Parallel floating point exception tracking and NaN propagation

By Agner Fog. Last updated 2025-12-20.

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 lets 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 1
2 Terminology 2
3 Exception trapping 3
4 Using a status register 4
5 Parallel exception detection 5
6 Experimental implementations 7
7 Can parallel exception handling be implemented in common platforms? 9
8 Propagation of NaNs 9
9 Conclusion 11

1 Introduction

Modern computing systems are using parallel data processing to an increasing degree in order to maximize performance. High levels of parallelism are particularly important in big data processing, vector processing, graphics processing, sound processing, neural networks, and machine learning.

Current methods for detecting numerical errors or exceptions are not attuned to parallel processing because they rely on sequential logic. The most common ways of detecting numerical errors or exceptions during floating point calculations are:

1. Exception handling through the trapping of exceptions
2. Reading 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 you 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. 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.

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. Current systems cannot adequately handle the situation if a SIMD instruction causes multiple exceptions.

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.

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.

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. 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.

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.

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 set the status register. Most compilers are unable to work around this problem.

One may propose a vector status register with one set of status bits for each vector element. However, such a solution would be quite messy if the program is using multiple vector registers of different sizes and different data types. It becomes even more complicated if the same program is compiled for different microprocessors with and without SIMD capabilities, or with different vector register sizes.

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.

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 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 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 and 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 include a payload containing information about 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, generate a trap when a value with an exception tag is stored to memory. 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.

- 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 known to be erroneous.

- 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 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.

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 also want to know the rounded result, then you have to execute the same calculation twice, with and without the 'inexact' exception enabled. Likewise, you cannot distinguish between positive overflow and negative overflow because the sign of a NaN does not propagate in the same way as the sign of an INF does.

- 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 exception code

The IEEE 754 standard specifies five flag bits for 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 number code for each type of exception. With a five-bit number code, we would be able to distinguish between 31 different types of exceptions. If we have more bits available, we will be able to add user-defined exception and error types and store 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 (9 with half 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 will 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

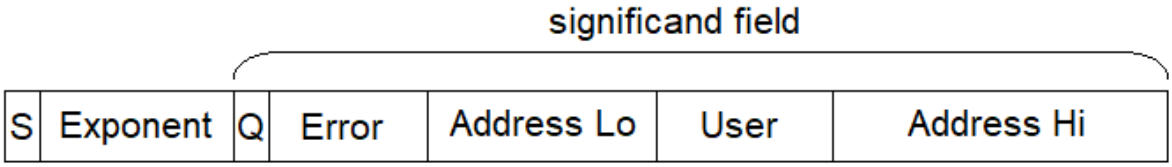
The metadata tag method is used in the Mill computer, that is under development by Mill Computing, Inc (millcomputing.com). Each data register and each vector element has a metadata tag. The tag can indicate the five exception types, as well as missing data. The tag is propagated through a sequence of 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 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 indicating the type of exception as well as additional diagnostic information. 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 error codes. The status code is structured as shown in this figure:



The NaN payload is using the significand field, which contains the fraction in normal floating point numbers. This is organized as a bitfield containing the quiet bit, error or exception code, low and high part of the code address where the exception happened, and a field with optional user-defined diagnostic information. Low precision floating point formats have a smaller significand field containing only the most important error information. This information is preserved when converting to a different precision. Details about the error codes and organization are documented in the ForwardCom manual.

The NaN and its payload is propagated through subsequent instructions. The type of exception or error is indicated when debugging or when printing out a NaN result. The possible status codes are listed in the ForwardCom manual. There is space for additional error codes and user-defined codes. A code for missing or uninitialized data eliminates the need for signaling NaNs.

If two NaNs with different payloads are combined, then the one with the highest payload is propagated. This method will prioritize the most serious errors or exceptions.

Function libraries are modified so that all functions are guaranteed to propagate NaNs.

The NaN method does not work for integers, because there is no integer NaN. Signed and unsigned integer overflow can be detected in ForwardCom by the metadata tag method. This is implemented in special vector instructions that use the even-numbered elements in a vector register for integer arithmetic while the odd-numbered elements contain propagating status flags. These instructions are used only when the detection of integer overflow is desired.

7 Can parallel exception handling be implemented in common platforms?

It would be very difficult to implement metadata tags in current computer platforms. First, all data registers and vector registers must be extended with additional tag bits. All arithmetic operations must 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 perhaps 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 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.

8 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.

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
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. 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, it would be better to select the highest of the two payloads.

Conversion between different precisions
Another problem with NaN payloads occurs when the value is converted from double to single precision, or vice versa. 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 undocumented, but should be preserved.

9 Conclusion

The standard methods for detecting floating point exceptions were designed long before the methods of out-of-order parallelism and SIMD 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 common 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 the detection of exceptions is designed in a way that matches the kinds of parallelism used. Any status information should follow the individual result. 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 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.

The IEEE 754 standard is not very clear about whether the status must be stored in a single global status register with five flags, or whether other ways of storing the status are allowed. The standard may need revision to make sure the improvements discussed here are allowed.