

Opensource Floating Point Adder

By Justin Schauer (jschauer@hmc.edu)

<http://www.hmc.edu/chips>

5/3/2002

This is an open source floating point adder designed by Justin Schauer as a summer research project at Harvey Mudd College for Dr. David Harris. It was designed to fully conform to the IEEE 754 standard. In its fully-featured format, this adder supports all rounding modes: round to nearest even, round to plus infinity, round to minus infinity, and round to zero; it supports denormalized numbers as both input and output; it supports all applicable exception flags: overflow, underflow, inexact, and invalid; it supports trapped overflow and underflow, where the result is returned with an additional bias applied to the exponent. All of these features are supported in hardware. This adder was designed with area minimization in mind, so it is single path, has a leading zero encoder rather than predictor, etc. The adder is also a single-cycle design.

How It Works

Inputs and Outputs

The adder takes three inputs: the two floating point operands and a control field (Fig. 1). The control field is 5 bits long. The two least significant bits (LSBs) define the rounding mode as shown in Table 1. Bit 2 is used to enable the hardware underflow trap handler. Bit 3 is used to enable the hardware underflow trap handler. Bit 4 is used to select the operation, as shown in Table 2.

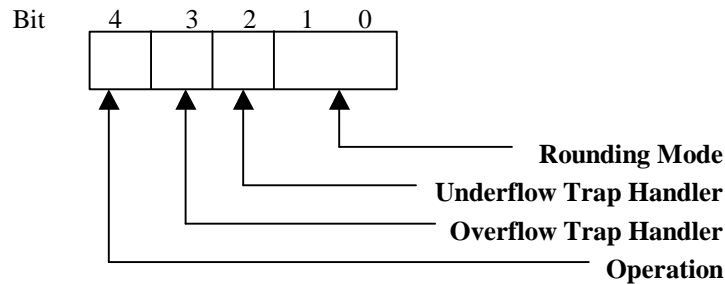


Fig. 1: The Control Field

Control Field Bit		Rounding Mode
1	0	
0	0	Round to nearest even
0	1	Round to zero
1	0	Round to plus infinity
1	1	Round to minus infinity

Table 1: Description of rounding mode bits

Bit 4	Operation
0	Addition
1	Subtraction

Table 2: Operation Selection Bit

To enable the trap handlers, simply set the appropriate bit high.

The adder has two outputs: the floating point result, and a flag field (Fig. 2). The flag field is also 5 bits long and has flags for the divide by zero (for compatibility with other floating point components), invalid, inexact, overflow, and underflow exception flags as shown in Fig. 2.

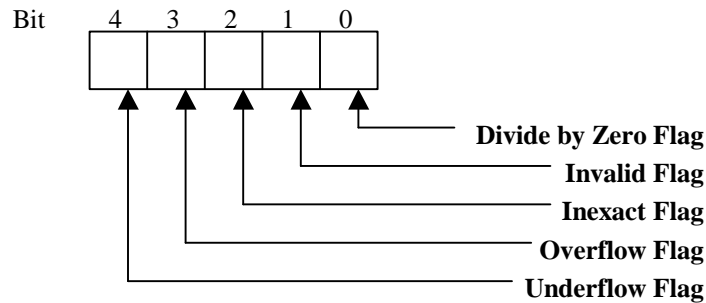


Fig. 2: The Flag Field

The Process

First, information about the inputs is gathered for later. The inputs are checked for being infinities, not a numbers (NaNs), signaling NaNs (SNaNs), and zero. Signaling NaNs are considered to be NaNs where the most significant bit (MSB) of the significand is 0. Whether an input is zero is determined by its exponent; if the exponent is all zeros, then the input is called zero. This means that denormalized inputs will result in a high zero input flag. So that denormalized numbers are handled correctly, all inputs that are determined to be zero have their exponent set to 1, since denormalized numbers have a base exponent of the minimum exponent which is 1 after the bias has been added. This ensures all exponent comparison issues will be handled correctly for denormalized inputs. The inputs are then separated into the larger and smaller through a comparison of all their exponent and significand bits.

The amount to shift the significands to align them properly is determined next. The smaller exponent is subtracted from the larger to get the shift amount. If this shift amount is more than the number of bits in the significand plus three, the significand that is being shifted can be just shifted by the number of bits in the significand plus three. This will result in correct addition or subtraction while still maintaining all sticky and guard bit information. After the shift amount has been determined, the significands have the assumed leading one prepended unless they were determined to be zero as described above (so neither zeros nor denorms have leading zeros prepended), and then the significand of the smaller input is right shifted by the calculated amount.

The shifted mantissas are then sent to the adder. First the effective operation is determined. It is the XOR of the signs of the inputs and the operation bit. If the effective operation is determined to be subtraction, the smaller mantissa is one's complemented. Only the top [# bits in significand + 3] bits of the mantissas are added, because this is the minimum

number needed to get an accurate result. There is also a single carry bit that is added to make up for the 1 bit loss of one's complement, but it is only added if the effective operation was subtraction and there were no guard or sticky bits. This is because if there is subtraction and there was a guard or sticky bit, the actual result would actually be one less but with both a round and sticky bit. This is also taken into account for the round and sticky bit calculations. This means many of the bits of the smaller mantissa are not involved in the addition (since it was right shifted), and the most significant of these bits is stored as the guard bit, and the OR of the rest is stored as the pre-sticky bit.

The [# bits in significand + 3] bit result of the addition or subtraction is normalized. First, the number of leading zeros is determined with some hardware similar to a priority encoder. The number of leading zeros is the amount to shift the mantissa for normalization. The exponent also has to be adjusted to make up for the shift, so the shift amount is also subtracted from the larger exponent. This subtraction serves a second purpose: if the result of this subtraction is negative, it indicates a tiny result and possible underflow or denormal result. Since the standard allows for tininess (for the purpose of determining underflow) to be detected before rounding, this is when it is done in this model. However, an underflow has not been encountered in our testing. This may be a lacking in the model, our tests, or underflow may not be possible with subtraction. For denormal results, instead of left shifting by the amount of leading zeros, the sum is left shifted by the value of the bigger exponent.

After all the shifting, the final significand is determined (pending special cases). If the effective operation was subtraction and the first bit of the larger mantissa cancelled, the guard bit becomes the LSB of the final significand and the round and sticky bits are calculated differently than if there was no cancellation. If there was cancellation, the sticky bit is simply the presticky bit, and the round bit is the XOR of the guard and sticky bits. If there is no cancellation, the sticky bit is the OR of the presticky bit and any other discarded bits. The round bit uses what was discussed in the determination of the carry bit above, as well as the bit after the LSB of the final significand (the MSB of the bits discarded after final significand determination) in its determination. Some useful flags are also determined at this point. If all the bits of the result of the effective operation on the mantissas are zero, the zero result flag is set. If the result of the subtraction of the shift amount from the larger exponent was negative and the final significand is not all zeros, the result denormal flag is set. If either the round or sticky bits are high, the inex flag is set. The inex flags indicates the result is inexact. Note that this is NOT the same as the inexact exception flag, as the inexact exception is set when the result is inexact or the result overflows and is not trapped.

The final sign is determined once it is known whether the result is going to be zero. This is because of the special case in round to minus infinity mode where $-x + x = -0$, whereas in all other rounding modes, the result is $+0$. Other than this factor, the final sign is calculated using the operation bit, the sign bits of the inputs, and which input was bigger in magnitude.

Now the final significand is rounded. The rounding modes are decoded as described in Table 1. The result of rounding is either the final significand incremented by one, or the final significand unchanged. Using the rounding mode, the sign of the final result, and the round and sticky bits, the choice whether or not to increment is made. If the final significand is incremented, it could overflow. The final exponent has to be adjusted anyway, because the leading zero encoder starts encoding at the MSB of the [# bits in significand + 3] bit of the sum, which is actually the overflow bit (this is done for a number of reasons, and it seems to be more area-efficient to put this fix later than to change the way the leading zero encoder works). Since

the final exponent has to be adjusted by one anyway, the case of overflow on rounding is accounted for by an incrementing of two rather than one. In binary, incrementing by 2 is almost the same as incrementing by one, except the incrementing starts at the second-to-least significant bit, rather than the LSB. At this point the final, correctly rounded, significand has been determined, as well as the final exponent (these are pending special case analysis, however).

Finally, the exception flags are determined and the final result is pieced together. Overflow is signaled when the exponent overflows or is all ones, the result wasn't tiny or zero, and the inputs weren't special cases (infinity and NaN arithmetic is exact). Underflow is signaled when the result is tiny and inexact or, if underflow is trapped, when the result is tiny. Invalid is signaled if an input was a signaling NaN or if magnitude subtraction of infinities occurred. Inexact is signaled if the result was inexact or if an untrapped overflow occurred, and the inputs weren't special cases.

The final significand and exponent are determined by looking at the inputs and exception flags and determining whether the exponent or significand need to be special, such as infinity, NaN, zero, largest representable number, smallest representable number, etc. One special case that requires slightly more hardware is trapped overflow or underflow handling. If the result under/overflows and the under/overflow trap is enabled, rather than returning infinity or something of that nature, the correct significand is returned with an exponent that has been biased again as described in the IEEE 754 standard. Since the logic to get the final result is so complicated due to all the possible special cases, the calculated significand and exponent don't enter the logic until as late as possible to reduce timing.

Verification

Method

This code has now been through a fairly extensive verification process. In order to test that the results are accurate for any combination of exponent and significand precisions, Dr. Harris wrote a floating point adder in software that also allows for varying precision in the exponent and significand length. We have tried to ensure against errors in having these two models tell each other they are correct when they are really both making the same mistakes in two ways. First of all, different people coded the hardware and software adders, using vastly different methods (since there was no concern for optimization in the software adder). Secondly, the software adder's results are checked against the Intel floating point unit's results whenever possible (when the precisions being tested correspond to either single or double precision). So the software adder generates the two input floating point numbers, the control field, and the expected output floating-point number and flag field.

Test Plan (by Dr. David Harris and Justin Schauer)

Test categories:

- Exhaustive testing of low-precision adders
- Directed tests
- Random tests

Exhaustive testing:

All possible input patterns for each input will be tried for adders of the following sizes:
1 sign, 3-5 exponent, 3-6 significand bits

Directed tests:

All permutations of the following inputs will be checked:

For 1 sign bit, e exponent bits, s significand bits

Exponent = 0, 1, 2, $2^{e-2}-1$, 2^{e-2} , $2^{e-1}-2$, $2^{e-1}-1$, 2^{e-1} , $3*2^{e-2}-1$, $3*2^{e-2}$, 2^e-3 , 2^e-2 , 2^e-1 , 42,
10 random numbers
Significands = 0, 1, 2, 3, 4, 5, 2^{s-3} , 2^{s-2} , 2^{s-1} , $3*2^{s-2}$, 2^s-3 , 2^s-2 , 2^s-1 , 42, 20 random
numbers

Random tests:

50 million tests for single and double precision with all options enabled
-provide different seed each time
5 million tests for all other scenarios

Modes to test:

Add/sub provide as 3rd input (not usually tested)
Rounding modes: RN, RP, RM, RZ (4 cases)
Special numbers: Infinities (2 cases)
No NaNs, regular NaNs, regular and signaling NaNs (3 cases)
Denorm (as inputs, as outputs) (4 cases)
Traps: overflow, underflow (4 cases)
Exception handling: inexact, overflow, underflow, invalid (16 cases)
Note: if trap is on, corresponding exception must be on
Default mode: add/subtract provided, all rounding modes, special numbers, denorms,
traps, and exceptions handled

Test Scenarios:

Note: all tests are on all directed and random cases unless exhaustive

- 1) Test 32-bit adder against C reference model (old test program) in all 4 rounding modes
- 2) Test 32-bit adder against new reference model in all permutations of modes
- 3) Test 64-bit adder against new reference model in all permutations of modes
- 4) Test low precision exhaustive cases against new reference model
- 5) Test other precisions against new reference model in:
 - a. Add vs. add/subtract mode, default all else
 - b. All four rounding modes, default all else
 - c. All special number options, default all else
 - d. All four trap options, default all else
 - e. All 16 exception combinations, no traps, default all else
 - f. 10 random combinations of options

For the cases of 1 sign bit,

$$e=4, s=7$$

$$e=5, s=10$$

$$e=6, s=9$$

$e=6, s=17$
 $e=7, s=16$
 $e=8, s=15$
 $e=9, s=38$
 $e=10, s=37$
 $e=15, s=64$
 $e=16, s=111$
 $e=32, s=223$

Progress

So far the final version of the code, both in regular form, and passed through the generator on the website (<http://www.hmc.edu/chips/generate.html>), has been tested against the C reference model (which checks the outputs of the hardware against the results as calculated by an Intel FPU). The hardware outputs have also been tested against the new reference model in single precision, double precision, exhaustively at $e=3, s=3$, and against 50 million random tests cases in both single and double precision. Essentially, we are now virtually certain this hardware adder will produce correct results at any chosen exponent and significand precisions, as long as all other options are left at the default values. An IMPORTANT NOTE, however, is that this adder does not support exponent or significand precisions lower than three bits, so the absolute smallest floating point adder that can be produced is a 6-bit one, with 1 sign bit, 3 exponent bits, and 3 significand bits.

Synthesis

Our adder was synthesized using the LSI G12p standard cell library. This library is based on a .18 micron process and has a fanout-of-4 inverter delay of 90 ps. This library is also a worst case library, which is why the FO4 delay is 90 ps instead of the usual 60 ps for a .18 micron process. Areas are reported in cell units where one cell unit is equivalent to 5.95 square microns. In general, the options that were used for compilation were high map_effort, high area_effort, and max_area set to zero for area optimization. For all options that were used, see our Synopsis Design Compiler scripts. In Fig. 3 are the area results for the fully featured adder operating in single precision mode at different timings.

Area vs Timing Plot for Area Optimized Floating Point Adder

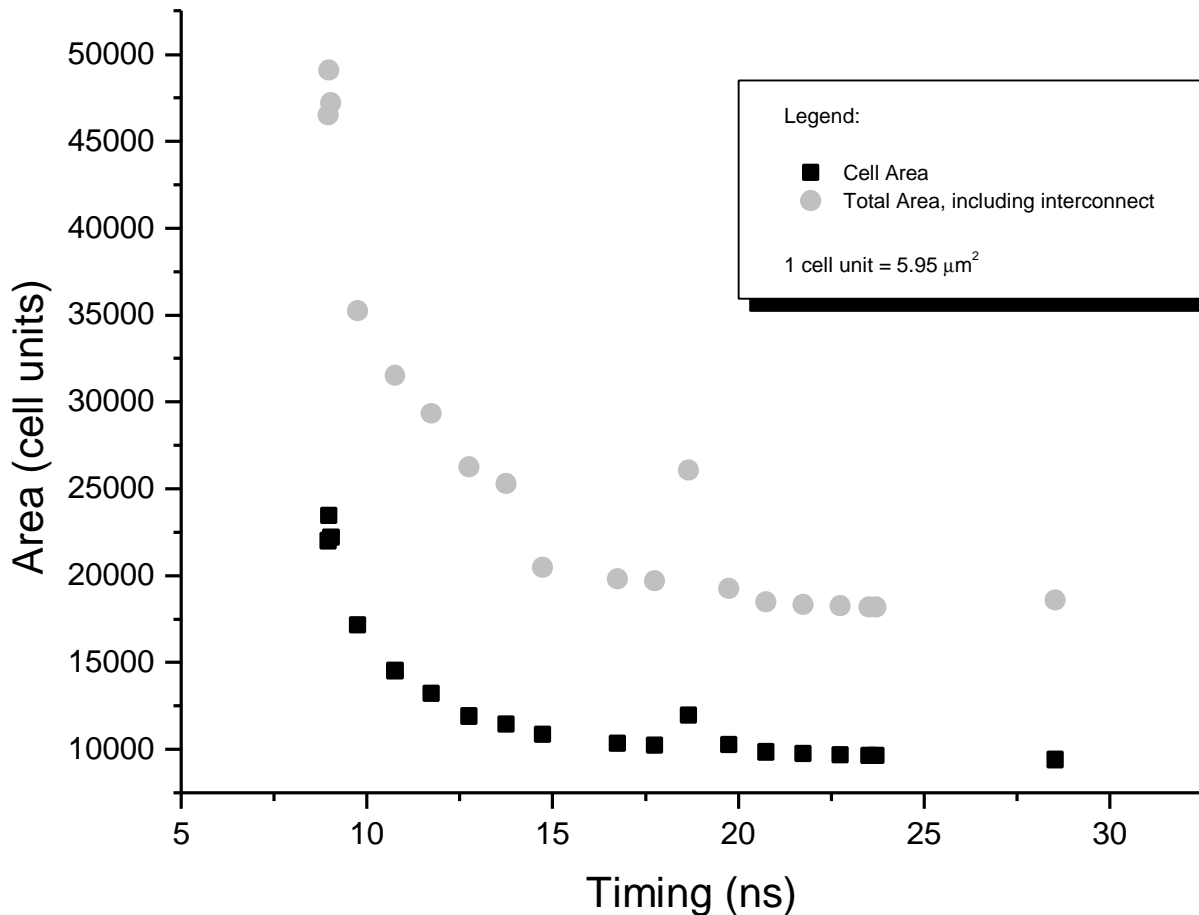


Fig. 3: Graph showing area and timing tradeoffs

Future Work

Additions and Improvements

The next steps involve better verification of the current design, and additional adder designs that focus on optimizing for time. Things I want to modify the adder with include two-path design, where one path has a large alignment shift and small normalization shift, and the other has a small alignment shift and a large normalization shift. Other ideas are using leading zero anticipation, rather than encoding, and in general code that is more optimized for speed than area. The first steps we will take in making better testing software is creating test software that is as modular as the Verilog code. This means a variable precision floating point adder that can have things such as denorm and rounding mode support deactivated or changed must be created in software, and linked to the scripts that parse the Verilog code. One idea could be to have the generate script on the website return both Verilog code and a batch file that will run the test case generation code with the desired options.