Notes on IEEE 754 Floating-Point Number Format. Further details and examples can be found at http://weitz.de/ieee/ and https://en.wikipedia.org/wiki/IEEE_754

# float16 (fewer resources, less accuracy)

The float16 library uses the IEEE 754 binary16 format for storing floating-point numbers:

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + - | exponent (+15) | | | | | mantissa (10 bits stored) | | | | | | | | | |

The exponent is stored with a bias of +15, as per the standard.

| EXAMPLES of float16 | | |
|---|---|---|
| Number | Binary Float | float16 hex and binary |
| 0 | 0.0 x 2 ^ 0 | 0000 0000000000000000 |
| 1 | 1.0 x 2 ^ 0 | 3C00 0011110000000000 |
| 2 | 1.0 x 2 ^ 1 | 4000 0100000000000000 |
| 3.14 | 1.57 x 2 ^ 1 | 4248 0100001001001000 |
| -100 | -1.5625 x 2 ^ 6 | D640 1101011001000000 |
| inf | inf | 7C00 0111110000000000 |
| NaN | NaN | FE00 1111111000000000 |

# float32 (more accuracy)

The float32 library uses the IEEE 754 binary32 format for storing floating-point numbers:

| 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 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| + - | exponent (+127) | | | | | | | | mantissa (23 bits stored) | | | | | | | | | | | | | | | | | | | | | | |

The exponent is stored with a bias of +127, as per the standard.

| EXAMPLES of float32 | | |
|---|---|---|
| Number | Binary Float | float32 hex and binary |
| 0 | 0.0 x 2 ^ 0 | 00000000<br>00000000000000000000000000000000 |
| 1 | 1.0 x 2 ^ 0 | 3F800000<br>00111111100000000000000000000000 |
| 2 | 1.0 x 2 ^ 1 | 40000000<br>01000000000000000000000000000000 |
| 3.1415927 | 1.5707964 x 2 ^ 1 | 40490FDB<br>01000000010010010000111111011011 |
| -100 | -1.5625 x 2 ^ 6 | C2C80000<br>11000010110010000000000000000000 |
| inf | inf | 7F800000<br>01111111100000000000000000000000 |
| NaN | NaN | FFC00000<br>11111111110000000000000000000000 |

Each library provides algorithms for conversion between floating-point representation and integers, addition/subtraction, multiplication, division, square root and basic comparisons.

Numbers that are too large to store in the relevant format return the largest possible number in the relevant format, and too small to store, return zero. Errors are indicated by the returned flags, discussed below.

## Usage

Where algorithms have “start” and “busy” signals, hold “start” to 1 for 1 clock, and wait for “busy” to return to 0. “a” represents the first operand, “b” the second operand, and “result” is the result in the appropriate format.

“addsub” and “dounsigned” are explained in the appropriate section when used.

For the comparisons, there are outputs for the result of the two comparisons, 1 for true, 0 for false.

**NOTE:** Due to the same name being used for the algorithms float16 and float32 cannot be used in the same project.

## Flags

All algorithms return a 7 bit flag indicating the error status of the conversion or calculation.

| 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|
| **IF** | **NN** | **NV** | **DZ** | **OF** | **UF** | **NX** |
| INFINITY | NaN | NOT VALID | DIVIDE BY 0 | OVERFLOW | UNDERFLOW | NOT EXACT |
| INF passed as an argument. | NaN passed as an argument. | Result not valid, due to incorrect arguments, ie, comparing NaN, square root of a negative. | Divide by zero attempted. | Result overflowed. The largest possible value is returned as the result. | Result underflowed. Zero is returned as the result. | Not exact, conversion (int to float) lost accuracy. |

## NaN (not a number)

Quiet NaNs are returned to indicate errors, or for NaN propagation. Signalling NaNs are detected, but are treated the same as quiet NaNs.

## VERILATOR Test Suite

https://github.com/rob-ng15/Silice-Playground/tree/master/VERILATOR

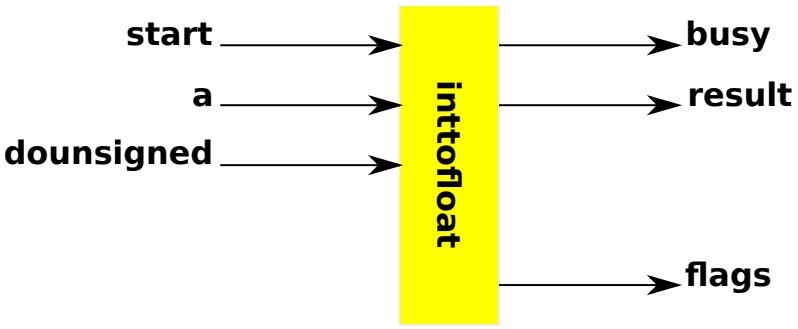
A test suite for use with VERILATOR is available that simulates a Risc-V RV32F floating-point unit. In the algorithm “main” set the Risc-V opCode, function7, function3, rs1 and rs2 as per the decoded floating-point instruction, along with sourceReg1 (the integer register), and sourceReg1F, sourceReg2F and sourceReg3F (the floating-point registers) and execute make RISCV-F32

Verilator will then simulate the instruction and provide output showing the progress, along with intermediate results, along with the final output from the floating-point unit.

NOTE: Risc-V only uses 5 of the 7 available status flags. FRD indicates if the result is to go to a floating-point or integer register.

# inttofloat

Converts integers (16 bit for float16, 32 bit for float32) to floating-point.



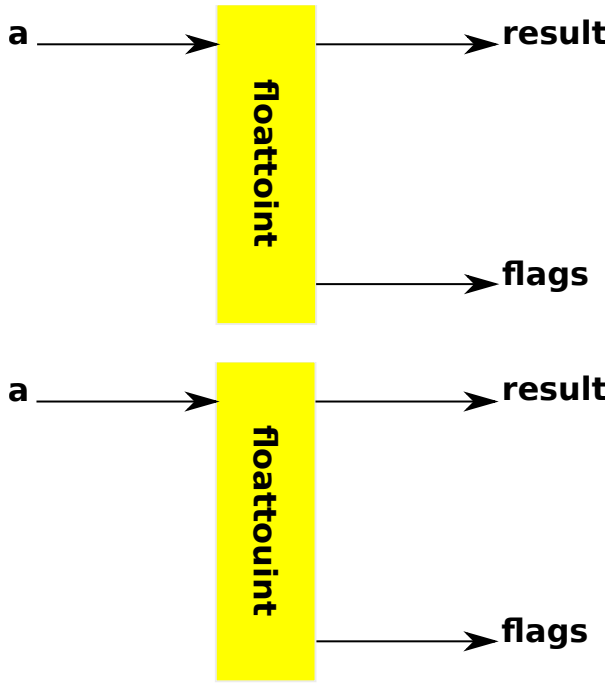

| Signal | Meaning |
|---|---|
| “start” | Start the conversion by holding to 1 for 1 clock cycle. |
| “a” | Integer to convert to floating-point (16 bit for float16, 32 bit for float32). |
| “dounsigned” | Set to 1 to treat a as an unsigned integer. |
| “busy” | Set to 1 whilst the conversion takes place. |
| “result” | Result of the conversion of a into floating-point as float16 or float32. |

### FLAGS

| Flag | Meaning |
|---|---|
| NX | Bits have been dropped due to too few bits in the mantissa. Affects large integers. |

# floattoint and floattouint

Converts floating-point numbers (16 bit for float16, 32 bit for float32) to integers. floattoint converts floating-point numbers to signed integers. floattouint converts floating-point numbers to unsigned integers, with negative numbers returning 0. Numbers that are too large return the maximum integer.



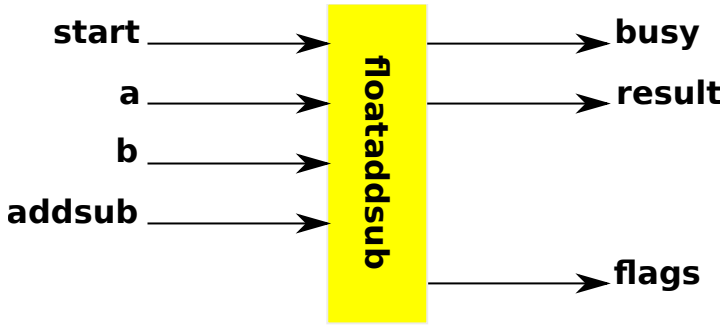

| Signal | Meaning |
|---|---|
| “a” | Floating-point number to convert to an integer. |
| “result” | Result of the conversion of a into an integer. |

## FLAGS

| Flag | Meaning |
|---|---|
| IF | INF as an argument. |
| NN | NaN as an argument. |
| NV | Floating-point number cannot be represented as an integer, such as being too large/small, a negative number for unsigned conversion, NaN or INF. |

# floataddsub

Performs addition or subtraction (16 bit for float16, 32 bit for float32) of two floating-point numbers.



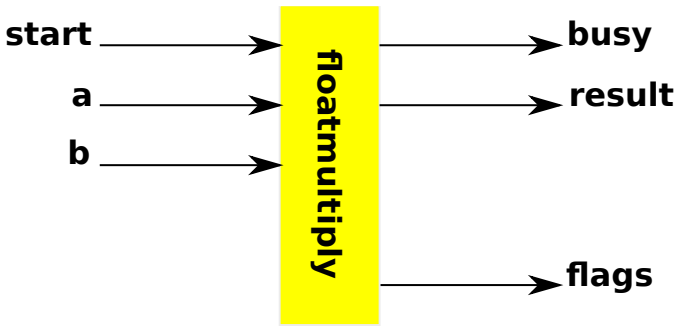

| Signal | Meaning |
|---|---|
| “start” | Start the additiion or subtraction by holding to 1 for 1 clock cycle. |
| “a” | First floating-point operand. |
| “b” | Second floating-point operand. |
| “addsub” | Control when 0 do addition, or 1 do subtraction. |
| “busy” | Set to 1 whilst the operation takes place. |
| “result” | Result of “addsub = 0” a +b, or “addsub = 1” a – b. |

## FLAGS

| Flag | Meaning |
|---|---|
| IF | INF as an argument. |
| NN | NaN as an argument. |
| NV | INF – INF. |
| OF | Result overflowed, INF returned. |
| UF | Result underflowed, zero returned. |

# floatmultiply

Performs multiplication (16 bit for float16, 32 bit for float32) of two floating-point numbers.




| Signal | Meaning |
|---|---|
| “start” | Start the multiplication by holding to 1 for 1 clock cycle. |
| “a” | First floating-point operand. |
| “b” | Second floating-point operand. |
| “busy” | Set to 1 whilst the operation takes place. |
| “result” | Result of a * b. |

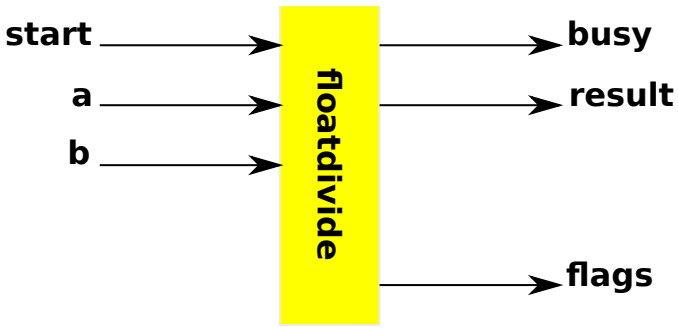
**NOTE:** The multiplication as written expects yosys to infer DSP multipliers.

### FLAGS

| Flag | Meaning |
|---|---|
| IF | INF as an argument. |
| NN | NaN as an argument. |
| NV | INF x 0. |
| OF | Result overflowed, INF returned. |
| UF | Result underflowed, zero returned. |

# floatdivide

Performs division (16 bit for float16, 32 bit for float32) of two floating-point numbers.




| Signal | Meaning |
|---|---|
| “start” | Start the division by holding to 1 for 1 clock cycle. |
| “a” | First floating-point operand. |
| “b” | Second floating-point operand. |
| “busy” | Set to 1 whilst the operation takes place. |
| “result” | Result of a / b. |

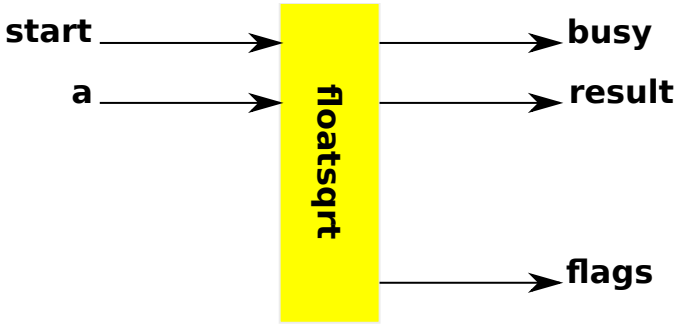
### FLAGS

| Flag | Meaning |
|---|---|
| IF | INF as an argument. |
| NN | NaN as an argument. |
| DZ | Divide by zero attempted. |
| OF | Result overflowed, INF returned. |
| UF | Result underflowed, zero returned. |

# floatsqrt

Adapted from https://projectf.io/posts/square-root-in-verilog/

Performs square root (16 bit for float16, 32 bit for float32) of a floating-point number.

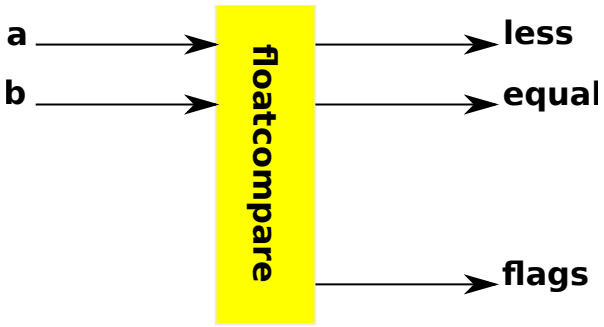



| Signal | Meaning |
|---|---|
| “start” | Start the square root by holding to 1 for 1 clock cycle. |
| “a” | Floating-point number to square root. |
| “busy” | Set to 1 whilst the operation takes place. |
| “result” | Result of √a. |

### FLAGS

| Flag | Meaning |
|---|---|
| IF | INF as an argument. |
| NN | NaN as an argument. |
| NV | Negative number. |
| OF | Result overflowed, INF returned. |
| UF | Result underflowed, zero returned. |

# Comparisons: floatcompare




Adpated from Berkeley SoftFloat https://github.com/ucb-bar/berkeley-softfloat-3

Performs comparisons (16 bit for float16, 32 bit for float32) of two floating-point numbers.

| Signal | Meaning |
|---|---|
| “a” | First floating-point operand. |
| “b” | Second floating-point operand. |
| “less” | Returns 1 if a < b. |
| “equal” | Returns 1 if a == b. |

### FLAGS

| Flag | Meaning |
|---|---|
| IF | INF as an argument. |
| NN | NaN as an argument. |
| NV | NaN given as an input. |

# Examples Of Working and Stages

ADDITION / SUBTRACTION ( 100 – 99 )

| Integer | float32 Hexadecimal | float32 Binary Format |
|---|---|---|
| 100 | 42c80000 | { 0 10000101 10010000000000000000000 } |
| 99 | 42c60000 | { 0 10000101 10001100000000000000000 } |

100.0 – 99.0 is done as 100.0 + (-99.0) by switching the sign of 99.

| | | |
|---|---|---|
| 100 | { 0 10000101 10010000000000000000000 } | + |
| -99 | { 1 10000101 10001100000000000000000 } | |

The exponents are expanded to 10 bit and have the bias (127) removed, and the mantissas expanded to 48 bits with the hidden initial 1 bit aligned at bit 47

If the exponents are not equal (they are in this example) then the mantissa of the smaller number is shifted right by the difference in the exponents.

The binary addition/subtraction is then performed.

| | | |
|---|---|---|
| 100 | { 0 0000000110 011001000000000000000000000000000000000000000000 } | + |
| -99 | { 1 0000000110 011000110000000000000000000000000000000000000000 } | |
| → | { 0 0000000110 000000010000000000000000000000000000000000000000 } | |

The result is then normalised, aligning the mantissa to the left, subtracting 1 from the exponent each time the mantissa is shifted to the left, until the leading bit is at 46, and then an extra shift is performed to fully normalise the mantissa.

| | |
|---|---|
| → | { 0 0000000000 100000000000000000000000000000000000000000000000 } |

This normalised result is then rounded, not required in this example, as the rounding bit highligthed in gold is 0, and packed, by adding the bias (127) back to the exponent and extracting the 8 lower bits, dropping the leading 1 from the mantissa and extracting the following 23 bits, giving a final result of 1.

| | | |
|---|---|---|
| → | { 0 01111111 00000000000000000000000 } | 1 |

# Examples Of Working and Stages

ADDITION / SUBTRACTION ( 50 + 1/3 )

NOTE: ( 1/3 cannot be accurately represented in float32 ).

| Integer | float32 Hexadecimal | float32 Binary Format |
|---|---|---|
| 50 | 42480000 | { 0 10000100 10010000000000000000000 } |
| 0.33 | 3eaaaaab | { 0 01111101 01010101010101010101011 } |

The exponents are expanded to 10 bit and have the bias (127) removed, and the mantissas expanded to 48 bits with the hidden initial 1 bit aligned at bit 46.

| | | |
|---|---|---|
| 50 | { 0 0000000101 011001000000000000000000000000000000000000000000 } | + |
| 0.33 | { 0 1111111110 010101010101010101010101100000000000000000000000 } | |

If the exponents are not equal then the mantissa of the smaller number is shifted right by the difference in the exponents.

| | | |
|---|---|---|
| 50 | { 0 0000000101 011001000000000000000000000000000000000000000000 } | + |
| 0.33 | { 0 0000000101 000000001010101010101010101010110000000000000000 } | |

The binary addition/subtraction is then performed.

| | | |
|---|---|---|
| → | { 0 0000000101 011001001010101010101010101010110000000000000000 } | |

The result is then normalised, aligning the mantissa to the left, subtracting 1 from the exponent each time the mantissa is shifted to the left, until the leading bit is at 46, and then an extra shift is performed to fully normalise the mantissa.

| | | |
|---|---|---|
| → | { 0 0000000101 110010010101010101010101010110000000000000000 } | |

This normalised result is then rounded, not required in this example, as the rounding bit highlighted in gold is 0, and packed, by adding the bias (127) back to the exponent and extracting the 8 lower bits, dropping the leading 1 from the mantissa and extracting the following 23 bits, giving a final result of 50.333332.

NOTE: ( 50 + 1/3 cannot be accurately represented in float32 ).

| | | |
|---|---|---|
| → | { 0 10000100 10010010101010101010101 } | 50.33 |