# **A Floating-Point Unit for**

# **Arithmetic Operations**

Jeff Walden – 6.111 13.12.06

In this paper we present a design for a floating point unit partially compliant with the IEEE 754 floating point standard. The unit fully implements comparisons and partially implements floating-point addition and multiplication. It is fully parametrized and may be used with floating point numbers whose composite fields have widths of any desired length.

# Tables

1. Floating Point Numbers and their corresponding values in UI

# Figures

1. The IEEE 754 floating point format.

### Overview

Integral arithmetic is common throughout computation. Integers govern loop behavior, determine array size, measure pixel coordinates on the screen, determine the exact colors displayed on a computer display, and perform many other tasks. However, integers cannot easily represent fractional amounts, and fractions are essential to many computations. Floating-point arithmetic lies at the heart of computer graphics cards, physics engines, simulations, and many models of the natural world.

A Turing-complete system can of course emulate floating-point logic if it desires. Unfortunately, the immense complexity of floating-point arithmetic makes this endeavor extremely expensive; furthermore, floating-point logic is substantially more complicated than one might expect. Unless the implementation conforms exactly to the IEEE 754 floating-point standard, deviations in behavior between the floating-point environment a client programmer expects and the actual environment can substantially complicate writing programs.

Implementing floating-point arithmetic in hardware can solve two separate problems. First, it greatly speeds up floating-point arithmetic and calculations. Implementing a floating-point instruction will require at a generous estimate at least twenty integer instructions, many of them conditional operations, and even if the instructions are executed on an architecture which goes to great lengths to speed up execution, this will be slow. In contrast, even the simplest implementation of basic floating-point arithmetic in hardware will require perhaps ten clock cycles per instruction, a small fraction of the time a software implementation would require. Second, implementing the logic once in hardware allows the considerable cost of implementation to be amortized across all users, including users which may not be able to use another software floating-point implementation (say, because the relevant functions are not publicly available in shared libraries). Getting it right is difficult, but once this happens once, it need not happen again for any user running on that hardware.

## A Brief Overview of Floating Point and IEEE 754

Floating-point arithmetic as differs in a number of ways from standard integral arithmetic. Floating-point arithmetic is almost always inexact. Only floating point numbers which are the sum of a limited sequence of powers of two may be exactly represented using the format specified by

IEEE 754. This contrasts with integer arithmetic, where (for example) the sum or product of two numbers always equals their exact value sum, excluding the rare case of overflow. For example, in IEEE arithmetic 0.1 +0.2 is not equal to 0.3, but rather is equal to 0.300000000000004. This has many subtle ramifications, with the most common being that equality comparisons should almost never be exact - they should instead be bounded by some epsilon. Another difference between floating-point and integer numbers is that floating-point numbers include the special values positive and negative infinity, positive and negative zero, and not-a-number (NaN). These values are produced in certain circumstances by calculations with particular arguments. For example, infinity is the result of dividing a large positive number by a small positive number. A third difference is that floating-point operations can sometimes be made "safe" against certain conditions which may be considered errors. In particular, floating-point exceptions provide a mechanism for detecting possibly-invalid operations such as the canonical division by zero, overflow to an infinity value, underflow to a number too small to be represented, or inexact when the result of a calculation isn't exactly equal to the mathematical result of the calculation. Other differences will be mentioned later as necessary.

IEEE 754 specifies a particular representation for a floating-point number, consisting of a sign, an exponent, and a significand.





A floating point number's value is usually (-1)^(sign) \* (1 . fbits) \* 2 ^ (ebits – BIAS), except in the special cases of NaN, infinities, and numbers of particularly small magnitude. fbits specifies the *binary* fractional portion of the number; in other words, if fbits is 1 followed by some number of zeros, the middle term is 1.1. Note also that this is a binary fraction: for example, 1.1 is the decimal 1.5, and 1.11 is the decimal 1.75. ebits is a positive number specifying the exponent in the value; the purpose of BIAS is to ensure the exponent value can be represented as a positive number. The special cases mentioned earlier are implemented by reserving numbers with an exponent field consisting of all zeros or all ones and giving them special meanings.

#### Implementation Overview

Initial plans for the floating-point unit envisioned were ambitious: full support for single-format IEEE 754 floating-point addition, subtraction, multiplication, and comparison, along with full support for exceptions. As has been mentioned, however, IEEE 754 is extremely complex, and as a result some goals were omitted. In particular exceptions were not implemented, and addition and multiplication have known bugs with respect to compliance with specified requirements; the test suite included in the appendix and the results from executing it detail the current known bugs.

Floating-point unit functionality can be loosely divided into the following areas: the Adder module, the Multiplier module, and the Compare module. Together these modules compose the internals of the FPU module, which encapsulates all behavior in one location and provides one central interface for calculation of floating point calculations.

### Using the Floating-Point Unit

At the hardware level, the FPU's inputs consists of two floating-point values and a selector to determine the calculation performed by the floating-point unit. Its outputs consist of the result of the specified calculation and the result of a comparison between the two specified values. The user is expected to do some interpretation of the returned values if it needs more specific information. (For example, the module provides no sign-detection facility, and it is expected the user should use an external bit comparison to determine sign; the user would also be required to synthesize support for greater-than-or-equal-to and less-than-or-equal-to operators if he needed them.)

At the human-operated user interface level, the interface consists of the labkit switches, buttons, and LED display. The demo module is preprogrammed with 16 separate floating-point values. The upper half of the switches select the value of the left operand by mapping directly onto the stored floating point values (0 to 15, big-endian bit order), and the lower half select the value of the right operand in the same manner.

Floating-point values in the interface							
0	+0.0	4	NaN	8	Smallest positive denormal	12	+25.0
1	-0.0	5	+Infinity	9	Largest non- infinite value	13	+0.1
2	+1.0	6	-Infinity	1 0	+5.0	14	+0.2
3	+7.0	7	+0.5	1 1	-5.0	15	~+0.3

Table 1: Floating Point Numbers and their corresponding values in UI

These values are also displayed in the UI using the hex LED display. Ideally we would show the left, right, and result values simultaneously, but with only 16 digits we can only show two at once. The result is the most important value to display, so it is always displayed in the right half of the LEDs. The left half alternates between displaying the left value and the right value every five seconds. The timer which controls this switching is resettable by pressing the Enter button, which simultaneously resets the timer to zero and changes the operation being performed to multiplication. The operation being performed may be switched by pressing button 0, alternating between multiplication and addition. The value of the comparison of the two operands is indicated on the single-dot LEDs 0 and 1 on the labkit. If neither LED is on, the left operand is greater than the right operand. If only LED 0 is on, the right operand is greater than the left operand. If only LED 1 is on, the left operand is equal to the right operand. (Note that since NaN is not equal to any value, this condition will not be matched if both operands are NaN.) If both LED 0 and LED 1 are on, then the left operand and right operand are incomparable. (This occurs if either operand is NaN.)

## **Description**

### Internal Overview

The floating point unit itself is basically a thin wrapper around the floating point adder, multiplier, and comparator. For simplicity and purposes of modularity each separate component is its own module; in a real implementation designed for efficiency the modules would most likely be inlined. The original intention was to include specific prediction logic to anticipate situations where the output could be determined without a full analysis of the arguments (e.g., a NaN in any operation automatically means the result is NaN), but implementation was too complicated to be worthwhile. Also, it was unclear whether this could actually increase efficiency. Furthermore, while efficiency was originally a goal, the substantially increased difficulty of implementation made simply getting something working of more importance than getting something working efficiently. The various papers [1] [2] consulted when considering efficiency issues made it clear that most of the efficiency gains would be the result of obscure hacks which would make the code that much harder to write and manipulate, so efficiency was discarded fairly early on as a concern in implementation.

#### Components

For further details on any component, see the appropriate section in the appendix. Each component's implementation is prefaced with an extensive description of its interface, and this description is intended to provide sufficient information to use the module within another circuit.

Of particular note regarding the components of the FPU taken as a whole is that, aside from the code used directly in the labkit (and, of course, the test code which must choose parameter values for testing), none of the modules which perform calculations ever rely on floating point number field widths, except through parameters. In fact, these parameters are exposed in documentation for external manipulation, for programmers who wish to modify them. This means that if one does not require the full precision of IEEE 754 single format floating point (which has an exponent width of 8, a fractional width of 23, and a sign bit), the floating point format widths may be tweaked by setting parameters appropriately to use a narrower floating point format. Alternately, users who desire double-width floating point can use it by redefining the parameters. This is arguably the most interesting aspect of the floating-point modules, because this means they can be used in any number of situations outside the one for which they were designed. (No performance claims can be made with this strategy except with respect to particular parameter values, but this seems unlikely to be a huge problem; if it is, the user is no worse off than he would have been had the FPU not been correctly parametrized, and since the dependencies are clear it may even make his optimization attempts easier.)

Adder

The adder provides a partial implementation of IEEE 754 addition. As is demonstrated in the included test script output, this module is far from

complete. The algorithm for floating point addition is actually particularly heinous (and would have been worse had the original plan to implement a 1995-state-of-the-art adder been accomplished) to implement, involving several carefully-made bit selects, several shifts, and in its full form normalization and rounding using guard digits to determine how the final significand must be rounded, if necessary. The full algorithm for addition in its simplest form is as follows[3]:

- 1. Exponent subtraction: Perform subtraction of the exponents to form the absolute difference  $|E_a E_b| = d$ .
- 2. Alignment: Right shift the significand of the smaller operand by d bits. The larger exponent is denoted E\_f.
- 3. Significand addition: Perform addition or subtraction according to the effective operation, which is a function of the opcode and the signs of the operands.
- 4. Conversion: Convert the significand result, when negative, to a signmagnitude representation. The conversion requires a two's complement operation, including an addition step.
- 5. Leading-one detection: Determine the amount of left shift needed in the case of subtraction yielding cancellation. Priority encode (PENC) the result to drive the normalizing shifter.
- 6. Normalization: Normalize the significand and update E\_f appropriately.
- 7. Rounding: Round the final result by conditionally adding 1 unit in the last place (ulp), as required by IEEE 754. If rounding causes an overflow, perform a 1-bit right shift and increment E\_f.

The given implementation follows this algorithm, even though it is not as efficient as some. Its chief advantage is that it is theoretically easier to reason about during implementation, enabling easier implementation. Even with its relative straightforwardness, however, the algorithm refused to fully yield to attempts to implement it. Some aspects of the algorithm are incorrectly not implemented, such as the final rounding if necessary; given more time these steps would be refined to implement correct behavior.

The internal implementation is composed of a series of always blocks, each of which corresponds to a stage in the algorithm. All variables assigned in each block are accessed using continuous assignment, with only a few exceptions. The exceptions occur during calculation of the number of leading zeros, where two blocking-assignment variables are used to implement a for-loop to count the number. (One of the variables is the loop variable, which clearly must be assigned in a blocking manner; the other is a flag used to determine when the first non-zero bit is touched to determine when the number of leading zeros should no longer be increased. It doesn't seem possible to convert either set of assignments to non-blocking while preserving width-agnostic syntax.) One key convention used in the Verilog is to post-fix each variable name with an underscore and the number of its stage within the internal pipeline, e.g. "leadingZeros\_5". Under this convention, every variable being assigned in an always block must have the same suffix (because each is part of the same stage). Additionally, and perhaps more importantly, every value on the right-hand side of an assignment must either be a literal, a wire whose value is determined using only variables assigned in the previous stage, or a variable assigned in the previous stage. So long as numbering is correctly maintained, this allows easy verification that no inter-stage dependencies (other than of one stage on its predecesssor) exist. Combined with a further convention of declaring stage variables immediately prior to each always block, correctly using per-stage registers was simple.

The Adder's implementation is given in the file Adder.v.

#### Multiplier

Floating point multiplication is significantly easier than floating point addition. Floating point addition requires at least two shifts, several delays for calculating shift arguments, and several additions and possibly an inversion. Multiplication, on the other hand, requires only multiplication of significands, addition of exponents, and normalization and rounding. The implementation presented here is not completely compliant with the subset of IEEE 754 it attempts to support (in particular it does not quite handle multiplications resulting in infinities or situations which involve rounding, particularly when round-to-nearest applies). However, it performs reasonably well on tests, passing more than 80% of the tests given to it.

The multiplier internally consists of a series of stages which implement the multiplication algorithm described above. One interesting aspect of it is that it special-cases multiplications which involve NaN, infinities, and zero from the very start of the algorithm. Searching through research papers yielded no novel solutions to the problem, so for those values the algorithm is the simplest that works – check for each specific combination of inputs that has a special output and override any calculated output with that specific output. One would hope generalized algorithms exist for this, but unfortunately it seems none do.

The multiplier's implementation is given in the file Multiplier.v.

#### Compare

Comparison of floating-point numbers isn't quite as trivial as it might seem; IEEE 754 specifies many edge cases in comparison which must be respected. For example, NaN is equal to no other value, including itself, and -0 is equal to +0. However, comparison requires no calculations, and since IEEE 754's format incorporates lexicographical ordering of values, comparison was actually fairly trivial; a person who understands the rules could implement the Compare module in perhaps half an hour at most (as was expected when this project was being considered). The Compare module is implemented as a simple tree of comparisons; edge cases such as NaN, zero, and infinities are handled first, followed by the general cases of finite positive and negative numbers. The module also has a good set of tests in TestCompare.v which should demonstrate its correctness.

The Compare module's implementation is given in the file Compare.v.

#### FPU

The FPU module merely serves as a container for the floating point adder, multiplier, and compare modules; its implementation is fairly straightforward.

The FPU module's implementation is given in the file FPU.v.

## Testing and Debugging

Testing of the FPU was conducted by writing Verilog test modules which exercised the adder, multiplier, and compare modules. Each test module consisted of a series of operand numbers and an expected result, run using a testing task. The module simply cycled through the tests, executing each in order, reporting errors and successes using the \$display function. This strategy actually worked extremely well for testing; the code was stored in a Subversion repository, and the cycle after making a change consisted of running the tests, checking the output for failed tests, and committing once errors had been removed. In particular the results of the Compare and Multiplier module tests verify that this works well.

The strategy had less success on the Adder module. This was partly due to the increased complexity of addition over multiplication; more stages and more interactions between variables made it harder to follow the logic. The logic itself was probably written four or five times, and no rewrite substantially improved the adder's correctness. Given more time my solution to fixing it would be as follows: take a set of floating-point numbers, walk through the logic for each by hand in full, and write logic based on the values generated. The primary problem with the adder tests was that they weren't granular enough – in retrospect the absence of inter-stage value tests substantially hurt progress at fixing issues (and keeping them fixed); determining exactly where an addition went astray was difficult.

I found two tools of particularly good use when debugging the modules: first and foremost, the free Icarus Verilog compiler for enabling working on the FPU while not at the lab, and second, the GTKWave program for viewing wavefile dumps generated by the Verilog compiler. Icarus Verilog's support for testing was invaluable, and for the most part it implemented the feature the compiler in the lab implemented. The only feature I consistently found myself missing was support for variable part-selects (e.g., "foo[loc +: WIDTH]"). There existed a simple (if verbose) workaround using a for-loop, so this issue was of only minor annoyance. GTKWave as a wavefile viewer was adequate, but ModelSim was thoroughly better except for not being accessible outside of the lab.

The other big issue encountered was inconsistent support for Verilog semantics between the Xilinx compiler and Icarus Verilog. This issue occurred fairly often with comparisons, and in general I found support for signed values to be inconsistent in both compilers. Verilog's highly unintuitive treatment of signed values also probably played a role in making signed arithmetic particularly buggy. This issue was sometimes avoidable, but in certain cases no workaround could be found.

The final issue encountered with respect to cross-compiler consistency was that Xilinx would simulate the FPU correctly, but when asked to implement it on the labkit it actually failed to succeed at doing so, failing with an Out of Memory error (at over two gigabytes of memory); this completely prevented actually running and executing the demonstration code, which as surprises go was extremely unwelcome. It is unclear what in the code caused this to happen; all for loops were bounded correctly, and the only other possibility that comes to mind is excessive parametrization such that the compiler can't handle it. Neither issue appeared solvable without compromising the features supported by the code, so unfortunately the code as turned in cannot be compiled using the Xilinx toolchain.

### Conclusions

Overall, this project provided an excellent base for implementing the IEEE 754 floating point format; the details and idiosyncrasies of the format required a considerable amount of care and ingenuity with Verilog to implement correctly. Given more time, fully implementing the parts of IEEE 754 that were not implemented would be an extremely interesting challenge. Unfortunately, part of what would likely make it a challenge is buggy support for comparisons and signed arithmetic in Verilog compilers; this problem seems unlikely to go away soon, and it was a substantial issue in completing the implementation. Nonetheless, completing the implementation and optimizing it to reduce cycle count and reduce delays through concurrent calculation would be a substantial and interesting task.

### References

- 1. Bruguera, J. D. and Lang, T. 1999. *Leading-One Prediction with Concurrent Position Correction*. IEEE Trans. Comput. 48, 10 (Oct. 1999), 1083-1097. DOI= <u>http://dx.doi.org/10.1109/12.805157</u>
- Suzuki H et al. Leading-zero anticipatory logic for high-speed floating point addition. IEEE Journal of Solid State Circuits, 1996, 31(8): 1157-1164.
- 3. Stuart F. Oberman and Michael J. Flynn. 1996 *A Variable Latency Pipelined Floating-Point Adder*. Technical Report. UMI Order Number: CSL-TR-96-689., Stanford University.