Parallel floating point exception tracking and NaN propagation

By Agner Fog. © 2019-2026 CC-BY. Last updated 2026-05-03.

Abstract

The most common methods for detecting errors and exceptions in floating point computing are based on either exception trapping or a global status register. Both methods are relying on sequential logic. These methods are inefficient in modern systems that use out-of-order execution and single-instruction-multiple-data (SIMD) parallelism for improving performance. It would be more efficient to implement a system that attaches the status after an exception to the result register and let this information follow the same data path and the same form of parallelism as the normal data flow. This can be achieved either by attaching metadata tags to the result or by generating NaN results containing diagnostic payloads. These metadata tags or NaN payloads can propagate through subsequent calculations. Such a system would be more efficient than traps and global status flags.

Advantages and disadvantages of each solution are discussed, and examples of experimental implementations are presented here.

Contents

1 Introduction

Modern computing systems are using parallel data processing to an increasing degree in order to maximize performance. Massive parallelism is increasingly important in big data processing, vector processing, graphics processing, sound processing, video games, artificial intelligence, and machine learning.

Current methods for detecting numerical errors or exceptions are not attuned to parallel processing because they rely on sequential logic. The IEEE 754 floating point standard specifies two possible ways of detecting numerical errors or exceptions during floating point calculations:

1. Exception handling through the trapping of exceptions
2. Setting a global status register

These methods are all designed for sequential execution, dating back to a time when computers were mostly executing instructions one by one. This is not optimal in modern computers that rely heavily on parallelism for optimizing performance.

Today's state-of-the-art CPUs are using mainly 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.

- SIMD parallelism using vector registers. The advantage is that it can do multiple operations simultaneously with a single instruction.

- Out-of-order parallelism. Multiple instructions can be executed simultaneously if they are independent. Those instructions where the input operands are available first may be executed first, regardless of their order in the code.

The responses to exceptions are required to happen in program order for the sake of reproducibility. This is a big problem when calculations are done in parallel or out of order. Optimizations of both hardware and software are hampered by the discrepancy between parallel data processing and sequential exception handling.

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.

Software written in a high-level language often depends on sequential algorithms. The compiler may try to convert serial calculations to parallel using SIMD instructions. The CPU may add a further level of out-of-order parallelism. All this parallelism has to be undone in case an exception is detected, because subsequent instructions after the exception event need to wait for the event to be handled. Otherwise, the result would not be guaranteed to be reproducible. Current systems cannot adequately handle the situation if a SIMD instruction causes multiple exceptions.

Graphics processing units (GPU) are used increasingly nowadays for parallel data processing, including graphics applications and artificial intelligence. Current GPUs are not supporting the exception handling systems specified by the IEEE 754 standard because an exception would cause an unacceptable interruption of the parallel control flow. Instead, GPUs will usually generate infinity or NaN in case of an exception.

The most logical solution to these problems is to design an exception detection mechanism that shares the same parallelism as the instructions causing the exception. The status information, including information about possible exceptions or errors, should follow the same data paths as normal calculation results.

These problems and possible solutions are discussed in this document.

2 Terminology

Default value
This is the value that is placed in the result register in case an operation has an exception, but the exception is disabled. The default value produced by overflow or division by zero is ± infinity. The default value produced by underflow is ± zero or a subnormal number. The default value produced in case of inexact is the rounded result.

Exception
An exception is a technical term for a software event that requires special handling. The IEEE 754 standard defines five types of exceptions: invalid operation, division by zero, overflow, underflow, and inexact. An exception is usually considered an error, but the term 'error' is broader, including for example missing data.

GPU
Graphics processing unit. This is an extra processing unit that is used for parallel data processing in graphics and other applications.

IEEE 754
This is an international standard for floating point computing. It defines how floating point numbers are represented in computers and standardizes the operations and functions that can be applied.

NaN
Not a Number. The standard format for floating point data includes a code for 'not a number'. This is used to represent the result on an invalid operation, for example 0/0 or sqrt(-1).

Out-of-order execution
Whenever the execution of an instruction is delayed 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. Each instruction may be executed as soon as its input operands are ready. Top-level CPUs are capable of reordering hundreds of instructions in this way and execute up to four, five, or six instructions simultaneously in each clock cycle.

Payload
The floating point code for NaN includes vacant bits that can be used for diagnostic information or any other purpose. The value stored in these bits is called a payload.

SIMD
Single-Instruction-Multiple-Data. Some computers have advanced instructions that can operate on vector registers, containing multiple data values. For example, an SIMD 'add' instruction may add two vector registers, each containing eight floating point numbers, and return a result in the form of a new vector with the eight sums.

Status
An indication of whether exceptions have happened.

Trap
A trap is the same as a software interrupt. It is similar to a hardware interrupt, except that it is activated by a software event rather than a hardware event. A trap will stop the normal execution of instructions and transfer control to an interrupt handler routine that takes care of the event. When the interrupt handler is finished, it may return control to the normal code and continue execution where it left, or it may abort the program in case of an unrecoverable error.

3 Exception trapping

Microprocessors with hardware 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, etc. Exception trapping can be enabled or disabled by setting a global control word.

Exception trapping has the following advantages:

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

The disadvantages of exception trapping are:

- Exception 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.
- Stack unrolling is necessary in case a trap causes a function to exit prematurely. The software needs to recover all variables stored on the stack and call the destructors of all local objects. This functionality is complicated, and it is necessary to store all information needed for correct stack unrolling, even if the trap never actually occurs.
- Traps are problematic in layered software design with different authors at each level. Code at one level may encapsulate a particular functionality, but traps are likely to break out of the encapsulation and be caught at a higher level.
- Out-of-order processing is difficult to implement in hardware when traps are possible, because traps are required to occur in program order. All instructions have to be executed speculatively until all preceding instructions that could possibly cause traps 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 costs of this is increasing with modern out-of-order processors that can have hundreds of instructions in flight at the same time. The necessary bookkeeping for speculative execution requires extra hardware resources, even if the potential traps never occur.
- 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. The same code compiled with different vector register sizes may give different results.
- 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. Most 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.

4 Using a status register

Many programmers prefer to detect exceptions by reading a status register rather than catching exception traps. This is much simpler in terms of program logic, and often more efficient.

Most CPUs have one status register per thread so that it can handle thread-level parallelism, but current designs do not consider SIMD parallelism and out-of-order parallelism. If an exception occurs in a vector register then the program has to rerun the calculations in sequential mode if you want to know which vector element caused the exception. This requires so much extra code complexity that it is practically never done.

The problem with branches in SIMD code is the same as for exception trapping. An exception in a not-taken branch can signal a spurious exception in the status register. Most compilers are unable to work around this problem.

Reading or writing a status register is incompatible with out-of-order processing. Out-of-order processing is suspended every time the status register is being read. The read-status-register instruction has to flush the pipeline and wait for all preceding floating point instructions to retire before a valid value for the status register is available. This can be quite time-consuming when possibly hundreds of instructions are in flight in the pipeline at the same time.

The hardware for out-of-order scheduling of instructions becomes more complicated the more input and output dependencies each instruction has. A global status register adds an extra output dependency for all instructions that may generate an exception. This has high costs in terms of hardware complexity and power consumption.

4.1 Could a vector status register work?

Some have proposed a vector status register with one set of status bits for each vector element. This would facilitate error detection in SIMD code.

However, such a solution would have many of the same drawbacks as a single global status register. Reading the register will prevent out-of-order execution, and you can get spurious exceptions from non-taken branches. If you want to record the exception status of a particular task then you would need to clear the status register before the task and reading it after the task. You will have to run one task at a time. You cannot run multiple tasks concurrently because they would influence each other's status.

You will also need extra software instructions for connecting each status with each result whenever data are reorganized. This becomes even more complicated if the same program is compiled for different microprocessors with and without SIMD capabilities, or with different vector register sizes.

Another solution that has been proposed is to generate a separate status vector output from each SIMD instruction. This would be quite expensive to implement in hardware. It would also require a lot of extra software instructions for accumulating status vectors from a sequence of calculations and tracing each status through branches, loops, and reorganization of the data. In fact, the total number of instructions would be almost doubled when you need separate instructions for the flow of data and the flow of status. You would also need extra memory for storing both data and status of all intermediate and final results.

4.2 The old variable named 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 a thread-local variable 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 often rely 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 exception detection

The most logical solution to all these problems is to design a system of exception detection that follows the same parallelism as the instructions that generate the exceptions. The exception status information should preferably be attached to the individual result and follow the same data paths as normal calculation results. When an instruction receives an input with an exception status then it should attach the same status to its output so that the status information can propagate through a sequence of calculations to the final result.

A further advantage of parallel detection of exceptions is that it simplifies speculative execution after a branch prediction. The status information is automatically discarded together with the false result in case of a misprediction.

Parallel detection of exceptions can be achieved in various ways. The two most promising solutions are:

1. Attach a metadata tag to each result. All data registers in the CPU should have a few extra bits containing metadata about the status of the register value including any exception that may have occurred during the calculation of this value. The tag is set in case of an exception to indicate the type of exception. Every arithmetic operation should look at the metadata of all input operands and copy any status information to the output operand so that the status is propagated through all subsequent calculations.

2. Replace the normal result of an arithmetic operation with a NaN in case of an exception. The NaN should have a payload containing a status code indicating the type of exception and possibly any additional diagnostic information. Each type of exception can be enabled or disabled. For example, an operation that causes underflow will return a NaN with a diagnostic payload only if the underflow exception is enabled. It will just return zero or a subnormal number if the underflow exception is disabled. A mechanism for propagating NaNs through subsequent calculations already exists. The IEEE 754 standard specifies that an operation with a NaN input should generate a NaN output with the same payload.

Both of these solutions have advantages and disadvantages.

Advantages of the metadata tag method

- This method works with all data types, including integers and booleans.

- It preserves both the status and the default value. In case of the ‘inexact’ exception, we have the rounded result in the register and the information about whether it is exact in the metadata tag.

Disadvantages of the metadata tag method

- Requires extra bits in all hardware registers. This can be quite a lot of extra bit storage in case of vector registers holding many elements of a small data type.

- Requires extra functionality in current hardware designs to propagate metadata tags through all data operations.

- The number of metadata bits must be a compromise between the amount of diagnostic information we want and the cost of having more hardware bits. We will probably be able to store only the most basic status information because of the cost of extra hardware bits.

- The metadata tag is lost when storing a register value to memory. It is necessary to design a new method for storing both value and metadata tag, which conflicts with all standards and common practice for how data are stored. It also causes problems with data alignment and cache efficiency. Alternatively, we may generate a trap when a value with an exception tag is stored to memory, but this would defy the purpose of avoiding serial processing.

- Checking metadata tags from software requires system-specific intrinsic functions.

Advantages of the NaN propagation method

- Requires no extra storage.

- Register storage and memory storage contain the same information. The compiler can freely store data in registers or memory.

- It relies on an existing mechanism for NaN propagation that is supported by most microprocessors.

- Subsequent instructions will not generate further exceptions when receiving an input that is NaN.

- The payload has enough bits for storing detailed information about the exception type. Except for the lowest precision, there is also space for other diagnostic information, such as the code address where the exception occurred.

- The payload has enough bits for adding user-defined error codes. For example, a function library can add its own application-specific error codes.

- NaNs can be checked with normal software branches in normal program flow. Error handling is as simple as checking a result. There is no need for try-catch blocks, traps, or a global status register.

- SIMD branches can be handled efficiently by executing both sides of a branch with the whole vector, as explained above, and then picking each element from one side or the other depending on the branch condition for each element. Spurious exceptions are simply ignored because results from the not-taken branch are discarded. It is not necessary to use predication masks for suppressing exceptions in not-taken branches.

Disadvantages of the NaN propagation method

- This method does not work with integer data types.

- The default value is lost when a NaN is generated. If you want to check whether the result of a calculation is exact and you also want to know the rounded result, then you have to execute the same calculation twice, with and without the ‘inexact’ exception enabled. For the other exception types, the status code can contain enough information for recovering the default result.

- When two NaNs with different status codes are combined, only one is propagated.

- It may be necessary to check for NaNs at various places in the program code in order to treat exceptions gracefully.

- A few mathematical functions are not guaranteed to propagate NaNs. These are discussed below.

Flag bits versus status code
The IEEE 754 standard specifies five flag bits to indicate each of the five exception types. With five flag bits, we are able to detect if more than one exception has happened – but only if they are of different kinds. This information has limited value if we do not know where or how these exceptions occurred, we do not know which exception happened first, and we do not know whether the second exception is a consequence of the first one or an independent event.

We may prefer instead to define a status code for each type of error or exception. A single operation cannot generate more than one exception. We do not need the capability to signal more than one exception when the status information is localized. If a chain of instructions has a propagating error status then it need not generate additional exceptions.

With an n-bit error code, we will be able to distinguish between 2^n-1 different types of exceptions. If we have many bits available, we will be able to add additional diagnostic information, for example the code address where an exception happened.

The metadata tag method may give us enough bits for storing some diagnostic information, but the cost of storing extra bits is significant if implemented in hardware. The NaN propagation method gives us 22 vacant payload bits with single precision and 51 payload bits with double precision. These bits can be useful for all kinds of diagnostic information.

In the case that two NaNs with different payloads are combined, e.g. NaN1 + NaN2, we may propagate the one with the highest payload, which could indicate the most severe of the two exceptions. It is a disadvantage that we can preserve the information about only one exception. But if we want a system that can propagate information about multiple exceptions, then we have to return to the principle of one flag bit for each exception type. We would have to make the OR combination of the NaN payloads rather than selecting the highest payload. An OR operation would garble any additional diagnostic information.

6 Experimental implementations

The two ways of propagating exception information can be illustrated by two experimental CPU designs.

Metadata tag method
The metadata tag method is used in the Mill computer, that is under development by Mill Computing, Inc (millcomputing.com). Each data register and each vector element has a metadata tag. The tag can indicate the five exception types, as well as missing data. The tag is propagated through a sequence of instructions.

The Mill computer is capable of speculative memory reads. The data item is tagged as invalid in case a speculative read fails.

The Mill computer will produce a trap in case of attempts to store data with an error tag to memory. This may complicate parallel data storing and out-of-order processing.

NaN propagation method
The NaN propagation method proposed here is implemented in an experimental instruction set named ForwardCom (forwardcom.info). ForwardCom is an open source project where everything is specified and documented.

ForwardCom allows each type of exception to be enabled or disabled globally or locally. If an exception occurs, and it is enabled, then a NaN result is generated. This NaN has a payload containing the status code as well as additional diagnostic information.

7 Proposal for an exception propagation scheme

This is a proposal for how to encode exceptions in a propagating NaN payload. It is currently implemented in the ForwardCom instruction set for the purpose of testing it.

The status code in the payload has a more detailed distinction between exception types than defined by the IEEE 754 standard, and it is supplemented with additional exception types and error types. The status code is structured as a bitfield shown in this figure:

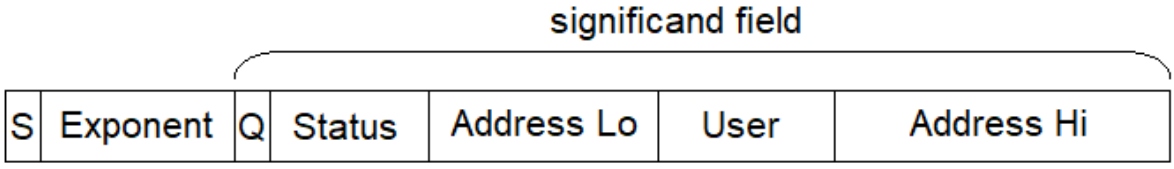

Figure 1. Layout of diagnostic payload in a NaN. S = sign bit. Exponent = all 1s. Q = quiet bit = 1. Status = status code. Address Lo = lower 13 bits of code address where exception happened. User = bits for arbitrary use. Address Hi = remaining bits of code address. The number of available fields and their size depend on the floating point precision.

The NaN payload is stored in the significand field, which contains the fraction in normal floating point numbers. The diagnostic payload is organized in a bitfield containing the quiet bit, status code, low and high part of the code address where the exception happened, and a field with optional user-defined diagnostic information. The availability and size of these fields depend on the precision of the floating point format according to table 1. Formats smaller than 16 bits can propagate a NaN indicating that an exception has happened, but it cannot contain any details about the type of exception.

Precision	Exponent	Status	Address Lo	User	Address Hi
bfloat16	8	6	0	0	0
float16	5	9	0	0	0
float32	8	9	13	0	0
float64	11	9	13	10	19
float128	15	9	13	10	≥ 19

Table 1. Number of bits of each field for status codes in different floating point formats. The S and Q fields always occupy only one bit.

Conversion between different floating point precisions is done by left-justifying the significand field. The fields of the NaN payload are organized according to table 1 in such a way that as much information as possible is preserved when a NaN is converted to a different precision. This applies to binary floating point formats only. Decimal floating point formats behave differently and are not covered by this proposal.

The proposed status codes are listed in table 2:

Error or exception	Status code	Default result	Control bit
data not initialized	111111 111	NaN	
data not available	111111 110	NaN	
data not accessible	111111 101	NaN	
data missing, any reason	111111 000	NaN	

user-defined high priority codes	111110 xxx		
division by zero, negative	111101 111	- INF	div Z
division by zero, positive	111101 110	+ INF	div Z
logarithm of zero	111101 011	- INF	div Z
division by zero (unknown sign)	111101 000	± INF	div Z
division overflow, negative	111100 111	- INF	overflow
division overflow, positive	111100 110	+ INF	overflow
multiplication overflow, negative	111100 101	- INF	overflow
multiplication overflow, positive	111100 100	+ INF	overflow
add/sub overflow, negative	111100 011	- INF	overflow
add/sub overflow, positive	111100 010	+ INF	overflow
overflow, any	111100 000	± INF	overflow
conversion overflow, negative	111011 111	- INF	overflow
conversion overflow, positive	111011 110	+ INF	overflow
other overflow, negative	111011 101	- INF	overflow
other overflow, positive	111011 100	+ INF	overflow
overflow, unknown sign	111011 000	± INF	overflow
∞ / ∞ invalid	111010 111	NaN	invalid
0/0 invalid	111010 110	NaN	invalid
0 * ∞ invalid	111010 101	NaN	invalid
∞ - ∞ invalid	111010 100	NaN	invalid
invalid, any	111010 000	NaN	invalid
sqrt of negative	111001 111	NaN	invalid
log of negative	111001 110	NaN	invalid
pow invalid	111001 101	NaN	invalid
modulo or remainder invalid	111001 100	NaN	invalid
asin/acos invalid	111001 011	NaN	invalid
acosh/atanh invalid	111001 010	NaN	invalid
division by INF	111000 111	0	infinity loss
exp(-INF)	111000 110	0	infinity loss
pow, compound	111000 101	0, 1	infinity loss
rsqrt, root	111000 100	0	infinity loss
atan, atanh, etc.	111000 011	±π/2, ±1	infinity loss
minimum, maximum	111000 010	input operand	infinity loss
unspecified loss of INF	111000 000		infinity loss
underflow, negative	110111 111	- 0	underflow
underflow, positive	110111 110	+ 0	underflow
abrupt underflow, negative	110111 101	- 0	underflow
abrupt underflow, positive	110111 100	+ 0	underflow
underflow, any	110111 000	± 0	underflow
inexact, round up	110110 111	rounded result	inexact

inexact, round down	110110 110	rounded result	inexact
inexact, round up by ≤ ½ ULP	110110 101	rounded result	inexact
inexact, round down by ≤ ½ ULP	110110 100	rounded result	inexact
inexact, any	110110 000	rounded result	inexact
other standard math functions	110101 xxx	NaN	invalid
other library functions	110010 xxx	NaN	invalid
other functions	110001 xxx	NaN	invalid
application-specific errors	01xxxx xxx	NaN	invalid
user-defined low priority codes and legacy codes	00xxxx xxx	NaN	invalid

Table 2. Status codes in NaN payloads, ordered by decreasing priority.
The first six bits of the status code indicate a category. The last three bits indicate subcategories, which are present in all formats except bfloat16.

The generation of a NaN result for each of the exception types defined in table 2 can be enabled with the respective control bit. The invalid exception will generate a NaN with a diagnostic payload if enabled, or a zero payload if disabled.

This scheme cannot signal more than one exception in the same result. Cases that would set both the underflow flag and the inexact flag according to the current IEEE 754 standard are handled as underflow if the default result is zero or inexact if the default result is nonzero.

Exception types and other error types differ by the way they are generated. Exception types are generated when an arithmetic operation signals an exception. In fact, the NaN status code *is* the signal. These status codes are mostly generated by hardware. Error codes are mostly generated by software or inserted by the user. User-defined status codes are not limited to error messages; they can be used for any purpose when the user wants some information to propagate, for example special values that need special treatment.

The generation of exception codes can be enabled or disabled by control bits. There is one control bit for each exception type, listed in table 2. Error types are generated by explicitly putting the error code into the payload. They have no control bit.

The NaN and its payload are propagated through subsequent instructions. The type of exception or error may be indicated when debugging or when printing out a NaN result.

If two NaNs with different payloads are combined, for example in an operation like NaN1 + NaN2, then the one with the highest payload should be propagated. This will prioritize the most serious errors or exceptions and make sure the operation is commutative.

7.1 New status types

The NaN payload has plenty of bits that can be used for specifying additional types of errors and other status conditions, including user-defined types.

The current proposal includes several new status types: “data not initialized”, “data not available”, and “data not accessible”, and “loss of infinity”

The “data not initialized” status code is intended to cover errors where software reads a variable without properly initializing it. This method can be used by filling an unused or uninitialized memory area with all 1-bits. If such data is accidentally read as floating point

data then it will become a NaN with the error code for “data not initialized”. This will work regardless of precision and alignment. This mechanism is intended as a replacement for signaling NaNs.

The “data not available” status code can be used when a data source fails to deliver a value. For example, if we are making a statistic of temperatures over time, we can use this error code if the thermometer is temporarily malfunctioning.

The “data not accessible” status code can be used if data cannot be retrieved because of failing access rights or network malfunctioning.

7.2 Loss of infinity

The IEEE 754 standard defines representations for positive and negative infinity (INF). INF is propagating through subsequent calculations in most cases so that the output is INF if the input is INF. For example, `INF * 5 = INF`. However, there are cases where an INF input does not generate an INF output. The most common case is division by INF. A finite number divided by INF will give zero. There are also certain mathematical functions that return a finite number when the input is INF, for example `exp(-INF) = 0`.

Here, we propose a new status type named infinity loss to facilitate the propagation of INF through a sequence of calculations. When the infinity loss option is enabled, it will generate a NaN in cases where INF would not propagate. For example, division by INF will generate zero when infinity loss is disabled, but NaN when infinity loss is enabled. This NaN contains a status code indicating the infinity loss condition.

This gives the user a choice between two different ways of tracking overflow and division by zero:

1. Enable the NaN propagation method for overflow and division by zero exceptions. The result will show as a NaN if any of these exceptions has happened during a series of calculations.

2. Enable the infinity loss option and disable the overflow and division by zero exceptions. The result will show as an INF if overflow or division by zero has happened during a series of calculations. In situations where the INF would not propagate, such as division by INF, the result will be a NaN instead. This method will not distinguish between overflow and division by zero, and it cannot show diagnostic information about the place where the INF was generated.

The second method can be used where it is desired to stay close to legacy behavior or where infinity arithmetic is used purposefully. It may also be useful where INF has an application-specific use.

The infinity loss status is intended for NaN propagation only. It is listed as an exception in table 2, but it needs not be supported for legacy exception handling methods, such as traps or a status register.

The infinity loss status will happen in cases where a function has a finite asymptote for an argument going towards ± infinity. This includes:

- division by INF
- exp(-INF)
- pow(a,INF) where |a| < 1
- pow(a,-INF) where |a| > 1
- pow(±1,INF)
- rsqrt(INF)

- rootn (INF, a) where a < 0
- atan(±INF)
- atanh(±INF)
- compound (INF, a) where a < 0
- minimum(a,INF) where a is finite
- maximum(a,-INF) where a is finite

Comparison operations are not affected by this option. INF is considered bigger than any finite number, and -INF is considered smaller than any finite number. This means that the minimum and maximum functions will not activate the infinity loss status if they are implemented as a branch, for example `min(a, b) = a < b ? a : b`.

The infinity loss feature does not apply to the copySign function, which just copies the sign bit from one operand to another.

7.3 Loss of NaN

We propose a similar feature for preventing situations where NaNs would not propagate. There are only a few cases where the IEEE 754 specifies that NaNs do not propagate, as explained in section 9. These cases are:

- pow(NaN,0)
- pown(NaN,0)
- pow(1,NaN)
- hypot(INF, NaN)
- compound (NaN, 0)

When the NaN loss option is enabled, the result will be a NaN in these cases. This option makes it possible to make sure that NaN inputs are propagated through all functions that have a floating point output.

We do not need a special status code for the NaN loss option because it is preferred to just propagate the status code of the input NaN. Therefore, this option has no entry in table 2.

The NaN loss option may be enabled explicitly, or the implementation may enable it automatically when any other NaN propagation feature is enabled.

7.4 Other uses of NaNs

NaNs are traditionally used as results of invalid operations such as zero divided by zero. They are generated by hardware and are quiet NaNs, usually with a payload of zero.

The statistical software system named R is using a NaN payload to distinguish between invalid values and data not available. The payload is zero for invalid values and 1954 for data not available. 1954 is an arbitrarily chosen value. Other statistical systems are apparently not using NaNs for indicating data not available.

These uses of NaNs will not interfere with the propagation of error and exception information in NaN payloads according to the scheme listed above.

Programming languages with dynamic typing of variables, such as JavaScript, are using NaNs with specific payloads for representing values that are not numeric or not floating point, for example Booleans or text strings. This method is called NaN boxing. The coding of NaN boxes differs among languages and is not standardized. Typically, the payload contains a tag indicating the type of object, followed by a value or a pointer to the address of

the object. Some systems store additional information in the sign bit of a NaN box. The sign bit is not part of the payload and has no meaning in other uses of NaN.

The payload in NaN boxes may interfere with the codes for exceptions and errors specified here if both can appear in the same context. NaN boxes are not propagating, but detected immediately. Some current implementations are using quiet NaNs for NaN boxing. Future implementations should preferably use signaling NaNs for NaN boxing to make sure they are distinguished from propagating error codes, which must use quiet NaNs.

8 Can parallel exception handling be implemented in common platforms?

It would be very difficult to implement metadata tags in current computer platforms. First of all, data registers and vector registers would have to be extended with additional tag bits. All arithmetic operations would have to be modified so that they can generate and propagate these tag bits.

However, the biggest problem is how to store data in memory when metadata tags are included. Data stored in memory are usually aligned, relying on the fact that all data types have power-of-two sizes. The combined data plus tag does not have a power-of-two size unless we are adding a lot of unused bits to double the total size. This means that the memory use is doubled. Compilers would need a major update to adapt to this change.

A lot of existing software relies on the assumption that data have specific sizes when calculating data addresses. This assumption fails if metadata tags are added to all data elements. This means that a metadata tag system will prevent the use of legacy software, unless the feature can be turned off.

It is more realistic to implement the NaN propagation method in current systems. NaN payloads were originally introduced for the purpose of carrying diagnostic information. They are rarely used for this purpose today, but now it is time to activate this dormant functionality. The propagation of NaNs with payloads is specified by the IEEE 754 standard and is already supported by most microprocessors.

What is needed for implementing the NaN propagation method is a set of control bits to enable NaN generation for each kind of exception. Floating point arithmetic instructions need a new feature to generate a NaN with a diagnostic payload in case an exception happens and is enabled.

We would like debuggers to display the status code when encountering a NaN. Common print functions may also show the name of the exception or error type when printing out a NaN.

The propagation of NaNs in current systems is not perfect. It has some minor flaws that should be fixed if NaN payloads are used for the purpose described here, or for any other diagnostic purpose for that matter. These flaws, and possible corrections, are discussed below.

9 Propagation of NaNs

A few mathematical functions are not guaranteed to propagate NaNs. These functions are listed here:

Min and max functions

The 2008 version of the IEEE 754 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. A revision of the IEEE 754 standard in 2019 has fixed this problem by defining functions named minimum and maximum, that do the same but with propagation of NaN inputs.

The actual implementation of min and max in many programs 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) = b < a ? 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.

Software programmers should be aware of this and use the proper minimum and maximum functions when NaN propagation is desired.

Power functions
The pow function fails to propagate a NaN in the two 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.

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

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

The less common functions `'hypot' and 'compound'` have similar problems.

The “loss of NaN” exception described in section 7.3 is intended to make sure that NaNs will propagate in these functions.

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.

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 C++ 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 processors have hardware instructions for all cases of comparisons including negated comparisons and 'unordered' comparisons which evaluate as true for NaN inputs.

Avoiding loss of NaN in comparison operations
A comparison operation does not propagate NaN operands. The programmer may want to make special precautions to preserve any information contained in NaN operands.

Systems that use exception trapping should preferably detect an error when a NaN is generated rather than when it disappears. The IEEE 754 floating point standard has some rather confusing rules for detecting NaNs in comparisons. The standard specifies two kinds of comparison operations: quiet and signaling. The signaling comparisons will raise the exception “invalid” when one or both operands are NaN, while the quiet comparison operations raise no exception. The high-level language operators `<`, `<=`, `>`, `>=` generate signaling comparisons, while the operators `==` and `!=` generate quiet comparisons. This is not very intuitive, and few programmers are actually relying on it. The high-level language may or may not provide ways of circumventing this difference. It would be more convenient to have a separate exception type for this purpose rather than the general “invalid” exception. This would allow the programmer to enable or disable exception trapping for all compare operations.

If we are not using exception trapping and we want to preserve the information contained in NaN operands at compare operations, then it may be necessary to check the operands for NaN before the comparison operation. A simple way to check for NaN is `if (a != a)`. This condition will be true only if `a` is NaN. It may be convenient to implement a hardware instruction that checks both operands for NaN and then jumps to an exception handling branch if at least one operand is NaN.

Combining two NaNs
If two NANs with different payloads are combined, then the result will be one of the two inputs, but the standard does not specify which one. Most microprocessors are just returning the first of the two NaNs. This has the unfortunate consequence that the expressions A+B and B+A give different results if A and B are NaNs with different payloads. To prevent unpredictable results, we recommend to propagate the highest of the two payloads.

Conversion between different precisions
Another problem with NaN payloads occurs when the value is converted to a different precision. Experiments on various types of microprocessors show that the NaN payload is handled in the same way as the fraction 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 fraction, as an integer where the least significant bits are preserved. This behavior is currently undocumented, but should be preserved.

10 Conclusion

The standard methods for detecting floating point exceptions were designed long before the methods of SIMD parallelism and out-of-order parallelism became common. 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 situations where 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 hardware of the out-of-order scheduling logic.

Modern compilers are able to convert code with sequential loops to SIMD code. But most compilers can do this 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 the detection of exceptions is designed in a way that matches the kinds of parallelism used. Any status information should follow the individual result and the subsequent data flow. If an exception result is used in subsequent calculations then the status information should propagate to the subsequent results.

This can be obtained either by using metadata tags or by NaN propagation to trace floating point exceptions. The metadata tag method has a problem when tagged data are saved to memory, while the NaN propagation method needs no extra storage. A propagating NAN can contain a payload with diagnostic information about the type of exception or error, and possibly also the code address where it occurred. The mechanism for propagating NaNs and their payloads is already in place and is supported by most microprocessors.

Such a system where status information follows the data would need no traps for floating point exceptions and no global status register. We can improve the efficiency of parallel execution and out-of-order execution by adding a hardware mechanism that will generate NANs in case of enabled floating point exceptions and insert diagnostic payloads in these NANs. Microprocessor hardware can be made more efficient and consume less power if NaN propagation is the only method of handling exceptions.

The IEEE 754 standard needs revision to make sure the improvements discussed here are permitted. Such a revision is currently being discussed.