered, you will be able to convert from single binary format to double binary format on input (compile time or run time) and from double binary format back to single binary format on output. All operations during program execution are performed with double binary format or extended double binary format. The Fortran 90 `REAL` type and the C and C++ `float` and `double` types are implemented in the double binary format (64 bit). The Fortran 90 `DOUBLE PRECISION` type and the C and C++ `long double` type are implemented in the double extended binary format (128 bit).
- Denormalized numbers as operands to floating-point operations are treated as zeros. A denormalized number that results from a floating-point operation is forced to zero, and the underflow exception is raised.

The following areas of nonconformance are for reasons other than performance:

- Among the recommended features, Cray Research did not implement precise traps.
- When an overflow occurs, Cray Research does not always follow the IEEE standard completely.
- Cray Research recommends that the standard adopt an extended double format (128 bits), a format that Cray already has implemented.

NaN - Not a Number

In the Cray format floating point, there is a concept of an positive or negative indefinite number. With IEEE, there is a different concept of “Not-a-Number” or NaN. The standard calls for two formats of NaN: quiet and signaling.

Table 2 shows some floating-point number examples.

Table 2. IEEE floating-point number examples

Value	Sign	Biased Exponent (octal)	Fraction (octal)
+0	0	0000	0000000000000000
-0	1	0000	0000000000000000
+1	0	1777	0000000000000000
+1.75	0	1777	1400000000000000
-3.14	1	2000	1107534121727024 37
+GRN*	0	3776	1777777777777777
+LRN†	0	0001	0000000000000000
+∞	0	3777	0000000000000000
qNaN‡	0	3777	1xxxxxxxxxxxxxxx
sNaN‡	0	3777	0xxxxxxxxxxxxxxx

* Greatest representable number, approximately 1.8E308.

† Lease/smallest representable number, approximately 2.2E-308

‡ Quiet NaN, signaling Not-a-Number. At least 1 bit of the fraction must be a 1.

Exceptions

A floating-point operation can generate several possible exceptions. Exception status is always available in a status register SR0. These exceptions are sticky, meaning that once a status bit is set in SR0, it stays set until an operation explicitly clears it. Users can read and write status register bits as required. Reading status from SR0 clears all status register bits to zero.

An interrupt is associated with each exception status bit. Users can enable and disable the interrupts individually, or they can write all the control/mode bits simultaneously. If a particular interrupt is enabled when the corresponding status bit is already set, an interrupt is not generated. Previous Cray PVP systems generated an interrupt immediately in that case. An interrupt is generated only if an interrupt enable is set and there is an attempt to set the associated status register bit, whether or not the status bit was set previously. For instructions that can change the interrupt mode bits, issue is halted until all floating-point functional units are quiet. The following are possible exceptions:

- **Invalid**. An attempt was made to generate a result that cannot be represented by a single, numeric (real) number. Invalid is signaled for the following operations: A signaling NaN received as an input for any floating-point function. Addition or subtraction of infinities such as (+∞) + (-∞). Multiplication of 0 by ∞. Division of 0 by 0 or ∞ by ∞. Square root of any number less than 0. Signed compare (a greater-than test, for example) with either or both inputs of NaN.
- **Divide-by-zero**. The denominator of a division was zero with a finite, normal numerator. A divide-by-zero exception is not generated if the numerator is ∞ or NaN. A result of ∞ is returned except for invalid inputs (0/0 or NaN/0).
- **Overflow**. A result larger than the greatest representable number was generated. Generally, ∞ is returned; however, this depends on the current rounding mode.
- **Underflow**. A result smaller than the least representable number was generated. Zero, with the appropriate sign, is returned.
- **Inexact**. A result was returned that is different than what would have been delivered if all possible significant bits were returned. Thus, 1 divided by 3 returns 0.3333 and signals inexact, and 0.5 divided by 2 returns 0.25 and does not signal inexact because all significant bits of the operation were returned. Inexact is signaled on either overflow or underflow results, but not if the returned result is exactly 0. The interrupt associated with this exception is often turned off.
- **Exceptional input**. A floating-point functional unit received an operand of ∞ or NaN. The returned result is controlled by the kind of input operand and the function being performed. This exception is used for cases where other exceptions could be generated in normal operations so that the corresponding invalid and overflow interrupts are turned off, but the user does not expect to receive any nonfinite inputs.

Rounding

Floating-point computation is done as if an infinitely precise result will be calculated, but only the most significant bits of that computation are returned as the result. Generally, this means that one of two results can be returned, one result that is slightly greater than the infinitely precise result and one slightly less. The bits of the computation that will not be returned and the rounding mode are used to choose which result to return.

If determined to be required by the rounding mode and by any bits of less significance than that of the least-significant bit, rounding is done by adding 1 to the last place, least-significant bit of the delivered result. In rounding, the first bit below the least-significant bit of the delivered result is known as the guard bit. All bits below the guard bit are ORed together into a sticky bit. If both the round and sticky bits are 0, the result is exact and is returned without modification. If either bit is a 1, inexact is signaled, and 1 is added to the least-significant bit, depending on the rounding mode.

There are four rounding modes that apply to floating-point computation. All floating-point results are delivered with the rounding mode in effect at the time the floating-point instruction is issued. The four rounding modes are as follows:

- **Round to nearest**. The result closest to the infinitely precise result is returned. If the undelivered bits below the least-significant bit have a significance of more than half the least-significant bit, 1 is added to the least significant bit. If the result is exactly half, 1 is added if the least-significant bit is 1. (That is, the round is to even.)
- **Round up**. The more positive result closest to the infinitely precise result is returned. If the result is positive and either the guard or the sticky bit is 1, the result is rounded. If the result is negative, the result is not rounded because the unrounded result is the most positive result that is closest to the infinitely precise result.
- **Round to zero**. The result closest to zero is returned. Nothing is added to the least-significant bit. This is equivalent to truncation.
- **Round down**. The more negative result is returned. If the result is negative and either the guard or the sticky bit is 1, 1 is added to the least-significant bit. If the result is positive, nothing is added to the least-significant bit.

CF90 Status and Control Functions

The IEEE standard also includes a number of functions to control the behavior of the floating-point arithmetic, such as rounding, and to inquire about the status of individual numeric values. The CF90 compiler implements intrinsic procedures to obtain, alter, and restore a single word that contains the entire floating-point status. The flags are 6 sticky exception bits, 6 interrupt enable bits, and a rounding mode flag. Several procedures described here use named constants provided in module CRI_IEEE_DEFINITIONS.

```
GET_IEEE_STATUS(STATUS)
SET_IEEE_STATUS(STATUS)
GET_IEEE_EXCEPTIONS(STATUS)
SET_IEEE_EXCEPTIONS(STATUS)
TEST_IEEE_EXCEPTION(EXCEPTION)
CLEAR_IEEE_EXCEPTION(EXCEPTION)
SET_IEEE_EXCEPTION(EXCEPTION)
```

The possible values for EXCEPTION are the following named constants:

```
IEEE_XPTN_CRI_INVALID_OPND
IEEE_XPTN_INEXACT_RESULT
IEEE_XPTN_UNDERFLOW
IEEE_XPTN_OVERFLOW
IEEE_XPTN_DIV_BY_ZERO
IEEE_XPTN_INVALID_OPR
IEEE_XPTN_ALL
```

The purpose of the floating-point interrupt enable and disable bits, and the intrinsic procedures that manipulate them, is to let users control whether interrupts occur when IEEE exceptions are encountered during program execution.

```
GET_IEEE_INTERRUPTS(STATUS)
SET_IEEE_INTERRUPTS(STATUS)
TEST_IEEE_INTERRUPT(INTERRUPT)
ENABLE_IEEE_INTERRUPT(INTERRUPT)
DISABLE_IEEE_INTERRUPT(INTERRUPT)
```

The constants passed by the floating-point interrupt routines are as follows:

```
IEEE_NTPT_CRI_INVALID_OPND (CRI only)
IEEE_NTPT_INEXACT_RESULT
IEEE_NTPT_UNDERFLOW
IEEE_NTPT_OVERFLOW
IEEE_NTPT_DIV_BY_ZERO
IEEE_NTPT_INVALID_OPR
IEEE_NTPT_ALL
```

The intrinsics described below let users control the direction of floating-point rounding. All four IEEE rounding modes are supported. The rounding mode stays in effect until reset or until program completion. The CF90 compiler implements the IEEE standard requirement that a program begin in round-to-nearest mode.

```
GET_IEEE_ROUNDING_MODE(ROUNDING_MODE)
SET_IEEE_ROUNDING_MODE(ROUNDING_MODE)
```

The choices for STATUS when calling the preceding routines are the following constants:

```
IEEE_RM_NEAREST
IEEE_RM_POS_INFINITY
IEEE_RM_ZERO
IEEE_RM_NEG_INFINITY
```

Other Supported Functions

```
IEEE_COPY_SIGN(x, y)
IEEE_BINARY_SCALE(x, 2)
IEEE_EXPONENT(x)
IEEE_NEXT_AFTER(x, 1.0)
IEEE_FINITE(x)
IEEE_IS_NAN(x)
IEEE_CLASS(x)
IEEE_REMAINDER(x, y)
IEEE_INT(x, 1)
IEEE_REAL (X)
INT_MULT_UPPER(ii, jj) (CRI only)
```

Numerical Differences

It is well-known that the imprecise numerical representation of floating-point numbers used by different kinds of computers may introduce numerical differences due to rounding and truncation. A favorite example of this is by the results of summing the numbers: `1 + 1/2 + 1/3 + 1/4 + ...+ 1/n - 1 - 1/2 - 1/3 - 1/4 - ... - 1/n`. The algebraic sum is zero, but different answers are obtained, depending on the order of operations and the internal representation of the numbers.

On CRAY T90 series systems with Cray floating-point format: (n=500)

```
scalar frwd.  -5.131325919727203E-12
scalar back.  -5.130118552187923E-12
vector frwd.  -4.272970866026071E-14
vector back.  -3.286260152890463E-14
```

On a CRAY T90 series system with IEEE and the default rounding mode of IEEE_RM_NEAREST:

```
scalar frwd   -3.57136195616725161E-15
scalar back   -3.77475828372553224E-15
vector frwd   -8.50014503228635476E-17
vector back   -8.67361737988403547E-18
```

With rounding mode: IEEE_RM_POS_INFINITY

```
scalar frwd   2.73850820492471669E-13
scalar back   2.74447131687338697E-13
vector frwd   1.90993054705046461E-15
vector back   2.18922102668273055E-15
```

With rounding mode: IEEE_RM_ZERO

```
scalar frwd   -2.24533499432189032E-13
scalar back    2.74447131687338697E-13
vector frwd   -1.15359111152457672E-15
vector back    2.18922102668273055E-15
```

With rounding mode: IEEE_RM_NEG_INFINITY

```
scalar frwd   -2.24533499432189032E-13
scalar back   -2.24487095579206652E-13
vector frwd   -1.15359111152457672E-15
vector back   -1.60982338570647698E-15
```

While all of these numbers are close to zero, none of them is exactly zero, and more importantly for this paper, different values are calculated on the different machines. These numerical differences may be observed in real production codes, while porting a program from one machine to another. A programmer must determine if these differences a truly significant, and the result of a numerically unstable algorithm, or the insignificant result of different numerical format or different order of operations.

Numerical Differences - Order of Operations

One of the most common causes of numerical differences, is the compiler created order of operations for non-parenthesized arithmetic. For example:

```
C = 1.0 + A + C
```

if `A = -1.0` and `B = 1.0E-20`

then, depending on the order chosen by the compiler:

```
C = (1.0 + A) + C
C = (1.0-1.0) + 1.0E-20
C = 0.0 + 1.0E-20
C = 1.0E-20
```

or

```
C = 1.0 + (A + C)
C = 1.0 + (-1.0 + 1.0E-20)
C = 1.0 - 1.0
C = 0.0
```

The next step in your algorithm that uses "C" could do something different depending on its value, this is especially noticeable if it is used as a divisor.

Isolation of Differences - Step-by-step

There are two major steps in converting to the CRAY T90 computer with IEEE:

1. Convert from CF77 to CF90
2. Convert to the IEEE arithmetic

These two steps should be done independently. The compiler change can bring out many unsuspected numerical differences, and isolating and fixing these first will help immensely toward the second conversion to the IEEE hardware.

Isolation of Differences - CF77 to CF90

On the Cray format system, the CF90 compiler has a "-t *n*" switch, this switch is used to truncate the last *n* bits of precision after all floating-point calculations. For example: `f90 -t4` *file.f* compiles truncating the last 4 bits of all calculations. If you run this version of the program and get significantly different answers, you have a numerical differences problem to isolate.

One technique to narrow down where these differences occur, is to step through the two programs side-by-side with two TotalView windows, looking for differences. These differences could be either the values of key variables, or checksums of important arrays.

If you are really desperate, you could use `atchop` comparing a normal compile, to a compile with "-t5",to help narrow down which subroutine is causing the numerical differences.

File and Data Conversion

Automatic data conversion is available for files which need to be transported to and from IEEE from other Cray PVP systems. When your program is being run on a non-IEEE Cray PVP platform, and you wish to read or write a file compatible with the CRAY T90 series system with IEEE, use an `assign` statement like:

```
assign -N ieee_64 u:u
```

When your program is being run on CRAY T90 series system with IEEE, and you want to read or write a file compatible with a traditional non-IEEE Cray PVP system, use an assign statement like:

```
assign -N cray u:u
```

Also explicit conversions are done using library functions to convert data internally, use library subroutines `cry2cri` and `cri2cry` for 64-bit data; and `ieg2cri` and `cri2ieg` for 32-bit data.

What does not work with the `assign` implicit data conversion

Implicit data conversion works on the principal that the type of the data being read or written is determined by the type of the data on the `READ` or `WRITE` statement. If a program is using `EQUIVALENCE` or has created a "container" array that holds more that one data type, the implicit data conversion will not work correctly.

Type `LOGICAL`

The internal format for `LOGICAL` operands also is changing in the CRAY T90 series systems with IEEE:

```
CRI FP .TRUE.  : negative     (-1)
CRI FP .FALSE. : non-negative (0)
IEEE   .TRUE.  : non-zero     (1)
IEEE   .FALSE. : zero         (0)
```

Reference

For more information, see *Migrating to the CRAY T90 Series IEEE Floating Point*, publication SN-2194.
