Area Efficient Floating-Point Adder and Multiplier with IEEE-754 Compatible Semantics

Andreas Ehliar
Dept. of Electrical Engineering
Linköping University
Linköping, Sweden
Email: andreas.ehliar@liu.se

***Abstract*—In this paper we describe an open source floating-point adder and multiplier implemented using a 36-bit custom number format based on radix-16 and optimized for the 7-series FPGAs from Xilinx. Although this number format is not identical to the single-precision IEEE-754 format, the floating-point operators are designed in such a way that the numerical results for a given operation will be identical to the result from an IEEE-754 compliant operator with support for round-to-nearest even, *NaN*s and *Inf*s, and subnormal numbers. The drawback of this number format is that the rounding step is more involved than in a regular, radix-2 based operator. On the other hand, the use of a high radix means that the area cost associated with normalization and denormalization can be reduced, leading to a net area advantage for the custom number format, under the assumption that support for subnormal numbers is required.**

The area of the floating-point adder in a Kintex-7 FPGA is 261 slice LUTs and the area of the floating-point multiplier is 235 slice LUTs and 2 DSP48E blocks. The adder can operate at 319 MHz and the multiplier can operate at a frequency of 305 MHz.

I. Introduction

When porting floating-point heavy reference code to an FPGA there are several different implementation strategies. The first is to convert all (or most) of the reference code into fixed-point. This option will most likely lead to the most efficient implementation, as most FPGAs are optimized for fixed-point arithmetic. However, the development cost for this implementation strategy can be prohibitive, depending on whether for example techniques such as block floating-point can be used. In many cases it is also not possible to get the exact same results as in the reference code.

Another strategy is to simply implement the floating-point based algorithms directly using floating-point operators in the FPGA. This probably means that IEEE-754 compatible floating-point operators will be required, as the reference code is probably using either single or double-precision IEEE-754 compatible floating-point arithmetic. The main advantage of this strategy is that it is possible to create an application which behaves in exactly the same way as the reference code. The drawback is that the area overhead associated with fully compliant IEEE-754 operators can be rather large. (Although it should also be noted that porting reference code from a language such as C and C++, where the compiler has a relatively free reign in choosing whether single, double, or extended precision is used, is not trivial if bit-exact results are required, see [1] for more information.)

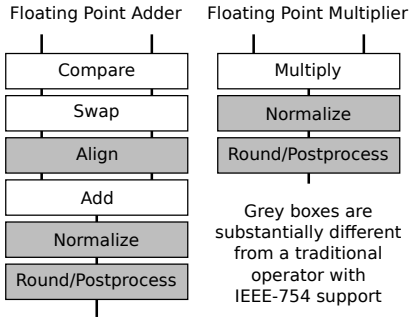

Fig. 1. Overview of the high radix floating point adder/multiplier

Yet another strategy that can be used is to analyze the reference code to determine if a custom floating-point format can be used in order to reduce the area cost required. This strategy exploits the fine-grained reconfigurability of FPGAs by not using more hardware than strictly required. For example, if the application doesn't require the precision offered by the single-precision IEEE-754 format, the size of the mantissa can be reduced in order to reduce the area cost, which is especially useful if the reduced mantissa size corresponds to a reduce in the number of required DSP blocks. The main drawback of this approach is that it is more difficult to compare the results from the FPGA implementation with the reference code, as it is no longer possible to get bit-exact compliance with the reference code.

A. Custom Floating-Point Format With IEEE-754 Semantics

The main subject presented in this paper is a new floating-point format which is designed to cater to the strength of FPGAs while retaining IEEE-754 semantics. This format is based on three main ideas:

- The area cost associated with normalization and denormalization can be reduced if a radix higher than two is used for the floating point number.
- Subnormal numbers are costly to implement. However, the extra dynamic range offered by subnormal numbers can also be had by extending the exponent by one bit.
- It is possible to create a floating-point format based on a high radix and an extended exponent range which can represent all possible values in a single precision

IEEE-754 number. A special post-processing stage can be used to ensure that the result is identical to the result from an IEEE-754 compatible single precision operator.

Under the assumption that the cost of converting between IEEE-754 and the custom format can be amortized over a few operators, this will lead to a net reduction in area while retaining the ability to give the exact same values as an IEEE-754 compatible implementation would give. The implementation details of such an operator will be similar to a traditional IEEE-754 compatible operator, although some of the pipeline stages will be substantially different as shown in Fig. 1.

II. High Radix Floating-Point Formats

Normally, a floating-point number is assumed to have radix 2. That is, the value of a floating-point number can be written as follows:

$$(-1)^{\text{sign}} \cdot \text{mantissa} \cdot 2^{\text{exponent}} \tag{1}$$

The most significant bit of the mantissa is often assumed to have the value 1, although this is not a strict requirement.

In contrast, if a high radix is used, the following definition is used instead, where base > 2:

$$(-1)^{\text{sign}} \cdot \text{mantissa} \cdot \text{base}^{\text{exponent}} \tag{2}$$

In order to make it easy to convert the exponent in a floating-point number with radix-2 to a high-radix representation it is also an advantage if the following holds as well:

$$\text{base} = 2^{2^k} \tag{3}$$

If the expression in Equation 3 holds, the exponent can be converted from radix-2 to the high-radix merely by removing the k LSB-bits from the radix under the assumption that the bias for the exponent is chosen appropriately.

The advantage of using a higher radix is that the floating-point adder can be simplified. In radix-2, shifters in the floating-point adder needs to shift the mantissa in steps of 1. If for example radix 16 is used these shifters only needs to shift in steps of 4. Such number formats have been shown to have a clear advantage over radix-2 based floating point numbers in FPGAs [2].

A. The HRFP$_{16}$ Format

The high radix floating-point format used in this paper (hereafter referred to as the HRFP$_{16}$ format to signify that the radix 16 is used) is based on radix-16 and has 7 exponent bits and 27 mantissa bits as shown in table I, for a total of 36 bits. This means that the mantissa is shifted in steps of four bits during normalization and denormalization rather than one bit as is the case in IEEE-754 (and most other floating-point formats). It should also be noted that since the format is 36 bits wide, this means that a value will can be conveniently stored in a 36-bit wide BlockRAM as well.

If bit 34 is set it signifies that the value is either a *NaN* or *Inf*, depending on the value in bit 33. (From a developers point of view, it is sometimes convenient if bit 34 is considered a part of the exponent, as this will automatically cause most overflows to be flushed to *Inf*.) Another special case is the value zero. This is stored as all zeroes in the exponent and mantissa field (and an appropriate sign in the sign field).

TABLE I. The Floating-Point Format Used in this Paper Compared to Single Precision IEEE-754

Format type	Bit number	Field name
IEEE-754 (single precision)	31	Sign bit
	30-23	Exponent
	22-0	Mantissa (with implicit one)
HRFP$_{16}$ (this paper)	35	Sign bit
	34-33	If 10: *Inf*. If 11: *NaN*. If `00` or `01`: Normal value
	33-27	Exponent
	26-0	Mantissa (with explicit one)
	Restriction: Only values that can be represented in IEEE-754 are allowed.	

TABLE II. HRFP$_{16}$ example values

Decimal	HRFP$_{16}$ format Sign	Exponent	Mantissa	Equivalent IEEE-754 encoding
0.0	0	0x00	0x0000000	0x00000000
1.0	0	0x60	0x0800000	0x3f800000
2.0	0	0x60	0x1000000	0x40000000
4.0	0	0x60	0x2000000	0x40800000
8.0	0	0x60	0x4000000	0x41000000
16.0	0	0x61	0x0800000	0x41800000
$1.401 \cdot 10^{-45}$	0	0x3a	0x4000000	0x00000001
$3.403 \cdot 10^{38}$	0	0x7f	0x7fffff8	0x7f7fffff

The following equation is used to determine the value of a given number in the HRFP$_{16}$ format

$$(-1)^{\text{sign}} \cdot \text{mantissa} \cdot 2^{4 \cdot \text{exponent} - 407} \tag{4}$$

where sign, mantissa, and exponent should be interpreted as integers. The encoding of a few different values in the HRFP$_{16}$ format can be seen in Table II.

B. On IEEE-754 Compatibility

The IEEE-754 standard only considers floating-point formats with a radix of 2 or 10 (although the decimal case will not be further discussed in this paper). This means that a straightforward implementation of a floating-point number format with another radix is not compatible with IEEE-754. However, the format described in Table I is capable of accurately representing all values possible in a single precision IEEE-754 number. This means that it is possible to create floating point operators based on the HRFP$_{16}$ format that will produce the exact same value as specified by IEEE-754, as long as a post-processing step is employed that guarantees that the result is a value that can be represented in single precision IEEE-754.

This was previously explored for floating-point adders in [3] where it was shown how rounding could be implemented in a high radix floating-point adder in such a way that the numerical result is the same as that produced by a IEEE-754 compatible adder. In the HRFP$_{16}$ format, this means that the guard, round and sticky bits are located at different bit positions, depending on where the explicit one is located as shown in Table III.

TABLE III. GUARD, ROUND, AND STICKY BIT HANDLING IN THE ADDER

Mantissa before post-processing and rounding
1xxx xxxx xxxx xxxx xxxx xxxx GRS. ..
01xx xxxx xxxx xxxx xxxx xxxx xGRS ..
001x xxxx xxxx xxxx xxxx xxxx xxGR S.
0001 xxxx xxxx xxxx xxxx xxxx xxxG RS

Key: x: These bits are allowed to be either zero or one in the final rounded mantissa
G: Guard bit, R: Round bit, S: Sticky bit, .: Bits that contribute to the sticky bit.
G, R, S, and . are set to 0 in the final mantissa.

C. Subnormal Handling

Subnormal numbers are typically not considered in FPGA implementations of floating-point numbers due to the additional cost of such support. Instead, subnormal numbers are treated as zeroes in for example the floating-point operators supplied by Xilinx [4]. The recommendation is instead to take advantage of the malleability of FPGAs and use a custom floating-point format with one additional exponent bit in such cases, as this will cover a much larger dynamic range than the small extra range afforded by subnormal numbers.

While this is a good solution for most situations, it is not sufficient in those situation where strict adherence to IEEE-754 is required, for example when an accelerator has to return a bit-exact answer as compared to a CPU based reference implementation. Rather than implement support for subnormal numbers in the HRFP_16 adders, IEEE-754 style subnormal values are instead handled by using an extra bit in the exponent, while at the same time implementing the rounding/post-processing step in such a way that the appropriate LSB bits are forced to zero while taking care to round at the correct bit position.

III. RADIX-16 FLOATING-POINT ADDER

This section contains the most interesting implementation details from the adder, readers who want even more implementation details are encouraged to download the source code. The adder is based on a typical pipeline of a floating-point adder with five pipeline registers as shown in Fig. III. The first stage compares the operands and the second operand swaps them if necessary, so that the smallest operand is right-shifted before the mantissas are added using an integer adder. After the addition the result is normalized and the exponent is adjusted. Finally, the result is post-processed and rounded to ensure that the output is a value which can be represented using a single precision IEEE-754 floating-point number.

A. Comparison Stage

The comparison stage is fairly straight-forward. It is possible to determine the operand with the largest magnitude merely by doing an integer comparison of the exponent and mantissa part of the number. By using the `LUT6_2` and `CARRY4` primitives, two bits can be compared at the same time. Normally XST will be able to infer such a comparator automatically. However, in order to handle a corner case involving +0 and -0 correctly it is convenient if this comparison could be configured as either "less than" or "less than or equal", depending on whether the sign bit of one of the operands is set or not. The author did not manage to write HDL in such a way that the synthesis tool would do this, hence this comparison contains hand instantiated `LUT6_2` and `CARRY4` components.

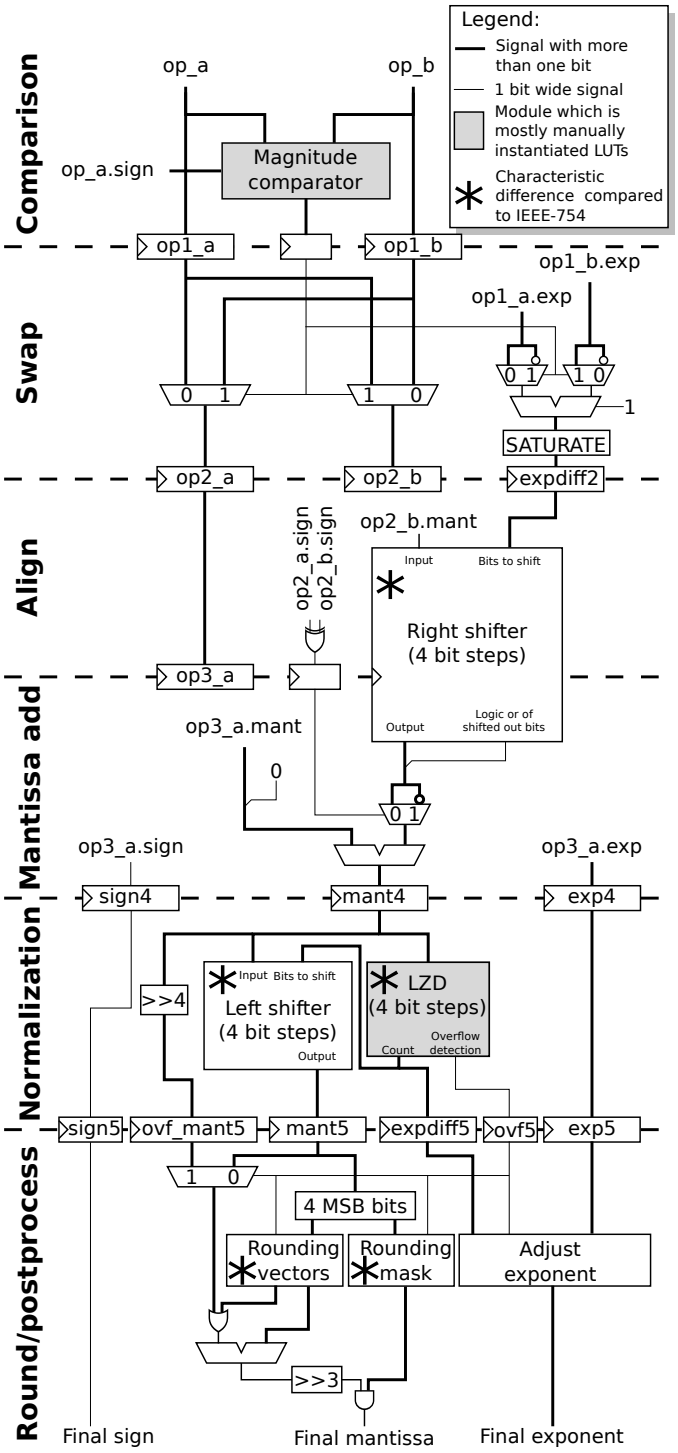

Fig. 2. High Level Schematic of the Adder

B. Swap Stage

The swap stage is responsible for swapping the operands to make sure that the smallest operand is sent to the shifter in the next stage. The multiplexers that swap the operands basically consists of inferred `LUT6_2` primitives where the `O5` and `O6` are used to either output the floating-point values on the input unchanged or swapped.

Besides swapping operands if necessary, this stage also calculates the difference between the exponents, so that the alignment stage can know how far to shift the right operand:

$$\text{expdiff} = \text{MIN}(7, |\text{exponent}_A - \text{exponent}_B|) \quad (5)$$

This calculation requires only one adder, as the previous stage has already performed a comparison between the operands. (However, some extra logic is still needed to determine whether to saturate the result to 7 or not.) Finally, this stage also sets the *NaN* or *Inf* flag (not shown in Fig. III).

C. Alignment Stages

To align the mantissa of the smaller operand a right shifter is required. Unlike a traditional IEEE-754 based adder, the shift distance is always a multiple four bits which will reduce the area and latency of this stage.

In the RTL code, the shifter is implemented as an eight to one multiplexer which has been divided into two four to one multiplexers. This division makes it possible to merge the final two to one multiplexer into the integer adder in the next stage. Besides the multiplexers, this stage also contains the or-gates required to calculate the sticky bit, which is necessary for correct implementation of round-to-nearest-even.

D. Mantissa Addition Stage

This stage contains the final step of the alignment operation as described above, as well as the integer adder/subtracter. The smaller operand is either added or subtracted from the larger operand depending on whether the sign of the operands are equal or not.

E. Normalization

It is well known that the normalization is one of the hardest, if not the hardest part of floating-point addition and this is still the case for the HRFP_{16} format, even if the complexity is reduced due to the fact that only shifts in multiples of four bits are required. Note that unlike floating-point formats based on radix 2 it is not feasible to use an implicit one in radix 16. Instead, a normalized number in radix 16 has at least one of the four most significant bits set to one.

The normalization stage consists of two parts. The first part calculates how many steps to shift by using a leading zero detector (LZD). Unlike a traditional IEEE-754 operator which operators on one bit at a time, the LZD in the HRFP adder works on four bits at the same time. This is implemented by manually instantiating `LUT6_2` instances configured as four input *AND* gates. As an additional micro-optimization, some of these LUTs are reused to detect groups of ones in the mantissa that would lead to an overflow when rounding is performed in the next stage. This is done by configuring the other output of the `LUT6_2` instances as four input *NOR* gates. Finally, as these modules are very latency critical they are floorplanned using `RLOC` primitives. The manual instantiation of these primitives was mostly done in order to floorplan these very latency critical part using `RLOC` attributes.

The second part consists of a left shifter which is implemented in the RTL code as an eight to one multiplexer configured to shift the mantissa in steps of 4 bits. Finally, this stage also contains some logic to set the result to *Inf* if necessary.

TABLE IV. AREAS FOR DIFFERENT PARTS OF THE ADDER IN A KINTEX-7 (XC7K70T-1-FBG484)

	Slice LUTs (LUT6)	Slice Registers
Compare	18	73
Swap	48	68
Align	35	70
Mantissa adder	32	40
Normalization	78	73
Round	50	0
Total	261	324

F. Rounding and Post Processing

This part is responsible for post-processing the result in such a way that it is numerically identical to what a IEEE-754 compliant unit would produce in round to nearest-even mode. This is also the part that differs the most when compared to the high radix floating-point adder proposed in [2].

Unlike the rounding module of a normal IEEE-754 based adder, rounding can only be done correctly if the position of the guard, round, and sticky bits are dynamic, depends on the content of the four most significant bits as shown in Table III. In addition, a post-processing stage is also required to ensure that an appropriate number of of the least significant bits are set to zero depending on where the most significant bit is set, according to table III. There are thus 24 consecutive bits that can be either one or zero for a given configuration of the MSB bits, regardless of where the most significant one is located, which corresponds to the number of mantissa bits present in a single-precision IEEE-754 formatted value. The remaining bits are set to zero.

Since the floating point adder has an extra exponent bit, all subnormal numbers that are possible in IEEE-754 can be represented as normalized numbers in the HRFP format. Note that it is not possible to add or subtract two IEEE-754 style values and get a result with a non-zero magnitude which is smaller than the smallest subnormal number. Thus, as long as only values that would be legal in IEEE-754 are sent into the adder, no special hardware is needed to handle the values that in IEEE-754 would be represented by using subnormal numbers. (This also extends to rounding/post-processing, since the guard, round, and sticky bit are guaranteed to be zero if a result corresponds to an IEEE-754 subnormal number.)

G. Area and Frequency

The total area of the adder when synthesized to a Kintex-7 FPGA is 261 slice LUTs, of which 109 are using both the `O5` and `O6` outputs. The areas for the different parts of the adder as reported in the map report file are shown in table IV. No route-thrus were reported for this design. The adder can operate at a frequency of 319 MHz in the slowest Kintex-7.

IV. RADIX-16 FLOATING-POINT MULTIPLIER

The multiplier contains six pipeline registers and is divided into three different parts. The first part uses an integer multiplier to multiply the two mantissas. The second performs

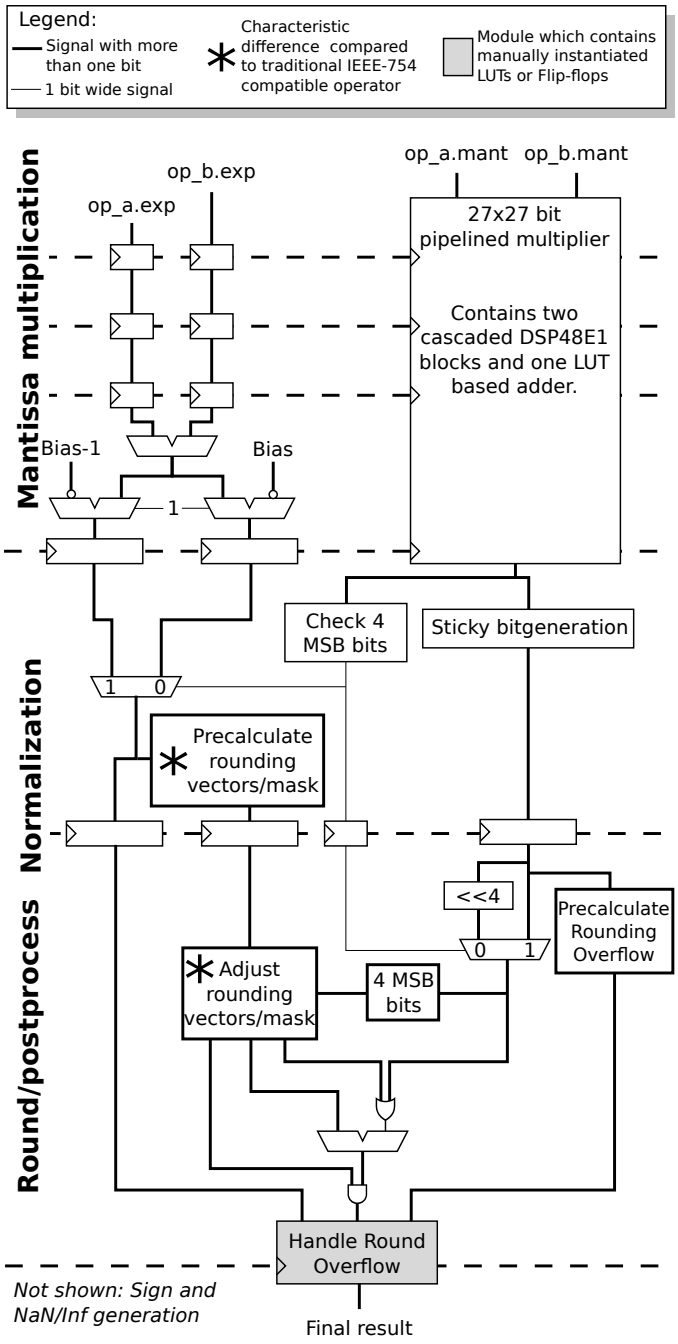

Fig. 3. High Level Schematic of the Multiplier

normalization and the third part post-processes the number and rounds it to ensure that the result is a value which can be represented using the single-precision IEEE-754 format.

A. Mantissa Multiplier Stage

The HRFP_{16} format has a 27 bit mantissa. This means that a 27×27 bit multiplier is required for the multiplier stage. In older Xilinx FPGAs where the DSP blocks were based on 18×18 bit multipliers this was not a big issue as four DSP blocks would be required for a 24×24 bit multiplier as a 27×27 bit multiplier. Unfortunately, the DSP blocks in the 7-series FPGAs are optimized for single-precision IEEE-754 multiplication, as two `DSP48E1` blocks can be combined to form a 24×24 bit unsigned multiplier but not a 27×27 bit multiplier.

TABLE V. THE IMPLEMENTATION OF THE 27×27 BIT MULTIPLIER

$A[2:0]$	Operation in cascaded `DSP48E1` blocks			Operation in slices		
000	$A \times B =$	$8 \times A[26:3] \times B$	$+$	$(0$	$+$	$0)$
001	$A \times B =$	$8 \times A[26:3] \times B$	$+$	$(B$	$+$	$0)$
010	$A \times B =$	$8 \times A[26:3] \times B$	$+$	$(0$	$+$	$2B)$
011	$A \times B =$	$8 \times A[26:3] \times B$	$+$	$(B$	$+$	$2B)$
100	$A \times B =$	$8 \times (A[26:3]+1) \times B$	$-$	$(2B$	$+$	$2B)$
101	$A \times B =$	$8 \times (A[26:3]+1) \times B$	$-$	$(B$	$+$	$2B)$
110	$A \times B =$	$8 \times (A[26:3]+1) \times B$	$-$	$(0$	$+$	$2B)$
111	$A \times B =$	$8 \times (A[26:3]+1) \times B$	$-$	$(B$	$+$	$0)$

While further DSP blocks could be added to increase the width of the multiplication to 27, this seems unnecessary for a bit width increase of only 3 bits. Fortunately it is possible to create a *slightly restricted* 27×27 bit multiplier suitable for use in HRFP_{16} by using two `DSP48E1` blocks and only 31 `LUT6` components.

This can be implemented by first cascading two `DSP48E1` components in order to provide a 27×24 bit unsigned multiplier. The remaining 27×3 bit multiplier is implemented using a combination of slice based logic and the pre-adder in the `DSP48E1` blocks as shown in the pseudo code in table V, where the key insight is that slice area can be saved by handling only five of the eight possible values of the smaller factor in the slice based adder. The remaining three values are handled by using the pre-adder to increment the A operand by one and signalling the `DSP48E1` blocks to subtract the value from the slice based adder. In this way the 27×3 bit multiplier will consume a very small number of LUTs, making this an attractive alternative to instantiating twice as many `DSP48E1` blocks.

It is important to note that the resulting multiplier is *not a general purpose* 27×27 bit multiplier, as the pre-adder will cause an overflow to occur if all bits are one in $A[26:3]$. Fortunately this event cannot occur when the HRFP_{16} format is used, as $A[26]$ can never be one simultaneously as $A[2]$ is one, thus avoiding this problem (see table III).

B. Normalization

As long as subnormal number support is not required, the normalization stage in a radix-2 based floating-point multiplier is trivial if implemented correctly. A two to one multiplexer shifts the mantissa at most one step depending on whether the MSB of the result is one or zero. However, if subnormal numbers are supported at the inputs of the multiplier, the normalizer must be able to shift the entire width of the result. In fact, such a normalization unit is therefore even more costly than the normalization unit found in a floating-point adder. A better alternative is to add support for normalization of one of the inputs to the multiplier before any multiplication is performed. (There is no need to normalize both numbers before multiplication as the product of two subnormal numbers will be zero.)

However, in the HRFP_{16} format, IEEE-754 style subnormal numbers are instead emulated by using an extended exponent

TABLE VI. GUARD, ROUND, AND STICKY BIT HANDLING IN THE MULTIPLIER

	Exponent	Mantissa before post-processing and rounding
	≥ 64	`1xxx xxxx xxxx xxxx xxxx xxxx GRS. ..`
Normal	≥ 64	`01xx xxxx xxxx xxxx xxxx xxxx xGRS ..`
numbers	> 64	`001x xxxx xxxx xxxx xxxx xxxx xxGR S.`
	> 64	`0001 xxxx xxxx xxxx xxxx xxxx xxxG RS`
	64	`00xx xxxx xxxx xxxx xxxx xxxx UGRS ..`
Emulated	63	`xxxx xxxx xxxx xxxx xxxx UGRS .... ..`
IEEE-754	62	`xxxx xxxx xxxx xxxx UGRS .... .... ..`
style	61	`xxxx xxxx xxxx UGRS .... .... .... ..`
subnormal	60	`xxxx xxxx UGRS .... .... .... .... ..`
numbers	59	`xxxx UGRS .... .... .... .... .... ..`
	58	`UGRS .... .... .... .... .... .... ..`
	≤ 57	Flushed to zero

Key: `x`: These bits are allowed to be either zero or one in the final rounded mantissa, `U`: this bit corresponds to the ULP of an IEEE-754 subnormal number, `G`: Guard bit, `R`: Round bit, `S`: Sticky bit, `.`: Bits that contribute to the sticky bit. `G`, `R`, `S`, and `.` are set to 0 in the final mantissa.

range. Therefore the normalization stage can thus be reduced down to a single two to one multiplexer. The only difference compared to a normal floating point multiplier is that the select signal of the multiplexer is controlled by ORing together the four MSBs of the result.

C. Rounding and Post-Processing

When no subnormal numbers are encountered, this stage essentially works in the same way as the rounding and post-processing stage of the floating-point adder described in section III-F. Or in other words, the guard, round, and sticky bit can be located in one of four different places as seen in the first part of table III.

However, unlike the adder, an IEEE-754 style subnormal result has to be handled as a special case here, as there is no guarantee that the guard, round, and sticky bit are zero when a subnormal result is encountered. This has to be taken into account by adjusting the rounding vector and rounding mask in such a way that the guard bit is located just to the right of the ULP bit of such a subnormal number. This is illustrated in table VI for the case where the exponent is between 63 and 58. There is also a special case when the exponent is 64, as subnormal emulation is only required if the two MSB bits are 0 in that case.

D. Area and frequency

The area of the different parts of the multiplier can be seen in table VII. The total area of the multiplier is 235 slice LUTs, of which 57 use both the O5 and O6 output. 6 LUTs are used as route-thrus. The total number of slice registers is 277. The maximum operating frequency of the multiplier is 305 MHz.

By removing support for subnormal emulation (i.e., the special rounding vectors/masks when the exponent is less than 64), the area can be reduced to 161 slice LUTs and 241 slice registers.

V. VERIFICATION

The correctness of the adder and multiplier has been tested by using two different test-benches. Both testbenches read IEEE-754 formatted single-precision values from an external file (actually a named pipe) and converts them into the HRFP_{16} format. After the addition or multiplication is finished, sanity checks are performed to ensure that the least significant bits of the mantissa are masked out correctly (see table VI) and then converted back to IEEE-754 format.

TABLE VII. AREAS FOR DIFFERENT PARTS OF THE MULTIPLIER IN A KINTEX-7 (XC7K70T-1-FBG484)

	Slice LUTs	Slice Registers	`DSP48E1` blocks
Mantissa multiplier	31	57	2
Normalization, rounding, and post processing	204	220	0
Total	235	277	2

The first testbench uses test vectors in the IEEE test-suite generated by FPGen [5][1]. These test vectors have proved very valuable as a quick way of verifying the correctness of small changes to the floating-point module as the coverage is nearly perfect.

The other testbench has been developed from scratch using more of a brute-force approach. This test-bench dumps all possible combinations of the sign and exponent for both operands while testing a number of different values of the mantissa. However, it is not feasible to loop through all possible mantissa combinations however. Instead, the mantissae of the two operands are set to all possible combinations in the following list:

- All but one bit set to zero
- All but one bit set to one
- All values from `0x7fff00` to `0x7fffff`
- All values from `0x000000` to `0x000100`
- One hundred random values

To verify the correct behavior of this testbench, the SoftFloat package [6] is used as a golden model.

In addition, a few other tests are run in this testbench, including testing for overflow during rounding, random small numbers (including subnormals), and random large numbers. However, as the run time of this test is measured in hours rather than seconds, this is only run after major changes had been undertaken.

VI. RESULTS

A comparison with a floating-point adder and multiplier generated by Coregen with the same number of pipeline stages as the HRFP_{16} adder and multiplier is shown in table VIII. To create a fair value for the f_{max} column, a wrapper was created with pipeline registers before and after the device under test to ensure that I/O related delays did not have any impact on timing closure. Furthermore, a number of different timing constraints were tried in order to arrive at a near-optimal f_{max} value. Note that the slice count is not included in the table since this value is largely dependent on whether the design is area constrained or not. (The author has seen this value differ by up to 50% for the same design when implemented with different area constraints.)

[1] Available for download at https://www.research.ibm.com/haifa/projects/verification/fpgen/ieeets.html

TABLE VIII. AREA/FREQUENCY COMPARISON IN A KINTEX-7 (XC7K70T-1-FBG484)

Unit	Subnormal support	Slice LUTs	Slice Registers	DSP blocks	$f$max
$HRFP_{16}$ Adder	Emulated	261	324	0	319 MHz
$HRFP_{16}$ Multiplier	Emulated	235	277	2	305 MHz
Adder [4]	Flush to zero	363	209	0	291 MHz
Multiplier [4]	Flush to zero	119	76	2	464 MHz

```
module normalizer(input wire  [23:0] x,
                  output wire [23:0] result,
                  output reg  [4:0]   s);

    always @* begin
      casez(x)
        24'b1???????????????????????: s=0;
        24'b01??????????????????????: s=1;
        24'b001?????????????????????: s=2;
        24'b0001????????????????????: s=3;
        // ....
        24'b00000000000000000000010: s=22;
        24'b00000000000000000000001: s=23;
        24'b00000000000000000000000: s=24;
      endcase
    end

    assign result = x << s;

endmodule
```

Fig. 4. Verilog Source Code for a Priority Decoder and a Shifter suitable for normalizing a 24 bit mantissa

It is important to note that the adder and multiplier provided by Coregen does not include any support for subnormal numbers, opting instead to flush those numbers to zero. (This seems to be the standard modus operandi for all recent floating-point modules optimized for FPGAs.) Therefore a straight comparison of the area and frequency numbers are misleading, especially for the multiplier. However, it is easy to arrive at a rough estimate of the area cost of a normalization module with support for subnormal numbers suitable for use in a floating-point multiplier. What is needed is essentially a priority decoder and a shifter as shown in the source code in Fig. 4. The area for this module is 80 slice LUTs, suggesting that at least 200 LUTs would be used by the Coregen multiplier if subnormal numbers were supported.

VII. FUTURE WORK

There are three obvious directions for future research in this area. First of all, it would be very interesting to implement a double precision version of the $HRFP_{16}$ format. This would be of particularly high interest for people interested in accelerating HPC algorithms with FPGAs. Similarly, a half precision format could be of interest to people who are interested in computer graphics algorithms on FPGAs. Secondly, support for other operators would be interesting to look into as well. For signal processing purposes, the most important such module is either a floating-point accumulator, or a floating-point multiply and accumulate unit. Thirdly, optimizing the code in more ways could be done, especially in terms of adding support for a configurable number of pipeline registers. The code could also be optimized for FPGAs besides the Kintex-7.

However, another research direction which would be interesting to look into is whether it would make sense to add full, or partial, support for the $HRFP_{16}$ (or a similar format) into the existing DSP blocks. For example, if one (or possibly two cascaded) DSP blocks contained support for a high-radix floating-point multiplier supporting only the round-to-zero rounding mode and no subnormal numbers, this would probably be enough for most FPGA developers. However, if a developer actually requires full support for IEEE-754, a round and post-process stage similar to the ones described in this paper could be implemented using regular slice based logic adjacent to the DSP block. This would allow such users to gain at least a partial benefit from the hardwired floating-point units while at the same time using a floating-point format more convenient for FPGA developers. This has been partially investigated in [7], but at the time that publication was written, the idea outlined in [3] had not occurred to us.

VIII. SOURCE CODE AVAILABILITY

Those who are interested in using, extending, or making comparisons with the work presented in this paper are welcome to download the source code of the floating point adder, multiplier and testbenches at http://users.isy.liu.se/da/ehliar/hrfp/ which is published under released under the terms of the X11 license. This license should enable convenient use in both commercial and non-commercial situations.

IX. RELATED WORK

Floating-point operators have been of interest to the FPGA community for a long time. However, only a few publications discuss the use of a high radix. One of the first such publications is [2], which shows the area-advantage of a floating point adders and multipliers which use a high-radix floating-point format with no compatibility with IEEE-754. Another publication is [8], which shows that it is feasible to build a floating-point multiply and accumulate-unit with single cycle accumulation if a very high radix (2^{32}) is used for the floating point format used in the accumulator.

In regards to the traditional floating-point operators such as addition, subtraction, and multiplication, it would be remiss not to mention the work done by the FPGA vendors themselves, where [4] and [9] are two typical examples. The focus here seems to be to supply robust floating-point operators with near full IEEE-754 compliance. The main exception is that subnormal numbers are not implemented. Unfortunately the internal implementation details are not discussed in these publications, but the tables with area and frequency numbers are very useful in order to establish a baseline which other reported results should improve upon.

Another well known resource for floating-point arithmetic in FPGAs is FloPoCo [10]. This project concentrates on floating point operators that are typically not implemented in mainstream processors such as transcendental functions and non-standard number formats. The basic philosophy of this project is that an FPGA based application should not limit itself to the floating-point operators that are common in microprocessors (e.g., adders/subtracters, multipliers, and to a lesser extent, division and square root).

One of the most interesting approaches to floating-point arithmetic in FPGAs is [11] and [12], where a tool has

been developed that takes a datagraph of floating-point operations and generates optimized hardware for that particular datapath. In essence, a fused datapath is created where the complete knowledge of the input and output dependencies of each floating-point operator has been taken into account to reduce the hardware cost. For example, if an addition is followed by another addition, some parts of the normalization/denormalization steps may be omitted. In the case of a 48 element dot product, the datapath compiler showed an impressive reduction of area from 30144 ALUTs and 36000 registers to 12849 ALUTs and 20489 registers [11]. In another case a Cholesky decomposition was reduced from 9898 ALUTs and 9100 registers down to 5299 ALUTs and 6573 registers [12]. However, a drawback of this work is that the arithmetic operations are not fully compliant with IEEE-754 (although in practice, a slight increase in the accuracy of the result can be expected [13]).

A key advantage to floating-point units in FPGA is that the precision of the unit can be adjusted to match the precision requirements of the application. In for example [14] it is shown that a double precision multiplier can be reduced from nine DSP48 blocks down to six DSP48 block if a worst case error of 1 ulp is acceptable instead of 0.5 ulp as required by IEEE-754.

Finally, it should also be noted that Altera recently announced that the Stratix-10 and Arria-10 FPGAs will have DSP blocks with built-in support for single-precision IEEE-754 floating-point addition and multiplication. These hard blocks are a welcome addition in any FPGA system with high floating point computation requirements. To reduce the cost of these floating point operators slightly, no support for subnormal numbers are present.

X. Conclusions

In this paper we have demonstrated that using a custom floating-point format based on radix-16 can be beneficial in an FPGA when support for IEEE-754 style subnormal numbers are required. The combinational area of a high-radix floating-point adder is shown to be around 30% smaller than the combinational area of a floating-point adder generated by Coregen, even though subnormal numbers are handled by flushing to zero in the later case.

The multiplier on the other hand has a combinational area which is almost twice as large as the multiplier generated by Coregen, although it is likely that the Coregen provided multiplier would be of comparable size if support for subnormals was added to it. If subnormal numbers are not required however, the HRFP_{16} operators presented in this paper can still be useful in situations where relatively few floating point multipliers are required compared to the number of floating point adders.

Acknowledgments

Thanks to the FPGen team at IBM for clearing up a misunderstanding on my side regarding exception handling in IEEE-754. Thanks also to Madeleine Englund for many interesting discussions regarding high-radix floating-point arithmetic in FPGAs.

References

[1] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 12:1–12:41, May 2008. [Online]. Available: http://doi.acm.org/10.1145/1353445.1353446

[2] B. Catanzaro and B. Nelson, "Higher radix floating-point representations for fpga-based arithmetic," in *Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on*. IEEE, 2005, pp. 161–170. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1508536

[3] P.-M. Seidel, "High-radix implementation of ieee floating-point addition," in *Computer Arithmetic, 2005. ARITH-17 2005. 17th IEEE Symposium on*. IEEE, 2005, pp. 99–106. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1467628

[4] Xilinx, "Pg060: Logicore ip floating-point operator v6.2," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_2/pg060-floating-point.pdf

[5] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "Fpgen - a test generation framework for datapath floating-point verification," in *In Proc. IEEE International High Level Design Validation and Test Workshop 2003 (HLDVT03*, 2003, pp. 17–22.

[6] J. R. Hauser, "Softfloat release 2b." [Online]. Available: http://www.jhauser.us/arithmetic/SoftFloat.html

[7] M. Englund, "Hybrid floating-point units in fpgas," Master's thesis, Linköping University, 2012.

[8] A. Paidimarri, A. Cevrero, P. Brisk, and P. Ienne, "Fpga implementation of a single-precision floating-point multiply-accumulator with single-cycle accumulation," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009, pp. 267–270. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5290905

[9] Altera, "Ug01058-6.0: Floating-point megafunctions," 2011. [Online]. Available: http://www.altera.com/literature/ug/ug_altfp_mfug.pdf

[10] F. de Dinechin and B. Pasca, "Custom arithmetic datapath design for fpgas using the flopoco core generator," *Design & Test of Computers, IEEE*, vol. 28, no. 4, pp. 18–27, 2011.

[11] M. Langhammer, "Floating point datapath synthesis for fpgas," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*. IEEE, 2008, pp. 355–360. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4629963

[12] M. Langhammer and T. VanCourt, "Fpga floating point datapath compiler," in *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*. IEEE, 2009, pp. 259–262. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5290908

[13] I. Berkeley Design Technology, "An independent analysis of floating-point dsp design flow and performance on altera 28-nm fpgas," 2012. [Online]. Available: http://www.altera.com/literature/wp/wp-01187-bdti-altera-fp-dsp-design-flow.pdf

[14] M. K. Jaiswal and N. Chandrachoodan, "Efficient implementation of ieee double precision floating-point multiplier on fpga," in *Industrial and Information Systems, 2008. ICIIS 2008. IEEE Region 10 and the Third international Conference on*. IEEE, 2008, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4798393