IEEE 754, FPUs and Other Animals

Lucy Wayland

Cambridge MiniDebconf

November 2015

Overview

Background

Fixed Point

Floating Point

Special Cases

Exceptions

Appendix

History of IEEE 754 (1/2)

- 1960's and early 1970's, floating point hardware restricted to minicomputers and mainframes

History of IEEE 754 (1/2)

- 1960's and early 1970's, floating point hardware restricted to minicomputers and mainframes
- Every manufacturer had their own, incompatible, quirky implementation. Mid 1970s the start of the rise of the microprocessor, and designs for floating point units for these started

History of IEEE 754 (1/2)

- 1960's and early 1970's, floating point hardware restricted to minicomputers and mainframes
- Every manufacturer had their own, incompatible, quirky implementation. Mid 1970s the start of the rise of the microprocessor, and designs for floating point units for these started
- Concerned with an explosion in individual FPUs, some visionaries came together to produce a common standard

History of IEEE 754 (1/2)

- 1960's and early 1970's, floating point hardware restricted to minicomputers and mainframes
- Every manufacturer had their own, incompatible, quirky implementation. Mid 1970s the start of the rise of the microprocessor, and designs for floating point units for these started
- Concerned with an explosion in individual FPUs, some visionaries came together to produce a common standard
- IEEE p754 committee created, with most of the big industry players taking part

History of IEEE 754 (1/2)

- 1960's and early 1970's, floating point hardware restricted to minicomputers and mainframes
- Every manufacturer had their own, incompatible, quirky implementation. Mid 1970s the start of the rise of the microprocessor, and designs for floating point units for these started
- Concerned with an explosion in individual FPUs, some visionaries came together to produce a common standard
- IEEE p754 committee created, with most of the big industry players taking part
- MATLAB's creator Dr. Cleve Moler used to advise foreign visitors not to miss the US's two most awesome spectacles: the Grand Canyon, and meetings of IEEE p754

History of IEEE 754 (2/2)

- Compromises, technical reports and eventually a showdown later, IEEE p754 produced

History of IEEE 754 (2/2)

- Compromises, technical reports and eventually a showdown later, IEEE p754 produced
- Not published as a standard until 1985 (IEEE 754-1985), but by then most manufacturers had implemented it

History of IEEE 754 (2/2)

- Compromises, technical reports and eventually a showdown later, IEEE p754 produced
- Not published as a standard until 1985 (IEEE 754-1985), but by then most manufacturers had implemented it
- IEEE 754-1985 had single and double precision binary floating point Updated to IEEE 754-2008, which added quadruple and arbitrary precision binary floating point, 64 and 128 bit decimal floating point, changed some terminology, and other minor changes

History of IEEE 754 (2/2)

- Compromises, technical reports and eventually a showdown later, IEEE p754 produced
- Not published as a standard until 1985 (IEEE 754-1985), but by then most manufacturers had implemented it
- IEEE 754-1985 had single and double precision binary floating point Updated to IEEE 754-2008, which added quadruple and arbitrary precision binary floating point, 64 and 128 bit decimal floating point, changed some terminology, and other minor changes
- Only going to refer to IEEE 754-2008, and ignore decimal floating point as not many FPUs support them.

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:
  - An overflow to negative or positive max full house saturates, and does not wrap

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:
  - An overflow to negative or positive max full house saturates, and does not wrap
  - The placement of the binary point is arbitary - just like in decimal arithmetic

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:
  - An overflow to negative or positive max full house saturates, and does not wrap
  - The placement of the binary point is arbitary - just like in decimal arithmetic
- Imagine an 4 bit fixed point number, which is a straight integer. The binary point is to the right of the least significant bit: $11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 1^1 + 1 \cdot 1^0$

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:
  - An overflow to negative or positive max full house saturates, and does not wrap
  - The placement of the binary point is arbitary - just like in decimal arithmetic
- Imagine an 4 bit fixed point number, which is a straight integer. The binary point is to the right of the least significant bit: $11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 1^1 + 1 \cdot 1^0$
- Now we shift the binary point one bit to the left: $5.5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 1^0 + 1 \cdot 1^{-1}$

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:
  - An overflow to negative or positive max full house saturates, and does not wrap
  - The placement of the binary point is arbitary - just like in decimal arithmetic
- Imagine an 4 bit fixed point number, which is a straight integer. The binary point is to the right of the least significant bit: $11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 1^1 + 1 \cdot 1^0$
- Now we shift the binary point one bit to the left: $5.5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 1^0 + 1 \cdot 1^{-1}$
- The arithmetic is just the same, we just need to keep track of where the binary point is

Fixed Point Arithmetic (1/2)

- To understand floating point, it helps to understand fixed point first
- Fixed point arithmetic is just like normal integer arithmetic, but two important differences:
  - An overflow to negative or positive max full house saturates, and does not wrap
  - The placement of the binary point is arbitary - just like in decimal arithmetic
- Imagine an 4 bit fixed point number, which is a straight integer. The binary point is to the right of the least significant bit: $11 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 1^1 + 1 \cdot 1^0$
- Now we shift the binary point one bit to the left: $5.5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 1^0 + 1 \cdot 1^{-1}$
- The arithmetic is just the same, we just need to keep track of where the binary point is
- Taken to its logical extreme, the binary point is completely to the left: $0.6875 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 1^{-3} + 1 \cdot 1^{-4}$

Fixed Point Arithmetic (2/2)

- Normally fixed point numbers are signed, and use standard 2's complement.

Fixed Point Arithmetic (2/2)

- Normally fixed point numbers are signed, and use standard 2's complement.
- These are normally termed "Q Numbers" (due to the saturation).

Fixed Point Arithmetic (2/2)

- Normally fixed point numbers are signed, and use standard 2's complement.
- These are normally termed "Q Numbers" (due to the saturation).
- The notation is Qx.y, where "x" is the number of bits (not including the sign bit) to the left of the binary point, and "y" is the number of bits to the right.

Fixed Point Arithmetic (2/2)

- Normally fixed point numbers are signed, and use standard 2's complement.
- These are normally termed "Q Numbers" (due to the saturation).
- The notation is Qx.y, where "x" is the number of bits (not including the sign bit) to the left of the binary point, and "y" is the number of bits to the right.
- Convention is that when x=0, it is omitted. E.g. Q15 for a signed 16 bit number, Q31 for 32 etc.

Fixed Point Arithmetic (2/2)

- Normally fixed point numbers are signed, and use standard 2's complement.
- These are normally termed "Q Numbers" (due to the saturation).
- The notation is Qx.y, where "x" is the number of bits (not including the sign bit) to the left of the binary point, and "y" is the number of bits to the right.
- Convention is that when x=0, it is omitted. E.g. Q15 for a signed 16 bit number, Q31 for 32 etc.
- Note that this Q15, Q31 etc. can represent -1.0, but cannot represent +1.0

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.
  - Sign (+ or -)

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.
  - Sign (+ or -)
  - Mantissa, also known as the significand (e.g. 3.1415)

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.
  - Sign (+ or -)
  - Mantissa, also known as the significand (e.g. 3.1415)
  - The exponent, which is the shift of the decimal or binary point, e.g. 10^{-3}

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.
  - Sign (+ or -)
  - Mantissa, also known as the significand (e.g. 3.1415)
  - The exponent, which is the shift of the decimal or binary point, e.g. 10^{-3}
- The mantissa is made up of multiples of decreasing powers of the base, as you work from most signifcant to less significant. For example: $3.1415 = 3 \cdot 10^0 + 1 \cdot 10^{-1}$

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.
  - Sign (+ or -)
  - Mantissa, also known as the significand (e.g. 3.1415)
  - The exponent, which is the shift of the decimal or binary point, e.g. 10^{-3}
- The mantissa is made up of multiples of decreasing powers of the base, as you work from most signifcant to less significant. For example: $3.1415 = 3 \cdot 10^0 + 1 \cdot 10^{-1}$
- Each multiplier is 0 .. (base - 1)

Floating Point Representation (1/2)

- Floating point numbers are made up of one assumption, and three parts:
  - Base - you are assumed you are working in base, usually binary, but sometimes decimal.
  - Sign (+ or -)
  - Mantissa, also known as the significand (e.g. 3.1415)
  - The exponent, which is the shift of the decimal or binary point, e.g. 10^{-3}
- The mantissa is made up of multiples of decreasing powers of the base, as you work from most signifcant to less significant. For example: $3.1415 = 3 \cdot 10^0 + 1 \cdot 10^{-1}$
- Each multiplier is 0 .. (base - 1)
- The exponent is just the base taken to a power

Floating Point Representation (2/2)

- Binary floating point is just just all of this in base 2 (rather like fixed point).

Floating Point Representation (2/2)

- Binary floating point is just just all of this in base 2 (rather like fixed point).
- Sign as before (not two's compliment).

Floating Point Representation (2/2)

- Binary floating point is just just all of this in base 2 (rather like fixed point).
- Sign as before (not two's compliment).
- Mantissa is sum of 1 or 0 times decreasing negative powers of two, starting with 2^0 e.g.: $1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$

Floating Point Representation (2/2)

- Binary floating point is just just all of this in base 2 (rather like fixed point).
- Sign as before (not two's compliment).
- Mantissa is sum of 1 or 0 times decreasing negative powers of two, starting with 2^0 e.g.: $1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$
- Exponent is power of 2 to multiply by.

Floating Point Representation (2/2)

- Binary floating point is just just all of this in base 2 (rather like fixed point).
- Sign as before (not two's compliment).
- Mantissa is sum of 1 or 0 times decreasing negative powers of two, starting with 2^0 e.g.: $1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3}$
- Exponent is power of 2 to multiply by.
- So the sign is a bit, the exponent is a signed number to take the multiplier of 2 by, and each bit in the mantissa is the amount of that decreasing power of two.

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)
  - Think very hard on the balance of precision vs. range for a certain bit length (32, 64 etc.), and come up with a defined length of mantissa and exponent for each

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)
  - Think very hard on the balance of precision vs. range for a certain bit length (32, 64 etc.), and come up with a defined length of mantissa and exponent for each
- This gives us:

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)
  - Think very hard on the balance of precision vs. range for a certain bit length (32, 64 etc.), and come up with a defined length of mantissa and exponent for each
- This gives us:
  - Symmetric range (no more negative than positive numbers)

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)
  - Think very hard on the balance of precision vs. range for a certain bit length (32, 64 etc.), and come up with a defined length of mantissa and exponent for each
- This gives us:
  - Symmetric range (no more negative than positive numbers)
  - A standard format that works on any compliant hardware

IEEE 754 Representation (1/2)

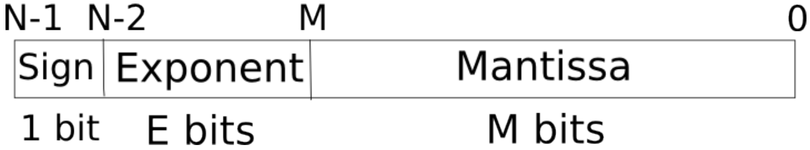
- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)
  - Think very hard on the balance of precision vs. range for a certain bit length (32, 64 etc.), and come up with a defined length of mantissa and exponent for each
- This gives us:
  - Symmetric range (no more negative than positive numbers)
  - A standard format that works on any compliant hardware
  - Fast mathematics (normal arithmetic works on each field)

IEEE 754 Representation (1/2)

- IEEE 754 cleverness:
  - Throw away leading 2^0 term and assume it is always set.
  - Make exponent normalised, so it is unsigned, minus an offset called the exponent bias
  - Make the exponent values of all 0s and all 1s special cases (more on these in a bit)
  - Think very hard on the balance of precision vs. range for a certain bit length (32, 64 etc.), and come up with a defined length of mantissa and exponent for each
- This gives us:
  - Symmetric range (no more negative than positive numbers)
  - A standard format that works on any compliant hardware
  - Fast mathematics (normal arithmetic works on each field)
  - Exact 1, but no zero!

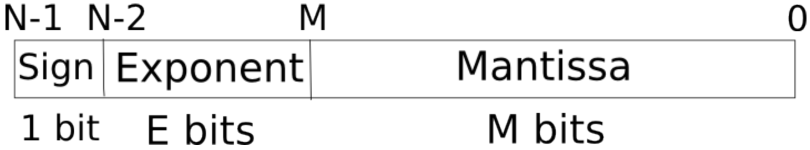
IEEE 754 Representation (2/2)

- All IEEE 754 floating point numbers are of the form:



IEEE 754 Representation (2/2)

- All IEEE 754 floating point numbers are of the form:



- Where N is the width of the format (e.g. 32)
- IEEE 754 defines three binary formats:
  - binary32 (single), N=32, E=8, M=23, bias=127
  - binary64 (double) , N=64, E=11, M=52, bias=1023
  - binary128, N=128, E=15, M=112, bias=16383
- Additionally, under the arbitrary precision, there is often:
  - binary16 (half), N=16, E=5, M=10, bias=15

IEEE 754 Example

- Half precision example 0x3555 = b0011 0101 0101 0101

IEEE 754 Example

- Half precision example 0x3555 = b0011 0101 0101 0101
- Sign = b0
  - Which means it is positive

IEEE 754 Example

- Half precision example 0x3555 = b0011 0101 0101 0101
- Sign = b0
  - Which means it is positive
- Exponent = b01101
  - = 13 normalized with bias of 15 = -2
  - $= 2^{-2} = 0.25$

IEEE 754 Example

- Half precision example 0x3555 = b0011 0101 0101 0101
- Sign = b0
  - Which means it is positive
- Exponent = b01101
  - = 13 normalized with bias of 15 = -2
  - $= 2^{-2} = 0.25$
- Mantissa = b0101010101
  - $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10}$
  - = 0.25 + 0.0625 + 0.015625 + 0.00390625 + 0.0009765625
  - = 0.3330078125
  - With the implicit $2^0 = 1.3330078125$

IEEE 754 Example

- Half precision example 0x3555 = b0011 0101 0101 0101
- Sign = b0
  - Which means it is positive
- Exponent = b01101
  - = 13 normalized with bias of 15 = -2
  - $= 2^{-2} = 0.25$
- Mantissa = b0101010101
  - $= 2^{-2} + 2^{-4} + 2^{-6} + 2^{-8} + 2^{-10}$
  - = 0.25 + 0.0625 + 0.015625 + 0.00390625 + 0.0009765625
  - = 0.3330078125
  - With the implicit 2^0 = 1.3330078125
- Final answer = +1x0.25x1.3330078215 = 0.3332519553725
- Which is about as close to a 1/3 as it can get

Special Cases (1/2)

- Exponent is all 0s:
  - If the mantissa is also all 0s, then the number is 0

Special Cases (1/2)

- Exponent is all 0s:
  - If the mantissa is also all 0s, then the number is 0
  - This gives us signed zeros (+0 and -0). Very important mathematically,

Special Cases (1/2)

- Exponent is all 0s:
  - If the mantissa is also all 0s, then the number is 0
  - This gives us signed zeros (+0 and -0). Very important mathematically,
  - IEEE 754-2008 declares that $(+0 == -0)$ as true, not $(+0 > -0)$

Special Cases (1/2)

- Exponent is all 0s:
  - If the mantissa is also all 0s, then the number is 0
  - This gives us signed zeros (+0 and -0). Very important mathematically,
  - IEEE 754-2008 declares that $(+0 == -0)$ as true, not $(+0 > -0)$
- If the mantissa is non-zero, then the number is "subnormal". Old IEEE 754-1985 terminology was "denormalized", and you may see this in documentation

Special Cases (1/2)

- Exponent is all 0s:
  - If the mantissa is also all 0s, then the number is 0
  - This gives us signed zeros (+0 and -0). Very important mathematically,
  - IEEE 754-2008 declares that $(+0 == -0)$ as true, not $(+0 > -0)$
- If the mantissa is non-zero, then the number is "subnormal". Old IEEE 754-1985 terminology was "denormalized", and you may see this in documentation
  - In this case, the always assumed 1 is actually 0, and the normalized exponent is maximum negative of the precision
  - How an FPU handles subnormal numbers is determined by version of the FPU, state of the control registers, and in some cases IMPLEMENTATION DEFINED. Thankfully, not the phase of the moon

Special Cases (2/2)

- Exponent all 1s:

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$
- If the mantissa is non-zero, then it is Not a Number (NaN)

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$
- If the mantissa is non-zero, then it is Not a Number (NaN)
  - If the top bit of the mantissa is set (the 2^{-1} value), it is a Quiet NaN (qNaN)

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$
- If the mantissa is non-zero, then it is Not a Number (NaN)
  - If the top bit of the mantissa is set (the 2^{-1} value), it is a Quiet NaN (qNaN)
  - If the top bit is clear, it is a Signaling NaN (sNaN). The rest of the bits can be any value EXCEPT all 0s as then it would be an infinity

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$
- If the mantissa is non-zero, then it is Not a Number (NaN)
  - If the top bit of the mantissa is set (the 2^{-1} value), it is a Quiet NaN (qNaN)
  - If the top bit is clear, it is a Signaling NaN (sNaN). The rest of the bits can be any value EXCEPT all 0s as then it would be an infinity
  - The rest of this field can encode information about the NaN and what caused it

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$
- If the mantissa is non-zero, then it is Not a Number (NaN)
  - If the top bit of the mantissa is set (the 2^{-1} value), it is a Quiet NaN (qNaN)
  - If the top bit is clear, it is a Signaling NaN (sNaN). The rest of the bits can be any value EXCEPT all 0s as then it would be an infinity
  - The rest of this field can encode information about the NaN and what caused it
  - NaNs are not signed (sign bit ignored)

Special Cases (2/2)

- Exponent all 1s:
  - If the mantissa is zero, then value is infinity
  - This provides $+\infty$ and $-\infty$
- If the mantissa is non-zero, then it is Not a Number (NaN)
  - If the top bit of the mantissa is set (the 2^{-1} value), it is a Quiet NaN (qNaN)
  - If the top bit is clear, it is a Signaling NaN (sNaN). The rest of the bits can be any value EXCEPT all 0s as then it would be an infinity
  - The rest of this field can encode information about the NaN and what caused it
  - NaNs are not signed (sign bit ignored)
  - By default, all standard IEEE 754 floating point operations which produce NaNs only produce Quiet NaNs

Rounding

- The output of an operation will almost certainly not be precisely represented by a format. Therefore, rounding is required IEEE 754 defines 4 rounding modes for binary floating point:

Rounding

- The output of an operation will almost certainly not be precisely represented by a format. Therefore, rounding is required IEEE 754 defines 4 rounding modes for binary floating point:
  - Round up, towards +infinity
  - Round down, towards -infinity
  - Round towards zero (truncation).
  - Round to nearest, ties to even - calculate the ideal value, and then on a tie, round to the nearest even digit

Rounding

- The output of an operation will almost certainly not be precisely represented by a format. Therefore, rounding is required IEEE 754 defines 4 rounding modes for binary floating point:
  - Round up, towards +infinity
  - Round down, towards -infinity
  - Round towards zero (truncation).
  - Round to nearest, ties to even - calculate the ideal value, and then on a tie, round to the nearest even digit
- Rounds to nearest, ties to even is the default but most computationally expensive mode

Rounding

- The output of an operation will almost certainly not be precisely represented by a format. Therefore, rounding is required IEEE 754 defines 4 rounding modes for binary floating point:
  - Round up, towards +infinity
  - Round down, towards -infinity
  - Round towards zero (truncation).
  - Round to nearest, ties to even - calculate the ideal value, and then on a tie, round to the nearest even digit
- Rounds to nearest, ties to even is the default but most computationally expensive mode
- There is a fifth optional mode, commonly used in decimal, but not in binary floating point - round to nearest, ties away from zero

Exceptions (1/2)

- IEEE 754 defines five exceptions

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN
  - Adding or subtracting when both operands are an infinity

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN
  - Adding or subtracting when both operands are an infinity
  - Dividing 0 by 0, or an infinity by an infinity

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN
  - Adding or subtracting when both operands are an infinity
  - Dividing 0 by 0, or an infinity by an infinity
  - Square root of a negative number

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN
  - Adding or subtracting when both operands are an infinity
  - Dividing 0 by 0, or an infinity by an infinity
  - Square root of a negative number
  - Some other edge cases

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN
  - Adding or subtracting when both operands are an infinity
  - Dividing 0 by 0, or an infinity by an infinity
  - Square root of a negative number
  - Some other edge cases
- Division by zero

Exceptions (1/2)

- IEEE 754 defines five exceptions
- A floating point exception does not necessarily mean a processor exception
- Invalid operation:
  - Performing an operation on a Signalling NaN
  - Adding or subtracting when both operands are an infinity
  - Dividing 0 by 0, or an infinity by an infinity
  - Square root of a negative number
  - Some other edge cases
- Division by zero
  - The output of the operation is the correctly signed infinity

Exceptions (2/2)

- Underflow
  - The output of the operation is non-zero, but smaller than the format can represent. This is called tinniness
  - Output is rounded, which can become zero or subnormal

Exceptions (2/2)

- Underflow
  - The output of the operation is non-zero, but smaller than the format can represent. This is called tinniness
  - Output is rounded, which can become zero or subnormal
- Overflow
  - Output would be finite but bigger than the format can hold
  - Depending on rounding, output can either be biggest finite number of the format, or an infinity

Exceptions (2/2)

- Underflow
  - The output of the operation is non-zero, but smaller than the format can represent. This is called tinniness
  - Output is rounded, which can become zero or subnormal
- Overflow
  - Output would be finite but bigger than the format can hold
  - Depending on rounding, output can either be biggest finite number of the format, or an infinity
- Inexact
  - The output of the operation means that neither the desired mantissa nor desired exponent can be represented by the format
  - Nearly always paired with an Underflow or Overflow

NaNs

- The NaN mantissa encodes information about the NaN

NaNs

- The NaN mantissa encodes information about the NaN
- Any operation that produces an Invalid Operation exception, generates a qNaN

NaNs

- The NaN mantissa encodes information about the NaN
- Any operation that produces an Invalid Operation exception, generates a qNaN
- Any operation that has a qNaN but not an sNaN as an input, shall pass on the input qNaN as the output

NaNs

- The NaN mantissa encodes information about the NaN
- Any operation that produces an Invalid Operation exception, generates a qNaN
- Any operation that has a qNaN but not an sNaN as an input, shall pass on the input qNaN as the output
- Any operation that has a sNan as an input shall signal an Invalid Operation exception

Operations

- The standard defines several operations, and how they should perform:

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate (A = A + B x C)

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate (A = A + B x C)
  - Integer $\Leftrightarrow$ floating point conversions

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate ($A = A + B \times C$)
  - Integer $\Leftrightarrow$ floating point conversions
  - Exponential and log functions

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate ($A = A + B \times C$)
  - Integer $\Leftrightarrow$ floating point conversions
  - Exponential and log functions
  - Trigonometric and hyperbolic functions

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate ($A = A + B \times C$)
  - Integer $\Leftrightarrow$ floating point conversions
  - Exponential and log functions
  - Trigonometric and hyperbolic functions
  - Various support, status and identity functions e.g. isInfinite()

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate (A = A + B x C)
  - Integer ⇔ floating point conversions
  - Exponential and log functions
  - Trigonometric and hyperbolic functions
  - Various support, status and identity functions e.g. isInfinite()
- Note that these do not have to be implemented in hardware

Operations

- The standard defines several operations, and how they should perform:
  - (In)equality comparisons
  - Standard arithmetic operations (+ - / *, power, square root and N root)
  - Fused multiply accumulate (A = A + B x C)
  - Integer $\Leftrightarrow$ floating point conversions
  - Exponential and log functions
  - Trigonometric and hyperbolic functions
  - Various support, status and identity functions e.g. isInfinite()
- Note that these do not have to be implemented in hardware
- atan2 is notorious for its edge conditions, and is classic example of where signed zero is required

Appendix