NaN payload propagation - unresolved issues

by Agner Fog 2018-04-11

Table of Contents

Introduction

The IEEE 754 floating point standard includes specification of not-a-number (NaN) representations. A NaN can contain payload bits with arbitrary information. An operation with a NaN input should produce a NaN with the same payload as output.

NaN payloads may be used for a variety of purposes. However, the propagation of payloads is uncertain and not fully standardized. Many potential uses of payload propagation are hampered by this uncertainty.

Problems with NaN payloads and propagation have been discussed at length by the standardization working group, but several problems remain unsolved. This document summarizes some of these discussions including background information and proposed solutions.

Hardware support for NaN payload propagation

The standard specifies that an operation with a quiet NaN input should produce a quiet

NaN result with the same payload. Most microprocessors obey this recommendation though the behavior is not always documented. The following information has been collected:

- PowerPC processors support payload propagation according to the manuals.
- Propagation of NaN payloads can be turned on or off on ARM processors.
- Support for payload propagation is optional on RISC-V processors.
- Payload propagation has been verified experimentally for Intel, AMD, and IBM processors.

Known and possible uses of NaN payloads

Hardware generated error codes

Invalid operations such as 0/0, INF-INF, and sqrt(-1) produce a quiet NaN. Most microprocessors set the payload to zero in these cases.

It may be useful to generate different payloads to distinguish between different kinds of numerical faults. For example, one might want to distinguish between operations that have no meaningful result, such as 0/0, and operations such as sqrt(-1) that would produce a valid result if complex numbers could be represented.

The Mill architecture has an experimental feature that puts information about the place where a fault occurred into the payload.

It has been suggested that it might be useful to put sequential numbers into NaN payloads in order to trace the first of a series of faults. These sequence numbers may be slightly inaccurate on out-of-order processors.

Software generated error codes

Library functions may return a NaN in case of error with a payload indicating the kind of error. For example, the C++ vector class library (VCL) returns a user-defined payload in case of error situations such as log(-1).

Uninitialized data

Many software programs put NaNs into memory variables in order to signal an error in case the variable is used without being initialized first. This may be a signalling NaN or it may be a quiet NaN containing a payload in order to trace the error.

Missing data values

Most statistical software packages have a feature for indicating missing values in a data set, and NaNs are often used for this purpose. It is useful to distinguish between NaNs that originate from numerical faults and NaNs that indicate missing data. For example the 'R' statistical programming platform uses NaNs with the arbitrary payload value of 1954 to indicate missing data, while numercal faults are indicated by a NaN with zero payload.[1] A

reliable propagation of payloads is necessary for preserving this distinction.

Non-numeric data

Programming languages with dynamic typing of variables need a way of representing non-numeric data such as text strings. Some implementations use NaNs for this. The payload may contain a pointer or index to a text string or some other non-numeric object. This technique is called NaN-boxing.

For example, some implementations of JavaScript use the lower 32 bits of a double precision signalling NaN for a string pointer or other data while the next 3 bits indicate the type of data. Bits d1 - d17 are zero.

Detection of floating point faults

There are at least three commonly used ways of detecting floating point faults:

1. Status flags. A flag is set when an invalid operation occurs. The flag remains set until it is explicitly cleared.

2. Fault trapping. A trap occurs when an invalid operation is encountered. Execution is interrupted and an interrupt handler routine is called.

3. Propagation of NaN and INF results. Variables are tested for NaN and INF in the final result or at useful points in the algorithm.

All of these methods have drawbacks. A problem with status flags is that the scope of the flag may be too wide. The value of the flag is usually global to the thread in which the fault occurs. The information stored in this flag is therefore rather unspecific and you need to check it often in order to get useful diagnostic information.

Fault trapping is certain to catch an error when it occurs, but it can cause inconsistent results in case of parallel processing. Modern CPUs make increasing use of vector processing and the single-instruction-multiple-data (SIMD) principle. The size of vector registers has been growing exponentially in recent years. Let's explain the problem with an example. Assume that a program is looping through an array of sixteen floating point numbers, and the last number generates a fault. A sequential program will raise a trap in the last iteration of the loop. If an optimizing compiler packs all the sixteen values into a single 512-bit vector register and processes them together, using SIMD instructions, then the trap will be generated at the beginning of the calculations. If the loop has any side effects, for example writing something to a file, then the result will be different when the code is vectorized. Different CPUs may have different vector register sizes and thus generate different results for the same data. A further complication is that a single SIMD instruction can generate multiple faults, but you get only one trap, even if the faults are of different kinds.

Superscalar processors using out-of-order processing become less efficient when the program relies on status flags or fault trapping. The processor must have several temporary status flags when it is executing multiple instructions simultaneously and out of order. These flags are later combined into the global flag. The processor must serialize all execution whenever the code needs to read the value of a flag.

Fault trapping makes it necessary to rely on speculative execution. The result of an instruction cannot be stored permanently until it is certain that no preceding instruction generates a trap. Instead, the processor must preserve all information necessary to roll back the instructions in case of a trap. Current microprocessors are capable of handling these complications, but the cost is higher power consumption and lower performance.

All these problems can be avoided by using NaN propagation instead of fault trapping or status flags. But NaN propagation has other problems, which are discussed below.

Needs for propagation of payloads

Error codes and codes for missing data are cases where the propagation of NaN payloads is useful. The programs become simpler and more efficient if you can check for NaNs in the final result rather than after every operation that might generate a NaN result.

NaN boxing of strings and other non-numeric data do not depend on NaN propagation because these data should never occur as inputs to numerical calculations.

Obstacles to reliable propagation of error information

Propagation of infinity

Overflow and division by zero give infinity as result. The value of infinity will propagate through most operations. Infinity is converted to NaN in cases like 0*INF and INF-INF, and this NaN will propagate to indicate that an error has occurred.

There are a few cases where INF does not propagate, such as: 1/INF = 0, and min(1,INF) = 1.

Conversion to integer or Boolean

A NaN cannot survive a conversion to integer or Boolean values. The programmer must check for NaNs before such conversions or enable a trap for this situation.

Optimization by the compiler

A compiler may optimize away expressions such as x*0 or x-x if aggressive floating point optimization is enabled. The programmer should consider whether the loss of NaN propagation in these cases is permissible.

Pow function

The pow function fails to propagate INF and NaN in the following cases:
pow(0,INF) = 0, pow(1,INF) = 1, pow(1,nan) = 1, pow(nan,0) = 1.

These problems are fixed with the powr function defined in the 2008 revision of the standard. However, powr is not a universal replacement for pow because powr(x,y) does not allow negative x, even when y is an integral number.

Maximum and minimum functions

The maxNum and minNum functions defined in the 2008 standard propagate a non-NaN when one input is NaN and the other input is a normal number.

This problem will be fixed by the forthcoming revision of the standard. The new functions named maximum and minimum are certain to propagate NaNs.

Some current implementations are deviating from both of these definitions. Max and min instructions in the x86 instruction set are implemented so that max(a,b) and min(a,b) give b if one of the inputs is NaN. This is useful because it corresponds to the behavior of the code expression a > b ? a : b. A compiler can translate this common high-level language expression into a single instruction.[2]

Future microprocessors may implement new instructions to match the new definitions of maximum and minimum, while the existing instructions may be preserved because they match a common high-level language expression.

What should happen when two payloads are combined?`

The current standard specifies that if an operation has multiple NaN inputs, then the result should be one of the input NaNs. The standard does not specify which one.

Most microprocessors obey this rule, but the exact behavior is rarely documented. The following behaviors have been observed experimentally for NaN1 + NaN2:

- Intel using x87 instructions: NaN2 if both quiet, NaN1 if NaN2 is signalling
- Intel using SSE instructions: NaN1
- AMD using x87 instructions: NaN2
- AMD using SSE instructions: NaN1
- IBM Power PC: NaN1
- IBM Z mainframe: NaN1 if both quiet, presedence to signalling NaN
- ARM: NaN1 if both quiet, presedence to signalling NaN

This confusing behavior is quite unfortunate because it can lead to unpredictable results. Different compilers can put the operands in different order so that A+B may be coded as B+A. If A and B are NaNs with different payloads then the result may be either A or B. The same code may produce different results with different compilers, different microprocessors, different instruction set extensions, or different optimization settings. This is a serious obstacle to reliable NaN payload propagation.

Several possible solutions to this problem have been discussed. The diffent possibilities for combining NaN payloads are summarized below.

1. Make a NaN with a fixed payload value when two NaNs are combined. This makes the instruction commutative and predictable, but the information stored in the payloads is lost.

2. Make a bitwise OR combination of the two payloads. This will preserve information from both payloads, but it will only be useful if error codes are limited to a single bit for each possible error condition.

3. Propagate the highest of the payloads. This makes it possible to store more

complex information in NaN payloads, such as the place and time where an error occurred. This solution also makes it possible to prioritize error codes by giving the most serious errors the highest code values. The error code with lower priority is lost.

4. Propagate the lowest of the payloads. This will prioritize the first error if the payloads contain sequence numbers. A disadvantage of this solution is that empty payloads will be propagated. Current implementations have mainly empty payloads. The prioritization of early errors is better obtained by using a saturating down counter for sequence numbers and propagating the highest payload instead.

5. Generate a trap when two different NaNs are combined. This avoids the loss of information, but it defies the purpose of avoiding fault trapping by using NaN propagation instead.

6. Make a new NaN payload containing the address of the instruction that combined the two NaNs. It is then possible to rerun the code in a debugger with a breakpoint at the specified address in order to recover the two payloads. This procedure seems rather inconvenient and it fits only a very specific use case.

7. Software solutions. This is very inefficient because it requires a NaN check at every operation in a sequence of calculations.

Solution 3 (propagate the highest payload) seems to be the best solution, but no decision has been made yet.

Any future solution should solve the problem of reproducibility for commutative operations such as addition, multiplication, maximum, and minimum. It will be natural to apply the same solution to non-commutative operations such as subtraction and division.

What happens to the payload when the precision is converted?

The 2008 standard specifies that "Conversion of a quiet NaN from a narrower format to a wider format in the same radix, and then back to the same narrower format, should not change the quiet NaN payload in any way except to make it canonical." The details of conversion are not specified.

There is confusion over whether conversions between different precisions preserve the upper bits or the lower bits of the payload. These details are rarely documented, but experiments show that the behavior is as follows on all the platforms that have been investigated:

Conversion of a NaN in a binary format to a binary format with a different precision preserves the upper bits of the payload. Narrowing conversions cut off the lower bits. Widening conversions append zeroes.

Conversion of a NaN in a decimal format to a decimal format with a different precision perserves the lower bits of the payload.

Conversion between binary and decimal formats reverse the order of the payload bits on IBM processors.

This difference between binary and decimal representations reflects the way the mantissa bits are treated in normal floating point numbers. The mantissa field represents a fraction in the binary floating point format but an integer in the decimal floating point format. The current behaviors are the cheapest to implement in hardware because the mantissa bits are treated in the same way regardless of whether the value is a normal number or a NaN.

Identifying payload values across precisions

The fact that binary floating point formats preserve the upper bits rather than the lower bits during precision conversions may be a source of confusion when payload values are defined. A payload value of 1 (as an integer) in single precision will be equivalent to 2^{29} in double precision. A payload value of 1 (integer) in double precision will be lost when converting to single precision.

This problem could possibly be solved by preserving the lower bits rather than the upper bits of NaN payloads during precision conversion, but this solution would have extra hardware costs and we would need a switch for compatibility with existing hardware and software.

The 2008 standard does not explicitly address this problem for the binary format, but one may interpret the document as implicitly favoring the upper bits by numbering the payload bits from the upper end as d1, d2, d3, ... and by preserving d1 as the quiet bit across precisions. This has also beeen discussed back in 2011.[3]

The payload may be interpreted in different ways:

- as an integer
- as a left-justified integer
- as a fraction
- as a bit string. The 2008 standard does this
- as a bit-reversed integer. IBM does this

An integer interpretation of payloads in binary floating point formats makes it problematic to represent equivalent payloads in different precisions. We need a way of defining payload values that gives equivalent values for different precisions. A bit-reversed integer may be problematic if we choose to let the combination of two NaNs propagate the highest payload.

A draft of the forthcoming revision of the standard includes functions for getting and setting NaN payloads. It is a problem that these functions use integer representation of payload values without warning that the values may change when converting to a different precision.

The documentation should either make an explicit warning that the values may be shifted left when converted to higher precision and right when converted to lower precision, or the payload should be represented in another way than as an integer.

Preventing different applications from using the same payload for different purposes

It would have quite serious consequences if the same payload values were used for error codes and NaN boxes. A numerical error might generate a code that gets misinterpreted

as a NaN boxed string. It would therefore be useful to list known uses of NaN payloads and the values that they use. We might also reserve certain ranges of payload values for different purposes before it is too late.

Only very little information about payload values in use has so far been found. One known application of NaN boxes uses double precision signalling NaNs where only the lower 35 bits are used and the remaining bits are zero.

Conclusion

NaN payload propagation has many useful applications. We may expect to see more uses of NaN propagation in the future as the increasing use of parallel processing makes NaN propagation an efficient alternative to status flags and fault trapping.

The 2008 standard leaves room for different implementations of NaN payload propagation. The behavior of existing hardware and software is poorly documented and it would be useful to make more precise recommendations.

Three unresolved issues are particularly problematic and need further discussion:

1. What should happen when two NaNs with different payloads are combined?

2. How to define NaN payloads for binary floating point formats in a way that is equivalent for different precisions.

3. Identify intervals of payload values that are used for particular purposes and avoid using the same payload values for other purposes.

Problem 1 requires a hardware solution, while those who experience the need for a solution are mostly software programmers. However, hardware architects may also have an interest in promoting the use of NaN propagation as an alternative to status flags and fault trapping because it avoids certain performance problems.

The working group has so far not decided on any solutions to these issues because current uses of NaN payload propagation are few and poorly documented, and because it is difficult to predict future needs.

These issues will probably remain open for several years to come. Software programmers who rely on NaN payload propagation need to be aware of the problems that these issues give rise to.

1 https://github.com/hadley/r-internals/blob/master/vectors.md
2 Intel Architecture Software Developer's Manual (1999 and all later versions).
3 http://grouper.ieee.org/groups/754/email/msg04182.html