Ternary27
A Balanced Ternary Floating Point Format
Version 3.1

Introduction

In recent years, hobbyists have successfully built proof of concept ternary logic components in silicon[1]. Based on my assumption that these recent successes will lead to further advances, I have begun looking at the next developments that are needed for the subject to progress. Specifically, processor architecture, data types, instruction sets and algorithms.

I have breadboarded and tested ternary adders and subtracters, worked out preliminary instruction sets, defined a feature-rich status register, developed circuits to calculate flags, defined primitive data types, discovered numerous useful masking and tritwise operations, and so on.

I have also been looking into floating-point numbers. This document is the technical definition for a balanced ternary floating point data type and the details needed to implement it in hardware, software or, more likely, a combination of them.

Contents:

1. Overview

   This document describes the format and various aspects of operation of a floating point standard for balanced ternary computer systems. It provides a single 27-trit format which approximates, but has higher precision and greater range than the IEEE 754 binary single-precision float (binary32). This format is referred to as ternary27 and has the purpose of providing a simple standard that takes advantage of the beneficial aspects of a three-valued balanced radix.

   To aid in study, a glossary of terms is included at the end of this document.

2. Compatibility

   Because programmers have become accustomed to the behavior of languages, libraries and hardware that implement IEEE 754 floating point numbers (binary32), and because their behaviors are based on sound mathematical principals, I have ensured that I included equivalent behaviors in ternary27. Where the radix itself makes this impossible or irrelevant, I have created logically consistent replacements. This work is not directly based on the IEEE 754-2008, Standard for Floating-Point Arithmetic[2] but I have sought to maintain consistency with the trained-in expectations that arise from the ubiquitous use of the standard. Much of the IEEE 754 terminology is in common use and some of those words are used here with the same basic meaning, adjusted as necessary to match the differences inherent in using a radix which is both ternary and balanced.

   IEEE 854-1987, the IEEE Standard for Radix-Independent Floating-Point Arithmetic[3] has been conformed to wherever possible. A perfect rendering of that standard is not possible because, while its authors did consider any integer radix value, they did not anticipate a balanced radix. Some aspects of the standard are inapplicable to a balanced radix. Where this is the case, equivalent and logically consistent alternatives have been included in the ternary27 standard.

3. Design Philosophy

   Anyone studying IEEE 754 will find that it is “seemingly rather complex”, an opinion stated by the primary author of the standard[4]. Nevertheless, the reasons for those complexities appear to have been thoroughly justified at the time the standard was drafted, when microprocessors were just coming into market dominance over mainframes and mini-computers. In modern times the limitations on die space and transistor count are essentially non-existent on anything less complex than a high end multi-core processor. With this in mind, the design philosophy behind ternary27 is not focused on the same problems as binary32 was when it was drafted. Instead it is focused on only two goals. First, that it is correct mathematically. Second, that it is logically consistent, and therefore comparatively simple. By this I mean that there are as few “special cases” as possible and that they behave in ways that are reasonable to expect. By following these two guiding principals, I have arrived at a comparatively simple to implement standard that produces correct and reliable results.

4. Balanced Ternary Notation

   A digit in a balanced ternary system is called a trit and can have one of three states. These states are represented by the symbols – (minus), 0 (zero), or + (plus). They have numerical values of -1, 0, or 1. In integer representation, the value of a multi-digit number is found in the same way as any other positional system, by multiplying the value of each digit by the weight of that digit and summing the products. In ternary, the least significant digit has a weight of 1, the next has a weight of 3, then 9, then 27, and so on (powers of three). As an example, the balanced ternary number +-0-, would have a value of (1*3^3) + (-1*3^2) + (0*3^1) + (-1*3^0) = (27 – 9 + 0 – 1) = 17.

   For a fractional number, each position further to the right of the radix point is one power of three less than the prior position. For example, +.-0+ would have a value of (1*3^0) + (-1*3^-1) + (0*3^-2) + (1*3^-3) = (1 – 0.333... + 0 + 0.037…) = approximately 0.704.

   The sign of a number is always the sign of the most significant non-zero trit. For this reason, no signed representation format is needed under any circumstance. All ternary integers are either signed negative or positive based only on the value of the most significant non-zero trit. When all trits of an integer are 0, then the value is zero and could be said to have “no sign”.

   To illustrate this point, take the above example ternary integer; +-0- (17). If we invert this number by making all -’s into +’s and all +’s into -’s and leave 0’s as the are, we get this number; -+0+ (-27 + 9 + 0 + 1 = -17). As you can see, no special allowances such as signed magnitude, 1’s complement, 2’s complement or anything else are necessary to represent negative numbers. Their representation is inherent in the digits themselves. The leftmost non-zero trit is the sign of the number, regardless of its position.

5. Floating Point Format

   The ternary27 format represents a subset of the real number line as well as infinities, quiet NaNs and signaling NaNs. The format is 27 trits wide and is composed of a 2-trit type field, a sign trit, a 5-trit exponent field and a 19-trit significand field. A notable aspect of Ternary27 is the inclusion of type codes to avoid using special bit patterns to represent non-numbers and to provide some meta-data about the values.

Binary32 Floating Point Format																															
Sign	Exponent, 8 Bits								Mantissa, 23 Bits																						
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Ternary27 Floating Point Format																													
Type		Sign	Exponent, 5 Trits					Significand, 19 Trits																					
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			

   The finite set of floating point numbers representable by this format is determined by:
   b = the radix, (3)
   s = the number of digits in the significand for normal numbers, including sign trit, (20)

es = the number of digits in the extended significand for subnormal numbers, including sign and exponent field, (25)
eMax = the maximum exponent, (121)
eMin = the minimum exponent, (-121)

Any number that can be represented by the ternary27 format is computed as the value of the significand times the base raised to the power of the exponent: s*(b^e).

The format can also represent the following infinities and NaNs by the use of type codes in combination with the sign trit:
Positive ∞
Unsigned ∞
Negative ∞
Positive signaling NaN
Unsigned signaling NaN
Negative signaling NaN
Positive quiet NaN
Unsigned quiet NaN
Negative quiet NaN

6. Minimal Test Format – Ternary6

   For test implementation purposes, I propose the “toy” format, ternary6. This has two type code trits, a sign trit, two exponent trits, and one significand trit. This is the absolute minimum format to test proof-of-concept hardware or software designed to implement the standard. In this format eMax is 4 and eMin is -4.

7. Type Codes

   The format specifies nine type codes. Each type code is two trits long:

++	=	Infinity
+0	=	Quiet NaN
+-	=	Signaling NaN
0+	=	Real Number (rounded down)
00	=	Real Number (infinite precision)
0-	=	Real Number (rounded up)
-+	=	Reserved, treat like a signaling NaN
-0	=	Reserved, treat like a signaling NaN
--	=	Reserved, treat like a signaling NaN

Type Code	Sign	Exponent	Significand	Type	Notes
+ +	+,0,-	Payload		Infinity	
+ 0	+,0,-	Payload		Quiet Not-A-Number	
+ -	+,0,-	Payload		Signaling Not-A-Number	
0 +	+,0,-	x x x x x	x x … x x	Real Number (Rounded Down)	The value is the result of rounding down
0 0	+,0,-	x x x x x	x x … x x	Real Number (Infinite Precision)	The value has infinite precision
0 -	+,0,-	x x x x x	x x … x x	Real Number (Rounded Up)	The value is the result of rounding up
- +	+,0,-	Payload		Reserved, treat like a signaling NaN	
- 0	+,0,-	Payload		Reserved, treat like a signaling NaN	
- -	+,0,-	Payload		Reserved, treat like a signaling NaN	

Type codes are set by arithmetic operations, at the time of initialization, or by direct type assignment operations. Any type code is a floating point datum but only the three real number type codes and the infinity type code are floating point numbers.

There are three type codes for Real Numbers; rounded down, infinite precision, and rounded up. These relate to the exactness flag but are also stored in the data type itself for the purpose of distinguishing true zero from zero's that are the result of rounding and are not exactly zero. This solves the problem of detecting negative zero, but also allows for greater precision in comparison operations. If x is a floating point number, one could distinguish between x (infinite precision), x (rounded down) and x (rounded up). The standard allows comparison operations and predicates that take these type codes into account for greater precision, but also requires conventional comparison operations that do not.

Newly initialized or existing real numbers directly assigned a new value are assigned type 0-, 00 or 0+ depending on whether rounding is needed to represent that value in ternary27. Newly instantiated variables that have not been initialized are assigned type +- (signaling NaN) with the sign trit set to 0 by default. The payload for uninitialized variables may be determined by the implementation but the default value is all-zeros.

8. Subnormal Numbers

Subnormal numbers are any value with a real number type code (0-, 00 or 0+) where the sign trit is 0. To illustrate, take a positive number that is decrementing toward zero. As the number approaches zero it eventually arrives at the lowest possible exponent of -121 (eMin). You can see this here where the significand and exponent have reduced down to all minuses, but the sign has not yet decremented to 0.

Type	Sign	**Exponent, 5 Trits**	Significand, 19 Trits
0 0	+	- - - - -	- - - - - - - - - - - - - - - - - - -

As the number decrements one more time, the sign trit changes from a + to a 0. When this occurs, the number is subnormal and the exponent of the number is implicitly assumed to be eMin (-121).

Type	Sign	**Extended Significand, 5 Trits**	Significand, 19 Trits **(24 trits with extension)**
0 0	0	+ + + + +	+ +

As the exponent trits are no longer needed, they can be used to increase the size of the significand, adding 5 more trits of precision to all operations involving subnormal numbers. However, because the sign trit is zero for all subnormal numbers, it is lost as a useful trit. This makes the effective increase only four trits. This behavior allows for increased precision in the subnormal area where it is most likely to be needed. Naturally, ternary27 suffers from gradual precision loss the closer your get to zero just as in binary32. However, the effect is more gradual due to the added precision and the fact that three values are represented with each digit.

Subnormal numbers have a sign trit set to 0, but the sign of the number still needs to be computed. This is done by taking the most significant non-zero trit of the significand as the sign of the number.

A consequence of this arrangement is the necessity to have a floating point register five trits wider than would otherwise be necessary (essentially six guard trits instead of one) and the need for a multiplexer to choose whether to place the 5 additional trits of significand or the exponent in the final result based on the value of the results sign trit. This very minor increase in hardware complexity yields substantially increased precision.

There is a limitation to be taken into account when dealing with subnormal numbers. When one or both operands are subnormal and the result is a normal number, the result does not benefit from the additional precision afforded by the extended significand. This is because the resulting normal number doesn't have the extra trits and thus cannot be rounded by them. When one or both operands are subnormal and the result is subnormal, it gains the full advantage of the extended significand. Final and intermediate results that are in the subnormal range will suffer from far less error than would otherwise be the case, but if further operations bring the results back into the normal range then those operations will have the usual degree of rounding error.

The primary benefit and purpose of this arrangement is that it entirely eliminates the problem of "catastrophic cancellation" arising from subtracting one normal number from another. Using this system, there is no combination of normal numbers that will, when one is subtracted from another, result in incorrectly rounding to 0 since the resulting subnormal number has greater precision than either of the original two numbers. Only when two normal numbers are identical in their sign, exponent and significand fields, will subtraction result in a 0. An entire class of floating point software bugs is eradicated using this system.

9. Types of Zeros

Zero is defined as a subnormal real number with a sign trit and extended significand (exponent and significand fields combined) composed entirely of 0's. Because zero is a subnormal number, the exponent is implicitly eMin. The type code can be any of the three real number types. The type code provides the data by which infinite precision zero can be distinguished from zeros that were arrived at by rounding and are not exactly zero. This can arise where the true value of an arithmetic result is not exactly zero, but is so close that there is no other nearer representable number.

A special type for negative (or positive) zero is unnecessary and no additional hardware or software is necessary to address a "special case" for negative 0. When a number is rounded up, the rounded up type code is set. The significand being all-zero's and the 'rounded up' type code being present, together demonstrate that the value is negative 0. All-zero's with the 'rounded

down’ type code would be positive 0. This information would be available via the sign flag and exactness flag at the completion of an operation but is also stored in the data type itself through the type code.

Besides the type code, there is no distinguishing feature to separate signed and unsigned zeros. In all three cases, the sign trit and all of the extended significand trits are zero and the exponent is implicitly eMin. Therefore, unless an operation specifically takes into account the type code of the operands, all zero’s are mathematically identical.

The standard requires that conventional comparison operations find zeros with any of the three real number type codes to be identical. In other words, -0 = 0 = +0. However, the standard also allows predicates and operations that distinguish between them such that -0 < 0 < +0. In either case, unsigned zero always compares equal to any other unsigned zero and to itself.

10. Rounding

The default rounding mode is round to nearest. This is the only required rounding mode. As noted by Donald Knuth; in balanced ternary, rounding to the nearest number and truncation are the same operation[5]. A balanced ternary number with more digits cannot be exactly halfway between the two next nearest floating point numbers with fewer digits, so there is no decision-making process to determine which way to round. You simply remove the extra digits and it is properly rounded. In the event of an overflow when the rounding mode is round to nearest, the overflow produces either a +∞ or a -∞ as appropriate.

In addition to the default round-to-nearest mode, there are three optional directed rounding modes. Operations must exist to set the rounding mode if these modes are implemented.

The directed rounding modes are defined as follows:

Round toward -∞
- All rounding is biased towards -∞.
- If the most significant trit lost to truncation is a +, the number is correctly rounded.
- If the most significant trit lost to truncation is a 0, the number is correctly rounded.
- If the most significant trit lost to truncation is a -, decrement the result.
- Positive overflow produces the most positive finite representable number.
- Negative overflow produces -∞.

Round toward +∞
- All rounding is biased towards +∞.
- If the most significant trit lost to truncation is a +, increment the result.
- If the most significant trit lost to truncation is a 0, the number is correctly rounded.
- If the most significant trit lost to truncation is a -, the number is correctly rounded.
- Positive overflow produces +∞.
- Negative overflow produces the most negative finite representable number.

Round toward 0
- All rounding is biased towards 0.
- If the number is positive then apply the same rules as round toward -∞.
- If the number is negative then apply the same rules as round toward +∞.

Positive overflow produces the most positive finite representable number.
Negative overflow produces the most negative finite representable number.
Overflow never produces an infinity.

Rounding a number also sets the type code to show which way it was rounded, up or down. A useful benefit is imparted by storing this meta-data directly inside the format. A number with a rounded type code gives you an upper and lower bound on where the true value of the number is. This is not as sophisticated as using interval arithmetic but gives users some of the benefits of interval arithmetic “for free”.

Call the distance between one representable number and the next for a given exponent the “gap” for that exponent. A rounded-down data type tells you that if the returned value is x, then the true value lies somewhere between x and x + ½gap. Thus, the exponent, the type code, and the value are all you need to put a comparatively tight upper and lower bound on the true value of the inexact number and each of those values are included in the data type itself.

For comparison, in binary32 the fact of inexactness is only reported in the status register and is thus obscured by the next operation. If one checks for inexactness at the time of the operation then you would know that the number had been rounded, but not whether it had been rounded up or down. Therefore the upper and lower bound for the true value ranges from halfway to the next representable number below and halfway to the next representable number above the returned value. The range for an inexact ternary27 number is only ½ of a gap and it is directed, giving a known maximum interval and direction.

Another point to take into account is that the subset of numbers representable without any rounding in any floating point format is different depending on the radix. Because the radix of this format is 3 rather than the ubiquitous 2 of binary, the numbers that can be represented without rounding are different than the numbers representable without rounding in binary32. For example, binary32 is incapable of representing 1/3 without rounding whereas ternary27 can do so with infinite precision. Amusingly, ternary27 cannot represent ½ without rounding, but binary32 has no difficulty with that. Neither format can represent 1/10 without rounding. This is not a failure on the part of either standard, but a natural consequence of the different radixes.

11. The Sign Trit

    This standard explicitly stores the first digit to the left of the radix point as the sign trit. This provides several benefits and does not suffer from the “waste” of a digit as it would in a binary system. Binary numbers need to have a separate bit to define whether they are negative or positive and thus have a sign bit. Balanced ternary integers include the sign natively in the number itself as the most significant non-zero digit. The sign trit serves as the most significant trit of normalized numbers and is therefore the sign of any normal number. When it is 0, indicating a subnormal number, the sign of the number is the most significant non-zero trit in the significand. Even in the case of subnormal numbers, the sign trit still mathematically serves as the digit to the left of the radix point. That digit just happens to be 0.

12. Balanced Ternary Radix – Range

    With 5 trits, the exponent can represent 3^5 = 243 different values. One of those values is zero and the other 242 are balanced above and below zero. This makes eMax 121 and eMin -121.

This is fewer than the -126 to +127 of binary32 but the fact that these numbers are in radix 3 means that each exponent represents a larger range than in binary. The overall range in decimal including subnormal numbers is from ±6.57E-70 to ±8.10E+57. For comparison, binary32 has a range from ±1.40E-45 to ±3.40E+38 including subnormal numbers. In fact, the entire binary32 subnormal range actually falls within the normal range of ternary27.

Binary32 Float Values for Sample Exponents		
Exponent	Smallest Magnitude	Largest Magnitude
-126 Subnormal	±1.4E-45	±1.2E-38
-126 Normal	±1.2E-38	±2.4E-38
-121	±3.8E-37	±7.5E-37
-50	±8.9E-16	±1.8E-15
-10	±9.8E-04	±2.0E-03
0	±1.0E-00	±2.0E+00
10	±1.0E+03	±2.0E+03
50	±1.1E+15	±2.3E+15
121	±2.7E+36	±5.3E+36
127	±1.7E+38	±3.4E+38

Ternary27 Float Values for Sample Exponents		
Exponent	Smallest Magnitude	Largest Magnitude
-121 Subnormal	±6.6E-70	±9.3E-59
-121 Normal	±9.3E-59	±2.8E-58
-50	±7.0E-25	±2.1E-24
-10	±8.5E-06	±2.5E-05
0	±5.0E-01	±1.5E+00
10	±3.0E+04	±8.9E+04
50	±3.6E+23	±1.1E+24
81	±2.2E+38	±6.7E+38
121	±2.7E+57	±8.1E+57

Any normalized floating point number in the Ternary27 format can be computed as the sum of the sign trit and each of the significand trits with the sign trit being raised to the power of 0 and each successive trit being raised to the power of 1 less than the preceding trit. The result is then raised to the power of the exponent.

Sign	Significand					(sign + sum(sig-1...sig-19))	Exponent				
^0	^-1	^-2	...	^-18	^-19		-121	-1	**0**	1	121
1	1	1	...	1	1	1.500000000E+00	2.782399188E-58	4.999999999E-01	**1.500000000E+00**	4.499999999E+00	8.086546347E+57
1	0	0	...	0	0	1.000000000E+00	1.854932792E-58	3.333333333E-01	**1.000000000E+00**	3.000000000E+00	5.391030900E+57
1	-1	-1	...	-1	-1	5.000000004E-01	9.274663969E-59	1.666666668E-01	**5.000000004E-01**	1.500000001E+00	2.695515452E+57

Any subnormal number can be computed as the sum of the trits in the extended significand with the first being raised to the power of -1 and each successive trit being raised to the power of 1 less than the preceding trit. The result is then raised to the power of eMin (-121). The sign trit can be ignored in computing subnormal numbers as it is always 0.

Sign	Extended Significand					sum(sig-1...sig-24)	Exponent
^-0	^-1	^-2	…	^-23	^-24		-121
0	1	1	…	1	1	4.99999999998E-01	9.27466396125E-59
0	0	0	…	0	1	3.54070616147E-12	6.56777196666E-70
0	0	0	…	0	0	**0**	**0**
0	0	0	…	0	-1	-3.54070616147E-12	-3.54070616147E-12
0	-1	-1	…	-1	-1	-4.99999999998E-01	-4.99999999998E-01

13. Precision Definitions

Terminology and definitions related to floating point precision are not entirely consistent from author to author and much confusion can arise due to the use of the same words with slightly different (or vastly different) meanings. Even basic texts and popular works on the subject do not always agree on these definitions. To avoid such confusion to the best degree possible, the formula is given for each word as used in this document.

b = base
e = exponent (the value, not the number of digits)
p = the number of digits in the significand
s = the number of digits in the significand plus the sign trit (p+1)

In ternary27, b = 3
For normal numbers, p = 19 and s = 20
For subnormal numbers, p = 24 and s = 25

Machine Precision (u) = b^{-p}
Machine Epsilon (ϵ) = (b/2) * b^{-s}

For normal numbers, u = 8.60E-10 and ϵ = 4.30E-10
For subnormal numbers, u = 3.54E-12 and ϵ = 1.77E-12

These measurements are constant regardless of the value being represented or the operation being performed. They depend on nothing other than the structure of the format itself.

Gap (g) = u * b^{e}
Maximum Relative Error (m) = ϵ * b^{e}

The values g and m are variables that change with the exponent. When the exponent is 0, g is identical to u and m is identical to ϵ.

The gap(g) is used to show how close together two adjacent representable numbers are. For instance, two adjacent numbers (in non-normal form for clarity) with an exponent of 14 are 4,782,968.99588477 and 4,782,969.00000000. The gap between the two is 0.00411523. Each successive increment of the exponent triples the range of the numbers it can represent but also triples the gap.

The Maximum Relative Error(m) is useful for determining the highest degree of error that might be introduced into an operation as a consequence of rounding the final result if it is a number that cannot be represented with infinite precision (no rounding). The worst case scenario is when the result of an operation is directly between two representable numbers. In this case, rounding will cause the maximum degree of error for that exponent. For example, an operation results in the number 1.500000005, which cannot be represented. The smallest exponent that can represent this range of numbers is 1 but the result is exactly halfway between two numbers that can be represented with exponent 1. Thus, the maximum relative error is realized after rounding has occurred. In the vast majority of cases the actual rounding error would be less than the maximum relative error.

14. Balanced Ternary Radix – Precision

The precision level of ternary27, expressed in terms of the gap between two adjacent numbers, is finer for every representable range of numbers than the same range in binary32. For example, if a number around 1,950.00 were represented in binary32, the normal form floating point number would use the lowest possible exponent that could represent that value, which is 10. That exponent has a precision level (gap) of 1.2E-4. The same range in ternary27 would use the

exponent 7 and would have a precision level (gap) of 1.9E-6, two orders of magnitude more precise.

Binary32's value of lowest magnitude is ±1.4E-45. Ternary27's lowest magnitude number is ±6.6E-70, 25 orders of magnitude more precise.

The highest magnitude binary32 number is ±3.4E+38 with a gap of 2.0E+31. The same range in ternary27 has a gap of 3.8E+29, two orders of magnitude more precise. Ternary27's extends up to ±8.1E+57, 19 orders of magnitude greater than binary32. At that range it has a gap of 4.6E+48.

The reason for the great difference in precision is due to the fact that in binary, adding an additional bit causes the number of unique values to double, but for every additional ternary trit, the number of unique values triples. Binary32 has 24 bits in the mantissa (including the "hidden" bit) which means that it can represent almost 17 million different values per exponent. Because of the additional sign bit, it can represent negative and positive numbers with the same mantissa bits, so that comes to about 34 million different values per exponent. Ternary27 has 20 trits between the significand and the sign trit, meaning that it can represent almost 3.5 billion different values per exponent. This is about 103 times more values per exponent than binary32. With the effective addition of 4 more trits of significand, ternary27 has over 282 billion values in the subnormal range.

15. Floating Point Comparisons

The subject of comparing floating point numbers is an exceptionally complex topic. The ternary27 format is specifically designed to reduce some of that complexity. Specifically, *from the minimum possible value to the maximum possible value, every real number is adjacent in proper sequence including zero and the infinities.* Incrementing the lowest value negative real number repeatedly will cause the value to eventually arrive at the highest value positive number, passing properly from negative to positive through the entire subnormal range and zero. The complementary exponent exists to make this possible and the position of the sign trit to the left of the exponent is also necessary to permit this. The type code, which is in the two most significant positions, is excepted from this rule because it is not part of the numerical value.

While the ordinary = = comparison operation exists in this standard, as with binary floating point, it is not a recommended method of comparing floating point numbers. Specific methods of comparison are not part of this standard. However, data about the standard that will be of use in designing useful "equality" comparisons include:

a. Every adjacent representable value has an adjacent integer representation.
b. -0, 0, and +0 have the same integer representation (only the type code varies).
c. There is no discontinuity between the integer representations of zero and the nearest non-zero subnormal values.
d. There is no discontinuity between the integer representations of normals and subnormals.
e. The gap between adjacent normal values triples each time the exponent increases by one.
f. The gap between the highest value in one exponent and the lowest value in the next higher exponent is double the gap of adjacent values in the lower exponent range (Max Relative Error of the lower exponent range times 2).

g. The ratio of the gaps between subnormal numbers increases as the values approach 0, ending with a 2/1 ratio between the smallest subnormal number and the second smallest subnormal number and an infinite ratio between the smallest non-zero subnormal and 0.

16. Flags

Because a trit is capable of representing three different values, ternary flags can be more expressive than their binary counterparts. This allows for more sophisticated decision-making based on flag values. It also requires a different approach to the vocabulary surrounding the subject of flags. With binary values a flag is said to be raised or set if the value is 1 and is said to be lowered or not set when the value is 0. In ternary, a flag can be either -1, 0 or 1 so what action constitutes “raising” the flag? Is it “lowered” only if set to -1? To ensure clarity, the term “set” is always used rather than “raised” or “lowered” and a flag is always “set” to something even if it is 0. “Reset” is used only to describe setting a flag to 0. The names of the flags also needs to be considered. In binary it is perfectly sensible to call one flag the “overflow flag” and another the “underflow flag” because each can only represent the condition being met or not being met. With ternary representation the flags can communicate more data and thus require more generalized names. For example; the “range flag” replaces the binary32 underflow and overflow flags. It is set to – for an underflow event, + for an overflow event, and 0 when neither has occurred.

There are five flags defined in this standard. These are the exactness flag, range flag, sign flag, computability flag and validity flag. The states of the exactness, range, computability and validity flags correspond to exceptions but the state of the sign flag does not. Its purpose is equivalent to the sign flag present in integer arithmetic-logic units; to aid in conditional jump instructions following arithmetic or comparison operations.

Reseting a flag means returning it to zero. Setting a flag means setting it to any of the three possible values. The default flag states are:
Exactness flag set to 0 (no rounding error, infinite precision)
Range flag set to 0 (no overflow or underflow, the result is a normalized real number)
Sign flag set to 0 (zero result, any exactness)
Computability flag set to 0 (computable operation)
Validity flag set to 0 (valid operation)

The different flags have different stickiness behaviors as described below. These behaviors are specifically designed such that the flags will give more data than their binary counterparts in most cases. There is one exception to this rule regarding underflows described below.

The exactness flag conditions are:
- The most significant non-zero trit of the significand lost to rounding was a -. This indicates that the result was rounded up.
0 No non-zero trits were lost to rounding. This indicates an infinite precision result.
+ The most significant non-zero trit of the significand lost to rounding was a +. This indicates that the result was rounded down.

The exactness flag – and + states may toggle back and forth with each successive inexact result, but will not reset to 0 without an explicit instruction. This means that any non-zero state shows

that there has been at least one inexact result since the last reset and that the flag always shows the type of inexactness of the most recent inexact result.

The range flag conditions are:
- The result is too small to be represented in normal form. An underflow has occurred and the result is a subnormal number.
0 The result is representable in normal form. No overflow or underflow has occurred.
+ The result is too high in magnitude to be represented in the ternary27 format. The most significant trit of the exponent overflowed off of the exponent field. Depending on the rounding mode, either a -∞, +∞ or the highest negative or positive magnitude representable number is returned.

The + state is treated as more sticky than the – state. Thus, a later overflow will obscure a previous underflow if underflow is not trapped. This is the one case where a ternary27 flag may provide less data than the equivalent binary32 flags in some circumstances.

The sign flag conditions are:
- The result is a negative floating point datum.
0 The result is an unsigned floating point datum.
+ The result is a positive floating point datum.

The sign flag is never sticky.

The computability flag conditions are:
- The operation was a finite nonzero number divided by 0 (any sign).
0 The operation was computable or resulted in a NaN.
+ The operation had finite operands but resulted in an infinity.

The + state is treated as more sticky than the – state. Thus, if a minus state is present then you know that a divide-by-zero occurred, but if it is + then you know that there has been at least one uncomputable operation of the more general category; an infinite result from finite operands since the last reset.

The validity flag conditions are:
- The operation was not valid and returned a quiet NaN.
0 The operation was valid and returned a floating point number.
+ The operation was passed one or more signaling NaNs and returned a quiet NaN.

The + state is treated as more sticky than the – state. Thus, if a minus state is present then you know that an invalid operation caused it, but if it is + then you know that there has been an invalid result has been caused by a signaling NaN being passed as an operand since the last reset.

Operations to change or test flags are as follows:

- resetFlag(flag)
  resets the given flag to its default state
- resetAllFlags()

resets all flags to default states
- setFlag(flag, state)
  sets the given flag to the given state (-, 0 or +)
- testFlag(flag)
  returns the value of the given flag
- saveAllFlags()
  returns the state of all flags
- testSavedFlags(flags)
  takes the result of saveAllFlags and returns the state of the saved flags
- restoreFlags(flags)
  takes the result of saveAllFlags and sets all flags to the saved states

17. Payloads

Binary32 NaNs have a payload in the 23-bit mantissa. This is primarily intended to provide diagnostic data about the operation that caused the NaN, but this behavior is not mandatory. The payload can be used for any purpose or ignored. Binary32 infinities do not have payloads.

In ternary27, both NaNs and infinites have a 24-trit payload that span the significand field and the exponent field. The sign trit is not part of the payload and continues to serve as the “sign” of the floating point datum. If it is 0, the datum is unsigned. Payloads should contain values that help diagnose the original cause of the NaN or infinity. However, this behavior is not mandatory and the payload may be used for any purpose at the discretion of the implementation. In binary32, infinites are designated by a specific bit pattern and thus cannot have a payload. In ternary27, infinities are designated by a type code leaving the significand and exponent free, and therefore they may be treated as a payload.

When a NaN or an infinity results from an operation not involving a NaN operand, the payload defaults to the computed result of the operation unless the payload is designated for a particular purpose by the implementation. When a NaN or infinity is used as an operand the default behavior is to make the resulting payload the same as the input payload of the NaN or infinity. NaNs take precedence over infinities if a NaN and an infinity are operands of the same operation. If two infinities or two NaNs are operands, the payload is that of the first operand.

18. Compatible Order and Total Order

The compatible order and total order predicates are a pair of operations that compare two floating point datums of any type and any value. They have a broader application than an ordinary comparison operation. For the purposes of this document, they have the form of compatibleOrder(x,y) and totalOrder(x, y). Both are required by the standard but their form is at the discretion of the implementation. They both return a + if the second datum is higher in order than the first, return a 0 if they are precisely equal (both type code and value) and return a - if the second datum is lower in order than the first.

The order predicates are not the same as a comparison operation. Comparisons consider NaNs invalid and can raise exceptions. The order predicates always determine an ordering no matter what the operands are, and never raise exceptions. Default comparison operations ignore the sign of zero, but order predicates do not. Default comparison operations also treat unsigned

infinity the same as positive infinity, which is true of compatibleOrder, but is not the case for totalOrder.

A comprehensive description of the ordering of all possible floating point datums is required to define how the order predicates should behave. Due to having three-valued digits, it is possible to do more detailed ordering than can be done in binary32. However, the way binary32 orders is a well known, established pattern. Therefore, the ternary27 format definition requires two sets of ordering. One set, compatible order, ignores some type code and sign data to maintain compatibility with commonly expected behavior while the other, total order, takes full advantage of all type, sign and payload data for more detailed ordering. Below is a table showing binary32 total order and the corresponding “compatible order” and “total order” for ternary27. When referring to greater or lesser payloads, it is meant greater or lesser in magnitude (distance from zero), not numerical value. For example, -38 would be greater than -4 because it is farther from zero, even though it is a lesser value.

The ternary27 compatible order predicate maintains strict adherence to binary32 total order by:

Ignoring the payload values of infinities.
Treating positive and unsigned infinites as identical.
Treating all three real number type codes as being identical.
Treating +0 (rounded down) and 0 (infinite precision) as identical.

Comparison of total order sequences		
Binary32 Total Order	**Ternary27 Compatible Order**	**Ternary27 Total Order**
+ Q NaN, Greater Payload	+ Or Unsigned Q NaN, Greater Payload	+ Q NaN, Greater Payload
+ Q NaN, Lesser Payload	+ Or Unsigned Q NaN, Lesser Payload	+ Q NaN, Lesser Payload
N/A	N/A	Unsigned Q NaN, Greater Payload
		Unsigned Q NaN, Lesser Payload
+ S NaN, Greater Payload	+ Or Unsigned S NaN, Greater Payload	+ S NaN, Greater Payload
+ S NaN, Lesser Payload	+ Or Unsigned S NaN, Lesser Payload	+ S NaN, Lesser Payload
N/A	N/A	Unsigned S NaN, Greater Payload
		Unsigned S NaN, Lesser Payload
+ ∞	+ Or Unsigned ∞	+ ∞, Greater Payload
		+ ∞, Lesser Payload
N/A	N/A	Unsigned ∞, Greater Payload
		Unsigned ∞, Lesser Payload
Positive Numbers	Positive Numbers	Positive Numbers Rounded Down
		Positive Numbers Infinite Precision
		Positive Numbers Rounded Up
0	+ 0 or 0	+ 0 (Zero Rounded Down)
		0 (Zero Infinite Precision)
- 0	- 0	- 0 **(**Zero Rounded Up**)**
Negative Numbers	Negative Numbers	Negative Numbers Rounded Up
		Negative Numbers Infinite Precision
		Negative Numbers Rounded Down
- ∞	- ∞	- ∞, Lesser Payload
		- ∞, Greater Payload
- S NaN, Lesser Payload	- S NaN, Lesser Payload	- S NaN, Lesser Payload
- S NaN, Greater Payload	- S NaN, Greater Payload	- S NaN, Greater Payload
- Q Nan, Lesser Payload	- Q Nan, Lesser Payload	- Q Nan, Lesser Payload
- Q Nan, Greater Payload	- Q Nan, Greater Payload	- Q Nan, Greater Payload

19. Infinities

The default behavior of ternary27 is to follow the affinely extended real number system in which -∞ < (any real number) < +∞. The exponent and significand fields serve as the payload. The sign trit is not part of the payload. If the sign trit is zero then the infinity is unsigned.

In order to adhere to the ordinarily expected behavior of floating point math, unsigned infinity is treated in every way exactly as if it were +∞. Only the total order predicate distinguishes unsigned from positive infinity. The standard comparison operations given in section 26 below equate them.

+∞ results when an operation causes a positive overflow and -∞ results when an operation causes a negative overflow. The result of an operation on one or more infinities is defined by the operation. For example, (x + -∞), where x is a real number, is a valid operation and simply produces another negative infinity. On the other hand, (-∞ + -∞) is not valid in the extended real number system. It signals invalid and returns a quiet NaN. Infinities also result from divisions by zero and some other operations listed in IEEE 754 that cause a divide-by-zero exception. The sign of the infinity is determined by the rules of the operation that caused it.

When infinity results from division by unsigned zero, the result is an unsigned infinity. The value of this is the ability to use the total order predicate, the sign flag, or the sign of the infinity to distinguished between a “true” infinity and an infinity caused by overflow. An overflow infinity (-∞ or +∞) is actually a real number that is too large to be represented by the format, whereas division by unsigned zero causes a true infinity because a result cannot be computed. Since -0 and +0 are actually real numbers that are too small to represent in the format, division by signed zeros causes an overflow and thus, the infinity produced is given the appropriate sign. In order to maintain compatibility with ordinarily expected behavior, division by unsigned zero still sets the range flag to + to indicate overflow.

An unsigned infinity, along with the means to fully distinguish -0, unsigned 0 and +0 allows for comparatively easy extensions to the real number system or the implementation of other systems such as the projectively extended real number line. Such extensions must be done through the use of additional operations defined by the implementation because all operations defined in this standard follow the affinely extended real line.

20. Quiet NaNs

Quiet NaN’s are caused by operations that have invalid results and cannot return a real number or an infinity. Quiet NaNs propagate through operations without signaling exceptions. The purpose of their payloads is to carry data that helps diagnose the cause of the NaN. Like infinities and signaling NaNs, the sign trit is not part of the payload and serves as the “sign” of the NaN. It may be negative, positive, or unsigned.

21. Signaling NaNs

Signaling NaNs are used as special floating point datums that will signal invalid when used as an operand. They are produced explicitly and do not result from any mathematical operation. Like infinities and quiet NaNs, the sign trit is not part of the payload and serves as the “sign” of the NaN. It may be negative, positive, or unsigned.

Signaling NaNs are also used to represent variables that have been instantiated but not initialized. It is recommended that they have a sign of zero and a payload of all-zero's, but this is determined by the implementation.

Copying a signaling NaN produces an identical copy. It does not signal invalid.

22. Operations Overview

Operations fall into six categories; arithmetic operations, utility operations, conversion operations, comparison operations, flag operations and rounding mode operations. The operations listed are a suggested set informed by IEEE 854 and IEEE 754, but do not preclude additional operations implemented by languages or libraries. Flag operations are given in section 16 above and rounding operations are described in section 10 above.

Many of the names of operations are examples as found in IEEE 754 and are given in the form of functions. Operations given here that are not found in IEEE 754 have similar naming conventions. Actual function names and operation symbols may vary by implementation. These are merely examples.

23. Arithmetic Operations

The following arithmetic operations are required for compliance with IEEE 854: addition, subtraction, multiplication, division, remainder, fusedMultiplyAdd and square root.

Remainder is defined by the following rules:

The operands are floating point numbers x and y.
The integer n is the integer closest to the value of x/y.
The integer r is the remainder of x REM y.
When y != 0, r is found according to the formula r = x – y*n.
When y = 0 (any sign), the invalid flag is set to – (invalid operation) and a quiet NaN is returned. The payload is the same as x if not otherwise specified by the implementation.

FusedMultiplyAdd(x,y,z) is computed as (x * y) + z. The multiplication occurs first, following ordinary arithmetic laws. The intermediate product is not rounded, but is immediately added to z with guard trits intact. Rounding only occurs after the addition operation. If z is a quiet NaN then it is at the discretion of the implementation whether or not to signal an invalid operation and set the validity flag to -.

The square root operation SQRT(x) returns a positive answer whenever x >= 0. When x = -0, square root returns the value -0. The square root of any negative number other than -0 sets the invalid flag to - (invalid operation) and returns a quiet NaN. The payload is the same as x if not otherwise specified by the implementation.

24. Utility Operations

Utility operations include various non-arithmetic operations and non-comparison predicates. Some operations are:

- roundToWhole(x)
  returns the nearest whole floating point number.
- roundToWholeTowardNegative(x)
  returns the nearest whole number closer to -∞.
- roundToWholeTowardPositive(x)
  returns the nearest whole number closer to +∞.
- roundToWholeTowardZero(x)
  returns the nearest whole number closer to 0.
- nextUp(x)
  returns the next floating point number closer to +∞. Returns x if x is a qNaN and sets the validity flag to -. If the next value up is +∞, nextUp will return +∞ and set the range flag to +.
- nextDown(x)
  returns the next floating point number closer to -∞. Returns x if x is a qNaN and sets the validity flag to -. If the next value down is -∞, nextDown will return -∞ and set the range flag to +.
- nextToward(x,y)
  returns the next floating point number from x which is closer to y. Returns x without signaling an exception if x and y are equal. If x or y is a qNaN, the result is x and the validity flag is set to -. If y is an infinity and is also the next representable datum, then y is returned and the range flag is set to +.
- minNum(x,y)
  returns the operand x or y that is closer to -∞.
- maxNum(x,y)
  returns the operand x or y that is closer to +∞.
- minNumMag(x,y)
  returns the operand x or y that has the lesser magnitude (closer to 0, regardless of sign).
- maxNumMag(x,y)
  returns the operand x or y that has the greater magnitude (farther from 0, regardless of sign).
- copy(x)
  returns the operand exactly.
- negate(x)
  returns the operand with the significand and sign trit inverted. This has the same effect as changing the sign of the number.
- abs(x)
  returns the operand with the significand and sign trit inverted if it was a negative number. Otherwise it returns the operand unchanged.
- copySign(x,y)
  returns the operand x with the significand and sign trit inverted if it does not have the same sign as y. Otherwise it returns the operand x unchanged.
-copyType(x,y)
  returns the operand x with the type code y.
-type(x,type code)
  returns the operand x with the given type code.
- payload(x)
  returns the payload in the form of an integer.
- setPayload(x,y)
  sets the payload of x to integer value y. Signals invalid and returns a quiet NaN when x is not a type having a payload or if y is too large to fit in the payload field.

The four roundToWhole operations return a quiet NaN for any NaN operand and signal invalid when the NaN operand is signaling. They return an infinity of the same sign when the operand is an infinity and return a zero of the same sign when the operand is a zero. They set the exactness flag to – or + as appropriate if the result differs from the operand.

Copy, negate, abs, and copySign do not set flags or signal exceptions.

The following predicates are required and do not signal exceptions:

- compatibleOrder(x,y)
  returns -, 0 or + as described in section 18 above.
- compatibleOrderMag(abs(x), abs(y))
  returns -, 0 or + as described in section 18 above, treating x and y as if they are both positive.
- totalOrder(x,y)
  returns -, 0 or + as described in section 18 above.
- totalOrderMag(abs(x), abs(y))
  returns -, 0 or + as described in section 18 above, treating x and y as if they are both positive.
- class(x)
  returns a string giving which of the nineteen classes the operand x belongs to:
    positive quiet NaN
    unsigned quiet NaN
    negative quiet NaN
    positive signaling NaN
    unsigned signaling NaN
    negative signaling NaN
    positive infinity
    unsigned infinity
    negative infinity
    positive normal number
    negative normal number
    positive non-zero subnormal number
    negative non-zero subnormal number
    positive zero
    unsigned zero
    negative zero
    positive reserved type code
    unsigned reserved type code
    negative reserved type code
  all nineteen classes can be determined by nothing more than the type code and sign.
- sign(x)
  returns – if the datum is negative (including -0), 0 if it is unsigned zero and + if it is positive (including +0).
- sign0(x)
  returns – if the datum is negative and non-zero, 0 if it is zero (regardless of sign) and + if it is positive and non-zero.
- range(x)

returns – if the datum is subnormal, 0 if it is normal and + if it is an infinity or a NaN.

- finiteness(x)
  returns – if the datum is a NaN, 0 if it is normal or subnormal(including 0) and + if it is an infinity.
- nan(x)
  returns – if the datum is a signaling NaN, 0 if it is not a NaN, and + if it is a quiet NaN.
- comp(x,y)
  returns – if x < y, 0 if x == y and + if x > y. Equivalent to the “spaceship operator”, <==>.

25. Conversion Operations

It is required that operations exist for conversion of floating point numbers to and from balanced ternary integers and decimal strings.

Conversions between floating point numbers and integers shall be correctly rounded for the rounding mode in use and always set the exactness flag to – or + as appropriate where the result differs from the operand. When overflow, infinities, or NaNs prevent correct representation as an integer, the validity flag is set to – as an invalid operation and a quiet NaN is produced.

Exact conversions from floating point numbers to decimal character sequences must be available. Conversions from decimal character sequences to floating point numbers that recover the exact value of the original number must be available. Where a decimal character sequence cannot be exactly represented in ternary27, correct rounding is to be employed and flags set appropriately.

The sign of all datums including infinities, NaNs and zeros must be kept during conversions from character sequences if the sign is given. If a character sequence representing a NaN, zero or infinity is unsigned it is unsigned when converted to a floating point datum. If a character sequence representing a non-zero number is unsigned, it is positive when converted to a floating point number.

When converting a quiet or signaling NaN to a character sequence, the default behavior must be to keep the NaN type (quiet or signaling). When converting from a NaN character sequence which does not define the NaN type into a floating point datum it shall be interpreted as a quiet NaN.

Conversion From Character Sequences	
Character Sequence	Ternary27 Datum
+INF or +Infinity	+ ∞
+qNaN or +NaN	+ Q NaN
+sNaN	+ S NaN
+0.0	+ 0.0
INF or Infinity	∞
qNaN or NaN	Q NaN
sNaN	S NaN
0.0	0.0
-INF or -Infinity	- ∞
-qNaN or -NaN	- Q NaN
-sNaN	- S NaN
-0.0	- 0.0

The destination character code (i.e. ASCII or Unicode) used for decimal character sequences is determined by the implementation. The character sequences generated by or accepted by a ternary27 implementation shall be case insensitive, but the examples given above are the preferred outputs for conversions to a character sequence. Attempting to convert from an unrecognized character string shall cause an invalid operation exception.

Implementations may provide an option to preserve the payloads of NaNs and infinities when converting to an external character sequence but this does not have to be the default behavior. NaN or infinity character sequences followed by an integer decimal number within the valid range of a 24-trit payload must preserve this payload when converting to a floating point datum. The valid range is from -141,214,768,240 to +141,214,768,240. An example would take the form of: -qNaN 4339429, which would be interpreted as a quiet NaN with a – in the sign trit and an integer payload of positive 4,339,429 (+ 0 -, 0 + +, + + +, - - +, - - +) [Leading 0's not shown]. Another example is: +INF -39205, which would be interpreted as an infinity with a + in the sign trit and an integer payload of -39,205 (- +, 0 0 0, + - 0, 0 0 -) [Leading 0's not shown].

26. Comparison Operations

Comparison operations include the commonplace comparisons, negated versions and quiet versions. Unlike the order predicates, comparison operations signal exceptions as appropriate. There are both signaling and quiet versions of each comparison operation. The difference is that signaling comparisons will signal invalid even on quiet NaNs whereas quiet comparisons only signal invalid on signaling NaNs.

Each comparison operation returns a ternary value representing one of the following: - for false, + for true, or 0 for unordered. Unordered occurs only when one or both of the operands is a NaN. Every NaN compares unordered with every floating point datum including itself. Comparisons ignore signed zeros such that -0 == 0 == +0. Infinity operands with the same sign compare as equal and -∞ < (any real number) < +∞. Comparison operations treat unsigned ∞ as +∞. Implementations may have additional operations that treat unsigned infinity as unordered.

In the following table, the symbol '!' indicates the inversion of a comparison and the symbol '~' indicates the quiet version of a comparison. These are used as prefixes to the comparison operator and may be used in any combination. For example; '>=' indicates greater than or equal to, '!>=' indicates not greater than or equal to, '~!>=' indicates not greater than or equal to and that quiet NaN operators will not signal an invalid exception, and finally '~>= indicates greater than or equal and that quiet NaN operators will not signal an invalid exception.

Comparison	Relations			Invalid On Quiet NaN
	Greater Than	Less Than	Equal	
==	-	-	+	Yes
!=	+	+	-	Yes
>	+	-	-	Yes
!>	-	+	+	Yes
>=	+	-	+	Yes
!>=	-	+	-	Yes
<	-	+	-	Yes
!<	+	-	+	Yes
<=	-	+	+	Yes
!<=	+	-	-	Yes
~=	-	-	+	No
~!=	+	+	-	No
~>	+	-	-	No
~!>	-	+	+	No
~>=	+	-	+	No
~!>=	-	+	-	No
~<	-	+	-	No
~!<	+	-	+	No
~<=	-	+	+	No
~!<=	+	-	-	No

IEEE 754 defines twenty-two comparisons but there are signaling versions of only twelve leaving an incomplete set. IEEE 854 defines twenty-six comparisons by adding some variations of operations which return true on unordered operands. Operations that specifically test for unordered are unnecessary in ternary27 because every operation returns a ternary value of 0 when either or both operands are unordered. This reduces the number of necessary operations.

The final count of ternary27 comparison operations comes to twenty. There are the five basic comparisons; <, >, <=, >=, ==. Then there are the inverted versions of each making ten operations. Each of these ten has a quiet version making twenty total. Each one simultaneously tests for unordered. This makes a complete set of all possible comparisons with fewer operations than either IEEE 754 or 854, both of which define incomplete sets.

27. Default Exceptions

Exceptions correspond to status flags and indicate the following conditions:

- Exactness flag: inexact, rounded up (-) or, inexact, rounded down (+)
- Range flag: underflow (-) or, overflow (+)
- Validity flag: invalid operation (-) or, signaling NaN operand (+)
- Computability flag: divide-by-zero (-) or, other infinite result from finite operands (+)

Status flags may be set or reset (set to a value of 0) without signaling an exception. Whenever an operation causes one of the above conditions, the default behavior is to return a standard result for the exception type, set the appropriate flag(s) and continue execution. If a trap has been set by the user and/or implementation, then the behavior is to set the appropriate flag(s) and take the trap.

There is a hierarchy of precedence that determines what the default behavior will be when multiple exceptions happen in combination. In these circumstances all appropriate flags for each exception are set but the returned result will be that of the highest priority exception. This ordering is based on the precedence used in Intel FPUs[6] and is therefore a de-facto standard.

Priority when multiple exceptions occur simultaneously:

1. Signaling or quiet NaN passed as operand(s)
2. Any invalid exception other than in 1 above
3. Any uncomputable exception
4. Underflow or overflow
5. Inexact

An invalid exception occurs when one or both operands of an operation is invalid for that operation. A quiet NaN is the default result for any operation that signals invalid. The payload of the NaN should provide diagnostic data but this is not required. The operations that signal invalid are:

1. Any operation on one or more signaling NaNs
2. Addition of infinities with opposite signs
3. Subtraction of infinities with the same sign
4. Addition or subtraction where one or both operands are unsigned infinity
5. Multiplying any 0 by any infinity
6. Dividing any 0 by any 0
7. Dividing any ∞ by any 0
8. Dividing any ∞ by any ∞
9. Remainder of 0 REM ∞ where the operands are either signed or unsigned
10. Square root of a negative number
11. Converting any infinity or NaN into an integer
12. Converting any value into a format that cannot represent a value that large (overflow on conversion)
13. Converting from an unrecognized input character sequence
14. Any of the signaling comparisons on a quiet NaN

Computability exceptions occur and the computability flag is set to - when a division operation is attempted where the divisor is zero (any sign) and the dividend is a finite nonzero number. The result is an infinity with a sign correctly determined using ordinary division rules (when both operands have the same sign, the result is positive. When they have opposite signs, the result is negative). When the dividend is not a finite nonzero number (such as zero or an infinity), the invalid exception is signaled and a quiet NaN is produced.

IEEE 754 defines several operations other than division where a divide-by-zero exception is produced, such as log(0). Some of these even signal divide-by-zero when neither of the operands is zero, such as atanh(±1). To maintain consistency with expectations, the computability exception shall be signaled and the computability flag set to + for each of the non-division operations listed in IEEE 754 that cause a divide-by-zero exception. For operations not given in this document or in IEEE 754, the use of the computability exception is at the discretion of the implementation. In these cases, operations may only set the computability flag to +. The – value is reserved for dividing a finite nonzero number by zero only.

Underflowed results produce a subnormal number, set the range flag appropriately and operation continues normally. If the resulting subnormal number is also inexact, then the exactness flag is also set appropriately.

Overflowed results produce a floating point number or infinity as defined by the rounding mode. They set the range and exactness flags appropriately and the operation continues normally. By definition, all overflows are also inexact and set the exactness flag appropriately.

Inexact results produce a rounded floating point number or, if overflowed, an infinity. They set the exactness flag and type code appropriately and program flow continues normally.

28. Traps

    The use of traps is permitted at the discretion of the implementation. Any trap handling function must be set explicitly and may not be the default behavior for exceptions.

29. Reserved Type Codes

    I have explored many different possible uses for the three unused type codes but have so far rejected each idea.

    At one point I had the idea to use one of the type codes as a NULL value that would be the one and only value permitted for instantiated but uninitialized variables. I decided against this because each type code has 25 trits behind it which equates to a little over 847 billion possible values. It would be wasteful to throw all of that potential away in service to an occasionally needed operand that is adequately handled with signaling NaNs. Signaling NaNs already don't do much so I figured they could continue serving as the NULL value, as is already common practice. Further, some languages don't permit uninitialized variables, which would waste the type code entirely.

    An idea I briefly explored was to use a type code to distinguish between countable and uncountable infinities. I rejected this simply on the basis that it would be used so rarely.

    In an earlier revision I thought to use different type codes for negative and positive NaNs and negative and positive infinities. This turned out to add a great deal of complexity because the sign of some floating point datums was determined by the sign trit or most significant non-zero trit in the significand, but the sign of other datums was determined by the type code. This made calculating the sign flag ugly and would have also forced every single operation to check the type code, sign trit and the most significant non-zero trit of the significand before being able to determine the sign of the result. It was a bit of a kludge.

    There's an idea that I was briefly enamored with after studying some of the *alternative* IEEE 754 proposals (the ones that didn't get selected and approved by the IEEE). There was a proposal from two Hewlett-Packard engineers named Robert Fraley and J. Stephen Walther which became known as the FW proposal[7]. One of the ideas suggested in their proposal was that overflow would have its own special operand. It occurred to me that it could be quite useful in some contexts to be able to distinguish between true infinities and infinities arising merely from overflow. After all, overflow just means "a finite value too big to represent in this format"

while true infinity isn't a finite value at all. They are two different things. This feature would correspond neatly to the ability to distinguish between true zero and zeros arising from extreme underflow. I thought to use one type code as a special overflow operand that would behave identically to the infinity operand, but would be distinguishable as an overflow infinity by its type code. This would maintain the same behavior users have come to expect from overflow events. I did not pursue this as I eventually worked out distinguishing true infinities from overflow infinities using the sign trit. Therefore, the benefits that Fraley and Walther envisioned are available in the ternary27 standard without the need for one of the unused type codes. This is described in section 19 above.

The idea that I am leaning toward at the moment is to have the three additional type codes be imaginary counterparts to the three real number type codes. This is aesthetically pleasing, but may not be of very much use. Imaginary numbers aren't commonly used except in the context of a complex number which requires both an imaginary and a real component. Existing complex number data types just use two floating point numbers with one being the real component and the other being the imaginary. Having type codes for imaginary numbers would create a new class of bug for people dealing with complex numbers, which is that a complex data type could be given two real components or two imaginary components as operands. Nonetheless, it would be quite interesting to have an a class of operations that natively worked with imaginary numbers and real numbers seamlessly. Taking the square root of a negative number wouldn't signal an exception, it would just return a correctly computed imaginary number. You could have operations that "rotated" and "translated" and otherwise manipulated values across the entire "number plane" instead of just the "number line". Unfortunately, this would also introduce a discontinuity in that there would not be imaginary infinities. The three infinities would have to serve both imaginary and real numbers and would be agnostic as to what "kind" of number they represented. In the case of the true infinity (unsigned infinity) this might be perfectly appropriate. But in the case of the "overflow" infinities (plus and minus infinity), overflowing would lose the data about what kind of number it had been unless the implementation chose to maintain that information in the payload.

Taking the above into account, I simply decided to leave the three unused type codes reserved for the time being. There are plenty of potential uses that these could be put to, but I don't want to make that decision until I've found something that is useful and doesn't break the consistency of behavior that the standard has so far maintained.

30. Glossary

Computability Flag: A flag that is set to indicate that an operation on finite operands resulted in an infinity. The flag is – to indicate an actual division by zero in accordance with IEEE 854-1987 where the dividend is "a finite nonzero number". The flag is + to indicate any other operation/finite operand combination that causes an infinite result, such as the many listed in IEEE 754-2008. The flag is 0 to indicate that an infinity has not been caused by an operation on finite operands. Note that even invalid operations that produce a NaN will leave the computability flag at 0.

eMax: The maximum possible exponent value, 121.

eMin: The minimum possible exponent value, -121.

Exactness Flag: A flag that is set to indicate whether any level of precision has been lost in the course of performing an operation or rounding. The flag is + if the most significant non-zero trit lost was a + which indicates the number was rounded down. The flag is - if the most significant non-zero trit lost was a – which indicates the number was rounded up. The flag is 0 if no rounding occurred and indicates infinite precision. The flag values may appear to be backwards, but if a number is rounded down, then it is actually larger than represented, thus the flag is + to signify the returned value has a greater true value. The same is true for – indicating a number that has been rounded up because it is actually smaller than is being represented.

Exception: Any of several signaled conditions that can arise as a result of an operation. The exceptions are “inexact” (rounded up or rounded down), “underflow”, “overflow”, “uncomputable” and “invalid”. Any exception produces a default result, sets the appropriate flag or flags and continues program flow. If a trap is set for that exception, then the trap is executed.

Exponent: The part of the ternary27 number that represents the power to which the radix is being raised. The exponent is an integer between -121 and 121. The radix is the integer 3. Therefore a balanced ternary floating point number can be described as s*(b^e) where s is the value of the significand including the sign trit, b is the base (radix), and e is the exponent.

Exponent Complement (or Complementary Exponent): When the sign trit is -, indicating a negative number, the five exponent trits are complemented. This means 0’s remain the same but -’s and +’s are reversed. This is akin to a form of exponent offset as found in the IEEE 754 standard. The reason for complementing the exponent of negative numbers is so that the most negative representable number has a bit pattern of all -’s from the sign trit to the end, zero has a bit pattern of all 0’s from the sign trit to the end, and the most positive representable number has a bit pattern of all +’s from the sign trit to the end. The exponent in the most negative representable number is large in magnitude, but must be shown in the bit pattern as having a very low value (five – trits in a row). By complementing the exponent trits of negative numbers, the most negative number (lowest value, but large in scale) has the lowest representation and the most positive number has the highest representation.

Extended Significand: Subnormal numbers are implicitly understood to have an exponent of eMin (-121). Therefore, the exponent field is unnecessary and the significand is considered to occupy the 19-trit significand field and the adjacent 5-trit exponent field. Thus, subnormal numbers have an extended significand giving them several orders of magnitude more precision than normal numbers.

Floating Point Datum: A floating point number or non-number that can be represented in the ternary27 format. This includes zeros, real numbers, infinities and NaNs.

Floating Point Number: A finite real number or an infinity that can be represented in the ternary27 format. This includes real numbers, zeros and infinities, but not NaNs. Only floating point datums with real number type codes or infinity type code are floating point numbers.

Gap: The difference between one representable floating point number and the next. The gap is smaller with lower exponents and larger with higher exponents.

Infinite Precision: When the actual result of an operation is exactly the same as the result would have been if the format had a significand of infinite size. An Infinite Precision result required no

rounding and did not lose any precision in conversion between exponents. The exactness flag would be 0.

Infinity: A floating point number with the type code for infinity. This includes infinities caused by arithmetic operations such as division by zero, as well as infinities caused by negative or positive exponent overflow.

NaN: Not-a-Number. Either of two types representing results that are invalid or undefined. For example, dividing infinity by infinity or taking the square root of a negative number. The two types are “quiet NaNs”, which propagate through operations without signaling exceptions, and “signaling NaNs” which signal invalid whenever they are used as operands. Signaling NaNs are also used for instantiated but uninitialized ternary27 variables. Signaling NaNs can only be directly assigned but quiet NaNs can be the result of an operation.

Normal Number, Normalized Number: A number stored or represented in normal form. By normal form is meant that the sign trit is non-zero. This indicates that there is one, and only one non-zero digit to the left of the radix point.

Payload: The data stored in the significand and exponent fields of a NaN or infinity. Mathematical operations presented in this standard do not affect payloads because a NaN or infinity operand cannot produce a result other than another infinity or NaN (depending on the operation) and the purpose of a payload is to propagate through operations unchanged. A payload may be treated as a 24-trit balanced-ternary integer. Payloads do not include the sign trit which serves as the sign for NaNs and infinities, primarily for the purposes of the totalOrder predicate.

Predicate: An operation that returns the relative truthfulnesses of an assertion. Conditional operations are one set of predicates because they assert a condition such as x > y, and return a value telling you whether or not this is true. For the purpose of this standard, a predicate is an operation that takes one or two floating point datums and returns a ternary value (-, 0 or +) similar to a binary boolean value.

Range Flag: A flag that is set to indicate whether the operation has resulted in a subnormal number (underflow) or a number with a magnitude so great that it would require an exponent larger than eMax to represent it (overflow). It is set to – for an underflow, + for an overflow, and 0 for neither.

Real Number: Any floating point datum with one of the three type codes for real numbers representable by this format. The three types are “real number rounded down”, “real number infinite precision” and “real number rounded up”. Real numbers do not include infinities.

Sign Flag: A flag that is set to indicate whether the result of an operation is positive, negative, or zero. It is – to indicate a negative value, is a 0 to indicate the value zero, and is a + to indicate a positive value. It is set to 0 for both signed and unsigned zeros.

Sign Trit: The part of a ternary27 number that represents the sign of the number. Being a balanced ternary digit, it can be -1, 0, or 1. If it is -1 or 1 (- or +) then the value is a normal number. If the sign trit is 0, then the number is subnormal and the exponent is assumed to be

eMin. In all cases, the sign trit is the digit directly to the left of the radix point, even it its value is 0.

Significand: The part of the ternary27 format that represents the digits to the left of the radix point. For normal numbers, this is a 19-trit wide field. For subnormal numbers, this is 24-trits wide and includes the exponent field which is unused because subnormal numbers are implicitly assumed to have an exponent of eMin.

Subnormal Number: A number not stored or represented in normal form. The sign trit is set to 0. This occurs when a result is very close to zero and sets the range flag to -. A subnormal number is implicitly treated as having the lowest possible exponent (-121) and the unused exponent field becomes part of the extended significand to grant additional precision.

Type: Any of the “type” codes for a floating point datum including real numbers, infinities, signaling NaNs and quiet NaNs. There are also three reserved type codes not assigned at this time which are not used and, if encountered, are to be treated as signaling NaNs.

Validity Flag: A flag that is set to indicate whether an operation is valid or not. A – indicates an invalid operation in which a quiet NaN is produced. A 0 indicates an ordinary valid operation and that a floating point number was produced. A + indicates that a signaling NaN was passed to an operation as an operand and that a quiet NaN was produced.

31. Citations

1. Hackaday.io project, Shared Silicon, https://hackaday.io/project/11779-shared-silicon
2. IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic
3. ANSI/IEEE Std 854-1987, IEEE Standard for Radix-Independent Floating-Point Arithmetic
4. An Interview with the Old Man of Floating-Point, Section: Early in the Process
5. The Art of Computer Programming, Third Edition, page 207
6. Intel Architecture Software Developer’s Manual, Chapter 31, Section 31.8.7
7. Analysis of Proposals for the Floating-Point Standard, W.J. Cody, March 1981

Written by [Mechanical Advantage]