Floating point exception tracking and NAN propagation

By Agner Fog. Technical University of Denmark.

Abstract

The most common methods for detecting floating point errors are based on exception trapping or a global status register. These methods are inefficient in modern systems that use out-of-order parallelism and single-instruction-multiple-data (SIMD) parallelism for improving performance. It is argued that a method based on NAN propagation is more efficient and deterministic. Problems with NAN propagation in current systems are discussed. Examples of implementation in the C++ vector class library and in an experimental instruction set named ForwardCom are presented. The IEEE-754 standard for floating point arithmetic may need adjustment to accommodate the needs of modern forms of parallelism.

Contents

1 Introduction

The most common ways of detecting numerical errors during floating point calculations are:

1. Exception handling through the trapping of exceptions
2. Reading a status register
3. Reading a global variable called errno
4. Checking results for infinity (INF) and not-a-number (NAN)

These methods are all causing problems in modern computer systems. The first three methods, in particular, are not recognizing the needs of modern computers that rely heavily on parallelism for optimizing performance.

Today's state-of-the-art CPUs are using three kinds of parallelism in order to improve performance:

- Thread level parallelism. The computation tasks are divided between multiple threads. Each thread may run in a separate CPU core independent of each other.

- Out-of-order parallelism. Whenever a CPU is delaying the execution of an instruction because the input operands are not ready yet, the CPU will try to find other instructions further up the stream that are independent of the first instruction so that they can be executed in the meantime. Current CPUs are capable of reordering hundreds of instructions in this way and execute up to four or five instructions simultaneously in each clock cycle.

- SIMD parallelism. SIMD means Single Instruction Multiple Data. For example, a CPU can have a set of 512-bit vector registers containing 16 floating point numbers of 32 bits each. It is possible to add or multiply two such vector registers in a single instruction to get a new vector of 16 floating point results. The advantage is that you can do 16 operations simultaneously in a single instruction with a throughput of one, or even two, such vector instructions per clock cycle.

The current systems of error detection were designed before these methods of parallel execution were common. There is a big problem when calculations are done in parallel or out of order while the error detection systems are based on sequential logic without regard for parallel execution. Optimizations of both hardware and software are hampered by this discrepancy.

Software written in a high-level language often depends on sequential algorithms. The compiler may try to introduce SIMD parallelism and the CPU may add a further level of out-of-order parallelism. All this parallelism has to be undone in case you want to detect where an error occurred.

Thread-level parallelism is generally not a problem here because each thread can run in a separate CPU core with its own trap handler and its own status register. But out-of-order parallelism and SIMD parallelism are executed within the same CPU core with just one trap handler and one status register. A more efficient and deterministic solution requires that the error detection mechanism shares the same parallelism as the instructions causing the errors. These problems and possible solutions are discussed in this document.

2 Exception trapping

Microprocessors with hardware support for floating point calculations have a feature for raising exceptions in the form of traps (software interrupts) in case of numerical errors such as overflow, division by zero, and other illegal operations. Exception trapping can be enabled or disabled by setting a control word. The x86 platform has two such control words, the x87 FPU Control Word for the old x87 style instructions and the MXCSR register for the newer SSE and AVX instructions.

Fault trapping has the following advantages:

- It is possible to detect an error in a try-catch block.
- A debugger can show exactly where the error occurred.
- It is possible to get diagnostic information because the values of all variables at the time of an error are available.
- It is possible to design the software so that it can recover from an error.

The disadvantages of fault trapping are:

- Fault trapping is complicated and time consuming. It will slow down program execution if it happens often.
- A trap without a try-catch block will cause the program to crash with an annoying error message that is difficult to understand for the end user.
- Traps are problematic in layered software design with different authors at each level.

- Out-of-order processing is difficult to implement in hardware when traps are possible, because traps are supposed to occur in order. Instructions have to be executed speculatively until all preceding instructions have been executed and retired. The speculative results have to be discarded or rolled back in case an instruction that comes earlier in the original instruction order is causing a trap. The necessary bookkeeping for speculative execution requires extra hardware resources.
- Current CPUs will only make a single trap in case a SIMD instruction generates multiple exceptions, even if the exceptions are of different kinds. The number of exceptions that are detected may therefore depend on whether SIMD parallelism is used and on the size of the vector registers.
- SIMD code is typically handling branches by executing both sides of a branch with the whole vector and then combining the two vector results by picking each element from one side or the other depending on a boolean vector representing the branch condition for each element. This has the consequence that a not-taken branch can cause a spurious trap. Current compilers are unable to generate SIMD code for loops that contain branches for this reason when traps are enabled.
- A compiler cannot optimize variables across the boundaries of a try-catch block.

3 Using a status register

Most CPUs have one set of status bits per thread. Reading or writing a status register is incompatible with out-of-order processing. Out-of-order processing is suspended when an instruction to read the status register is encountered. The instruction has to wait for all preceding floating point instructions to retire before a valid value for the status register is available.

The out-of-order scheduling of instructions becomes more complicated the more input and output dependencies each instruction has. All floating point arithmetic instructions are able to write to the status register. This has high costs in terms of hardware complexity.

4 Using errno

A global variable named errno was introduced early in the history of the C language before multithreading became common. This variable was replaced by a reference to thread-local variables when threads were introduced. The errno pseudo-variable contains a code number indicating the type of the last error. This includes all types of errors, for example file errors, and also floating point errors. The errno variable can only indicate a single error and it contains very little information about the error.

Implementations of errno rely mostly on exception trapping with the same disadvantages as listed above. Current compilers cannot produce SIMD instructions for branching code that contains functions that may set errno (e.g. sqrt) unless the errno feature is disabled.

5 Parallel error detection

The obvious solution to these problems is to make a system of error detection that follows the same parallelism as the instructions that generate the exceptions. For SIMD parallelism, we may replace the status register with a vector with one set of status flags for each vector element. However, this does not solve the problem with out-of-order parallelism as long as we have only one status vector. We need to have status flags that are somehow tied to the output of each instruction. We can think of several possible solutions:

1. Each instruction could have two output registers: a result and a status. This would make the code quite complicated, and each instruction would need extra bits to indicate which status register to use. The out-of-order mechanism needs to keep track of the extra dependencies for the status registers.

2. All floating point registers and vector registers could have extra bits to indicate exceptions. The exception bits can be propagated from input registers to output registers, and ORed with any exceptions generated by each instruction. This may cause problems when a result needs to be saved to memory because variable definitions in high-level languages have no such extra bits. This method does not give information about where an exception occurred.

3. The code can rely on the generation of INF and NAN results in case of floating point errors. These INF and NAN codes are propagated to the end result, except in cases discussed on page 5. This method cannot detect underflow and inexact exceptions.

4. The hardware may be designed so that enabled exceptions generate NAN results. The code for a NAN contains a number of bits called a payload that can contain extra information. The hardware should generate a NAN with a payload containing information about the type of exception and possibly also the code address where it occurred. Each type of exception can be enabled or disabled. For example, overflow will generate NAN when the overflow exception is enabled, and INF when the overflow exception is disabled. The NAN values can be detected at strategic places in the code as long as we can rely on the NAN values propagating to these places. The payloads are preserved in the propagating NAN values. Disadvantages of this method are that the exception information is lost in some situations such as conversion to integer. A programmer may enable exceptions in one part of the code and forget to disable them again, thereby causing errors in a different part of the code.

Examples of solution 3 and 4 are discussed on page 8 below.

6 Propagation of INF and NAN

The standard representation of floating point numbers includes codes for infinity (INF) and not-a-number (NAN). These codes are used for representing invalid results.

Infinity is coded in the following way. All the bits in the exponent field of the floating point representation are set to 1 and the significand (also called mantissa) bits are set to 0. The sign bit indicates the sign of infinity. A NAN is coded in the same way as infinity, but with at least one significand bit set to 1. There are two kinds of NANs. A *quiet* NAN has the most significant bit of the significand set to 1. A *signaling* NAN has this bit cleared and at least one other significand bit set. Propagated NANs are always quiet NANs. The significand bits can contain arbitrary extra information called a payload.[1]

The result of a calculation will be ±INF in case of overflow or division by zero. The result will be zero in case of underflow. The result will be NAN in the following cases:

- 0/0
- INF-INF
- INF/INF
- 0*INF
- An argument of a function is out of range, e.g. sqrt(-1) or log(-1).

The output of an operation will be INF or NAN if an input is INF or NAN, with a few exceptions listed below. This is useful because an INF or NAN will propagate to the end result if an error occurs in a series of calculations. We do not have to check for errors after each step in a series of calculations if we can be sure that any error will show up in the end result as INF or NAN.

The advantages of relying on the propagation of INF and NAN are:

- The performance is excellent because no resources are spent on error checking, except for the final result. Current hardware does not take extra time to process INF and NAN inputs.
- The code can be converted to SIMD without any change in the behavior.
- Extra information contained in a NAN payload will propagate to the end result in most cases. The payload can be used for information about an error. For example, library functions can return an error code in the payload of a NAN result.

The disadvantages are:

- An INF result will be converted to a NAN in cases like INF-INF, INF/INF, and 0*INF.
- INF and NAN results are not propagated in certain situations, listed below.
- All comparison operations involving a NAN will be evaluated as false. The programmer needs to take this into account and decide which way a branch should go in case of a NAN input. See page 6.
- When two NANs with different payloads are combined, only one of the payloads is propagated.

6.1 Operations that fail to propagate INF and NAN inputs

Dividing by infinity
Dividing a finite number by INF gives zero.

Min and max functions
The IEEE 754-2008 standard defines the functions minNum and maxNum giving the minimum and maximum of two inputs, respectively. These functions do not give a NAN output if one of the inputs is NAN and the other is not a NAN.[1]

A revision of the IEEE 754 standard in 2019 defines two additional functions, named minimum and maximum, that do the same but with propagation of NAN inputs.[2]

The actual implementation of min and max in many CPUs differs from both of these standards. A common way of defining min and max in a high-level language is:

min(a, b) = a < b ? a : b, max(a, b) = a > b ? a : b.

As comparisons involving a NAN return false, we have:

min(NAN,1) = 1, min(1, NAN) = NAN, max(NAN,1) = 1, max(1, NAN) = NAN.

The min and max instructions in the SSE and later x86 instruction set extensions are returning the second operand if one of the operands is NAN, in order to fit the high level language expression.

It is necessary to make software implementations of min and max with guaranteed propagation of NAN inputs as long as no hardware implementation of these functions is available.

Power functions

The pow function fails to propagate a NAN in the special cases pow(NAN,0) = 1 and pow(1,NAN) = 1. The IEEE 754 standard actually defines two additional versions of the power function. The function powr is defined as powr(x,y) = exp(y*log(x)). The powr function is certain to propagate NANs, and it makes no special case for integer y. Another function, pown, is defined only for integer y.[1]

Few function libraries are implementing the powr function, and few programmers have found it useful. It is inconvenient for programmers to use two different functions depending on whether the power is known to be an integer or not. Even if it is seldom used, the powr function is important for indicating that the definition of pow is not a mathematical truth, but an arbitrary convention that has been settled for historical reasons. This makes it more legitimate to deviate from the standard when a different behavior is needed.

The three functions differ in the following ways, according to the IEEE 754-2019 standard.[2]

x, y	pow(x,y)	powr(x,y)	pown(x,y)
0, 0	1	NAN	1
NAN, 0	1	NAN	1
1, NAN	1	NAN	not possible
negative, integer	x^y	NAN	x^y
negative, non-int	NAN	NAN	not possible
INF, 0	1	NAN	1
1, INF	1	NAN	not possible

Other functions
exp(-INF) = 0.
atan(± INF) = ± $\pi/2$. atan2(± INF, ± INF) = ± $\pi/4$.
tanh(± INF) = ± 1.
hypot(NAN, INF) = hypot(INF, NAN) = INF.
compound(INF,0) = compound(NAN,0) = 1

6.2 Comparison operations involving NAN

All comparisons with a NAN input will be evaluated as 'unordered'. The operators `>`, `>=`, `==`, `<`, `<=` will all evaluate as false if one or both operands are NAN. The `!=` operator evaluates as true if one or both operands are NAN. Comparison of a NAN with itself will be false:

```
float a;
if (a == a) {
   // false if a is NAN. True in all other cases
}

if (a != a) {
   // goes here only if a is NAN
}
```

The programmer may want to decide which way a branch should go in case of a NAN input. If you want a comparison to be evaluated as true in case of NAN inputs, you may negate the opposite condition. This is illustrated in the following two examples.

```
float a, b;
if (a > b) {
   do_if_bigger();
}
else {
   do_if_less_than_or_equal();    // goes here if a or b is NAN
}
```

The next example will do the same, except for NAN inputs:

```
float a, b;
if (!(a <= b)) {
   do_if_bigger ();              // goes here if a or b is NAN
}
else {
   do_if_less_than_or_equal();
}
```

There is no performance cost to negating a condition because most modern processors have hardware instructions for all cases of comparisons including negated comparisons and 'unordered' comparisons which evaluate as true for NAN inputs.

In addition to the 'ordered' and 'unordered' versions of all compare instructions, there are also 'signaling' and 'quiet' versions of the compare instructions. The signaling versions will generate a trap if an input is NAN, while the quiet versions will behave as explained above.[1] The signaling versions are needed only if the code has no branch that adequately handles the NAN cases.

7 Diagnostic information in NAN payloads

The single precision floating point format has 22 payload bit in quiet NANs, double precision has 51 bits of payload.

The payload is preserved when a NAN is propagated through a series of calculations. This makes it possible to propagate diagnostic information about an error to the end result.

Current microprocessors make a zero payload when a NAN is generated by hardware, for example in operations such as 0/0. Non-zero payloads can be generated by software. This may be useful for function libraries that put an error code into the NAN result when parameters are out of range.

NAN payloads are also used for non-numeric information in a technique called NAN-boxing. Programming languages with weak typing, such as JavaScript, are storing text strings and other non-numeric variables as signaling NANs where the payload is a pointer or index to the string, etc.

Signaling NANs may be used for uninitialized variables. This makes it possible to detect variables that have been used without initialization. But signaling NANs are rarely used today, except for NAN boxing.

Some statistical software packages are using a NAN with a particular payload to indicate missing data. The R statistical platform is using the arbitrary payload value of 1954 to indicate missing data. Few other current uses of NAN propagation are known.

Unfortunately, the propagation of NAN payloads is not fully standardized. When two NANs with different payloads are combined, the result will be one of the two inputs, but the standard does not specify which one. This is discussed below on page 9.

Another problem with NAN payloads occurs when the value is converted from single to double precision, or vice versa. Experiments on various types of microprocessors show that the NAN payload is handled in the same way as the significand bits of normal floating point numbers where the most significant bits are preserved when the precision is reduced. This applies to all binary floating point formats, while the less common decimal formats treat the NAN payload, as well as the significand, as an integer where the least significant bits are preserved.[3] This behavior is undocumented, and future systems may behave differently.

8 Detecting integer overflow

The x86 and other common platforms allow trapping of integer division overflow, but cannot trap overflow in integer addition, subtraction, and multiplication. Integer variables have no codes for INF and NAN.

It is particularly difficult to detect overflow in signed integers because the C standard specifies signed integer overflow as 'undefined behavior' which means that the compiler can assume that overflow does not happen, and some compilers will even optimize away an explicit overflow check for this reason.

A system that can detect integer overflow efficiently would have to use extra register bits or status flags to trace and propagate the overflow information.

9 Example: Vector Class Library

The vector class library is a C++ class library for utilizing the SIMD capabilities of modern CPUs (https://github.com/vectorclass). The preferred method for detecting floating point errors in the vector class library is to rely on the propagation of INF and NAN values. Library functions such as log and pow are generating NANs with diagnostic payloads in case of errors. Errors that occur directly in hardware instructions, such as 0/0 = NAN, have no payloads because the hardware lacks support for generating such payloads. This is probably the best solution that can be achieved with existing hardware.

10 Example: ForwardCom instruction set

ForwardCom is an experimental instruction set and computer system designed for improving performance and for overcoming many of the problems of existing systems (https://www.forwardcom.info).

ForwardCom implements method 4 for detecting floating point exceptions, as described on page 4. Floating point exceptions can be enabled globally or per instruction. The output of a floating point instruction in case of an enabled exception is a NAN with a diagnostic payload. SIMD instructions will generate NANs only in the relevant vector elements. The lower 8 bits of the NAN payload indicate the type of exception. The remaining bits indicate the code address where the exception occurred.

Floating point overflow will produce a NAN rather than an INF if the overflow exception is enabled. This is useful because a NAN will propagate in cases where an INF does not propagate, e.g. as 1/INF = 0, but 1/NAN = NAN.

When two NANs are combined, e.g. NAN1 + NAN2, the one with the highest payload is propagated to the result. This makes it possible for a debugger to find the exception that came first in a sequence of instructions. The address bits in the payload are inverted so that the lowest addresses come first. The debugger may have to check for NANs before any backward jump in order to identify the first exception.

NAN payloads are right-justified rather than left-justified when a NAN is converted to a different precision. This secures the propagation of payload information through different precisions.

All floating point functions are guaranteed to propagate NANs, including max, min, pow, etc.

Integer overflow can be detected using method 2 described on page 4. This is implemented in special instructions that use the even-numbered elements in a vector register for integer arithmetic while the odd-numbered elements contain propagating error flags. These instructions are used only when the detection of integer overflow is desired.

The ForwardCom instruction set thus represents a hardware implementation of the best possible method for detecting numerical errors and exceptions discussed in the present document.

11 Does the IEEE 754 floating point standard need revision?

The problems with out-of-order parallelism and SIMD parallelism have been discussed in the working group for the IEEE-754 floating point standard, but no decision has been made for the 2019 revision of the standard. The discussion is continuing and hopefully a decision can be made for the next revision, planned for 2028.

Traps are already on the way out in most applications, while global status registers are widely used for detecting floating point exceptions.

The original purpose of signaling NANs was to generate traps. I have never encountered any use of this feature. The main use of signaling NANs today is a different one, namely NAN boxing of non-floating point data. If traps are out, we may discuss whether it is still relevant that signaling NANs must generate a trap and be converted to quiet NANs.

It is unclear whether the proposed use of NAN payloads instead of a status register is permitted by the present standard.

Propagation of NAN payloads is rarely used today and the standard is not very specific about this topic. When two NANs are combined, i.e. NAN1 + NAN2, one of the two payloads is propagated, but the standard does not specify which one. Current microprocessors propagate only the first NAN operand. This can lead to unpredictable behavior since the compiler is allowed to swap the two operands. We should prefer a more consistent and predictable behavior. Propagating the highest payload is certainly permitted by the standard.

Another uncertain point is what happens to the NAN payload when the precision is converted so that the number of payload bits is increased or decreased. Current CPUs with a binary floating point representation have the payload bits left-justified, while computers with decimal representation have the payload bits right-justified. This behavior is undocumented, and a different behavior is certainly permitted. The compatibility of NANs with different precisions can be improved by right-justifying the payload so that the least significant bits (and the quiet bit) are preserved when the precision is changed.

The problems with NAN propagation through the max and min functions has been fixed in the 2019 revision of the standard, while problems remain with the pow, hypot, and compound functions.

12 Conclusion

The standard methods for detecting floating point exceptions were designed before the methods of out-of-order parallelism and SIMD parallelism became widely used. The use of exception trapping or a global status register are both based on sequential logic which does not fit well into a paradigm of parallel data processing. This discrepancy is hampering the optimization of both hardware and software, even in applications where errors or exceptions never occur.

Current CPUs are using a lot of resources on speculative execution and bookkeeping in case a trap requires that instructions that have been executed out-of-order needs to be purged or brought into order. Out-of-order execution is also suspended when reading a status register. The fact that all floating point arithmetic instructions are potentially writing to the status register places an extra burden on the out-of-order scheduling logic.

Modern compilers are able to convert code with sequential loops to SIMD code. But the compilers can do so only when exception trapping is disabled. SIMD code can become unreliable or inconsistent if the algorithm relies on reading a status register.

A more efficient and consistent system is possible if an error detection system is designed in a way that reflects the kinds of parallelism used. Any error information should follow the individual result. This can be obtained by using NAN propagation to trace floating point errors. The propagating NANs may contain payloads with diagnostic information about the errors. Such an improved system would have no traps for floating point exceptions and no global status register.

Current CPUs are able to generate NANs in case of invalid operations such as 0/0 or sqrt(-1), but not in case of overflow and underflow. Overflow generates the code for INF, which will propagate in many cases, while underflow generates a zero. It is possible to utilize the propagation of NAN and INF for detecting floating point errors with currently available hardware and compilers. The programmer needs to be aware of situations where NANs and INF values are not propagating to the end result.

A better performance can be obtained with a redesigned hardware that will generate NANs in case of enabled floating point exceptions and insert diagnostic payloads in these NANs. An experimental instruction set named ForwardCom is implementing this method.

The propagation of NAN payloads is seldom used today and the behavior is not fully standardized. A better standardization is recommended for the situations of combining two NANs, converting the precision, and certain mathematical functions that are not guaranteed to propagate NANs.

The IEEE-754 standard may need revision to better support the use of NAN payloads instead of a status register.

13 Notes

1. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008.

2. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019.

3. Agner Fog. NaN payload propagation - unresolved issues. 2018. http://754r.ucbtest.org/background/nan-propagation.pdf