A Library of Parameterized Floating Point Modules and Their Use*

Pavle Belanović and Miriam Leeser

Dept of Electrical and Computer Engineering
Northeastern University
Boston, MA, 02115, USA
{pbelanov,mel}@ece.neu.edu

Abstract. We present a parameterized floating point library for use with reconfigurable hardware. Our format is both general and flexible. All IEEE formats are a subset of our format, as are all previously published floating point formats for reconfigurable hardware. We have developed a library of fully parameterized hardware modules for format control, arithmetic operations and conversion to and from any fixed point format. The format converters allow for hybrid implementations that combine both fixed and floating point calculations. This permits the designer to choose between the increased range of floating point and the increased precision of fixed point within the same application. We illustrate the use of this library with a hybrid implementation of the K-means clustering algorithm applied to multispectral satellite images.

1 Introduction

Many image and signal processing applications benefit from acceleration with reconfigurable hardware. This acceleration results from the exploitation of fine-grained parallelism available in reconfigurable hardware. These applications are typically implemented initially in software using Matlab or C code, and variables and operations are assumed to be floating point implemented using the IEEE standard 754. Reconfigurable implementations that exhibit speedup are possible if fixed point or a more flexible floating point representation than full IEEE precision is used, thus allowing for greater parallelism. We present a parameterized floating point format and library of arithmetic operators to support it. This library supports a broad range of floating point formats, including the IEEE standard formats as a subset.

1.1 Fixed Point And Floating Point Arithmetic

Representation of every numeric value, in any number system, is composed of an integer and a fractional part. The boundary that delimits them is called the radix point. The fixed point format for representing numeric values derives its name

* Technical Report NEU-ECE-RPL-2002-01

from the fact that in this format, the radix point is fixed in a certain position. For integers this position is immediately to the right of the least significant digit.

In scientific computation, it is frequently necessary to represent very large and very small values. This is difficult to achieve using the fixed point format because the bitwidth required to maintain both the desired precision and the desired range grows large. In such situations, floating point formats are used to represent real numbers. Floating point formats resemble scientific notation, such as -3.2004×10^{17}. Every floating point number can be divided into three fields: sign s, exponent e, and fraction f. Using the binary number system, it is possible to represent any floating point number as:

$$(-1)^s \times 1.f \times 2^{e-BIAS} \tag{1}$$

Note that the exponent is *biased*, meaning that the stored value is shifted from 0 by a given value that depends on the bitwidth of the exponent field in the particular format. Also, the *fraction* represents the portion of the mantissa to the right of the radix point, while the term *mantissa* refers to the fractional and integer part.

A natural tradeoff exists between smaller bitwidths requiring fewer hardware resources and higher bitwidths providing better precision. Also, within a given total bitwidth, it is possible to assign various combinations of bitwidths to the exponent and fraction fields, where wider exponents result in higher range and wider fractions result in better precision.

The most widely used format for floating point arithmetic is the IEEE standard 754 [1]. This standard details four floating point formats - basic and extended, each in single and double precision bitwidths. The IEEE single precision format is the same as shown in Equation 1 with BIAS = 127, 8 bits for the exponent and 23 bits for the fraction, or a total of 32 bits. In IEEE format, numbers are normalized, and only the fractional part is stored.

Optimal implementations of algorithms frequently do not require the bitwidths specified by the IEEE standard. Often, much smaller bitwidths than those specified in the 754 standard are sufficient to provide the desired precision. Reduced bitwidth implementations require fewer resources and thus allow for more parallel implementations than using the full IEEE standard format. In custom hardware designs, it is possible, and indeed desirable, to have full control and flexibility over the exact floating point format implemented. Our library provides this flexibility.

1.2 Related Work

Investigation of using FPGAs to implement floating point arithmetic by Fagin et al [4] showed that implementing IEEE single precision operators was possible, but also impractical on FPGA technology in 1994. Area was the critical constraint, with the authors reporting that no device in existence could contain a single multiplier circuit. Nevertheless, the authors suggest that custom, smaller, formats present a viable solution for FPGA architectures.

This line of thought was expanded on by the significant work of Shirazi et al [8], who provide two custom floating point formats (16 bits and 18 bits total) as well as addition, subtraction, multiplication and division operators in those formats. A series of work [6, 5, 9, 7] ensued, all considering only IEEE single precision formats, usually with no rounding capabilities except in [5], where the authors implement rounding to nearest. Work on floating point arithmetic was mostly centered on creating operator modules and optimizing them for area in order to utilize the expanding capabilities of new FPGAs.

Recent work by Dido et al[2] discusses optimization of datapaths, especially in image and video processing applications, by providing flexible floating point formats. Their work presents a non-general floating point format that is primarily used to represent the application specific range of scaled integer values through a format that resembles floating point. Their format is unsigned and unnormalized, and cannot represent the IEEE single precision formats.

The floating point formats presented here are a superset of all the previously published floating point formats. They are both general and flexible. All the IEEE formats can be represented, as can all other instances of exponent and mantissa bitwidths.

2 Hardware Modules

We provide a library of parameterized components that are pipelined and cascadable to form pipelines of floating point operations. Each component is parameterized by exponent and mantissa bitwidths. Each component has a ready and a done signal to allow them to easily be assembled into larger designs. Some error handling is supported, and errors detected are propagated to the end of the pipelined design.

The hardware modules presented fall into three categories. Format control modules support denormalizing, round and normalizing operations. The arithmetic operators include addition, subtraction and multiplication. The format conversion modules convert between fixed point and floating point representations. The names, functions and latencies (in clock cycles) of all modules presented are shown in Table 1.

In order to operate on any custom floating point format, all the hardware modules have been parameterized. Every module's VHDL description accepts two parameters: *exp_bits* and *man_bits*, representing exponent and mantissa bitwidth respectively. These two values are sufficient to describe any floating point format, since the third field, s, always has bitwidth of 1. The total bitwidth of the format is $1 + exp\_bits + man\_bits$. These parameters allow for the creation of a wide range of different modules through a single VHDL description. Values of the parameters help resolve the description at compile time and ensure synthesis of the correct amount of logic needed to perform the function of the module for the given format.

Table 1. Hardware modules

Module	Function	Latency
`denorm`	Introduction of implied integer digit	0
`rnd_norm`	Normalizing and rounding	2
`fp_add`	Addition	4
`fp_sub`	Subtraction	4
`fp_mul`	Multiplication	3
`fix2float`	Unsigned fixed point to floating point conversion	4
	Signed fixed-point to floating-point conversion	5
`float2fix`	Floating point to unsigned fixed point conversion	4
	Floating point to signed fixed point conversion	5

2.1 Denormalizing

The *normalized* format of a floating point value is defined as the format in which exactly one non-zero digit forms the integer part of the mantissa. In binary, the integer part of every normalized floating point number is '1'. Since this is redundant information, only the fractional part of the number is stored; the integer part is referred to as the *implied '1'*. While this provides efficient storage, the implied '1' is necessary to carry out arithmetic operation on the number and must be explicitly represented before any operations involving the number.

The `denorm` module inserts the implied '1' into the representation, unless the value being processed is zero. In the latter case, it will insert a '0', since floating point formats represent the value zero by all zero exponent and mantissa. This module is purely combinational and is not pipelined.

2.2 Rounding And Normalizing

After arithmetic operations have been performed on the floating point value, it is necessary to renormalize. During processing, mantissa bitwidth may increase, due to the introduction of guard digits during addition, for example. To normalize a floating point value, we must remove the implied '1' which was introduced by the `denorm` module and reduce the fractional bitwidth to that specified by the particular floating point format. Reduction of bitwidth introduces the need for rounding. The IEEE standard specifies four rounding modes: (1) round to zero, (2) round to nearest, (3) round to positive infinity ($+\infty$), and (4) round to negative infinity ($-\infty$).

We support the first two modes of rounding in the `round_norm` module, which handles rounding and normalizing of floating point values. The input to this module is any floating point value with the implied '1' explicitly represented and the mantissa bitwidth equal to or larger than that specified by the floating point format in use. The output of the module is the normalized form of the input value, rounded to nearest (default) or to zero.

2.3 Addition and Subtraction

Addition is one of the most computationally complex operations in floating point arithmetic. The algorithm of the addition operation for floating point numbers is composed of four steps:

1. ensure the operand with larger magnitude is on input 1 (`swap`),
2. align the mantissas (`shift_adjust`),
3. add or subtract the mantissas (`add_sub`),
4. shift the result mantissa right by one bit and increment the exponent if addition overflow occurred (`correction`).

Each of the four steps of the algorithm is implemented in a dedicated module. The four sub-modules are assembled into the overall `fp_add` module as shown in Figure 1. The subtraction operation is similar to the addition operation, as $A - B = A + (-B)$. Thus, we use a slightly modified addition module to perform subtraction. The sign bit of B simply needs to be inverted.

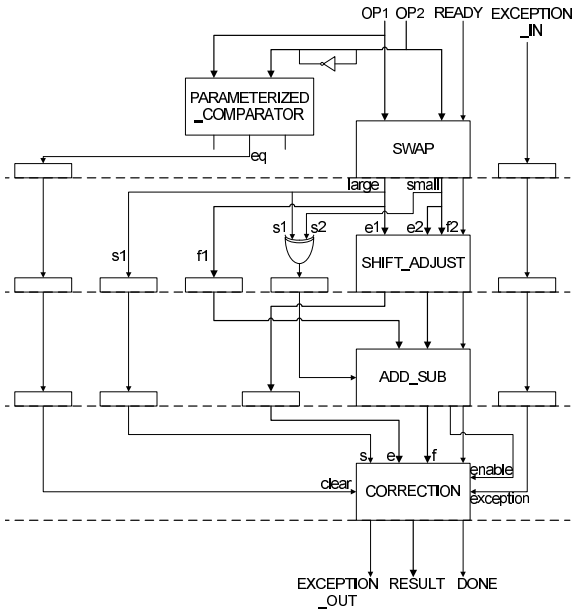

Fig. 1. Floating point addition

2.4 Multiplication

Unlike in fixed point, in floating point arithmetic multiplication is a relatively straight-forward operation compared to addition. This is due to the sign-exponent-magnitude nature of the floating point format. The sign of the product is the

exclusive OR (XOR) of the operand signs. The exponent of the product is the sum of the operand exponents. The mantissa is the product of the operand mantissas. Note that the operations on all three fields of the floating point format are independent and can be implemented in parallel. The structure of the floating point multiplier is shown in Figure 2.

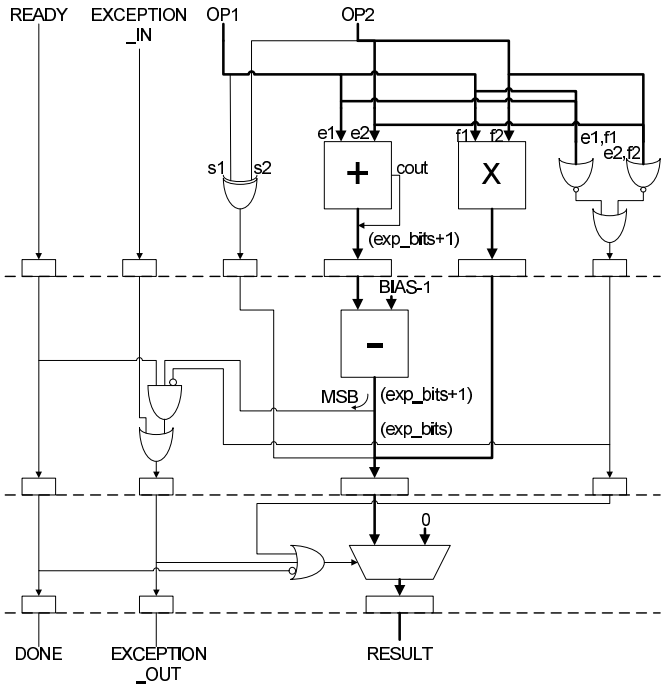

Fig. 2. Floating point multiplication

2.5 Fixed-To-Float and Float-To-Fixed Conversion

Custom hardware designs, which are most likely to profit from parameterization of floating point arithmetic, may derive extra efficiency from performing some parts of the algorithm in fixed point and others in floating point arithmetic. Hardware modules for converting between the two representations are thus required to implement hybrid designs.

The module `fix2float` converts from the fixed point representation of a value to its floating point representation, again rounding to nearest (default) or 0. This module has an additional parameter, *fix_bits*, to specify the bitwidth of the fixed point value at the input. The output of the module is the floating point representation of the value, and its format is again specified by *exp_bits* and *man_bits*. Two versions of the `fix2float` module have been developed: one for

converting from signed and the other from unsigned fixed point representations. The signed version is more complex due to handling of the two's complement representations of the input and this version hence has the longer latency of 5 clock cycles, as opposed to 4 clock cycles for the unsigned version (see Table 1).

The module `float2fix` implements the inverse function to that of the `fix2float` module: conversion from the floating point representation of a value to its fixed point representation. Similarly, rounding to nearest (default) and 0 are supported and the additional parameter *fix_bits* is in use to specify the width of the fixed point output. As before, two versions of the `float2fix` module exist: one for converting to signed and the other to unsigned fixed point representation.

2.6 Results

All these modules are specified in VHDL and implemented on the Wildstar reconfigurable computing engine from Annapolis Microsystems, using a Xilinx XCV1000 FPGA. Synthesis results for parameterized arithmetic operator modules are presented in Table 2, for a set of floating point formats labeled A0 through E2.

Table 2. Operator synthesis results

Format	Total bits	Exponent	Fraction	Area	
				fp_add	fp_mul
A0	8	2	5	39	46
A1	8	3	4	39	51
A2	8	4	3	32	36
B0	12	3	8	84	127
B1	12	4	7	80	140
B2	12	5	6	81	108
C0	16	4	11	121	208
C1	16	5	10	141	178
C2	16	6	9	113	150
D0	24	6	17	221	421
D1	24	8	15	216	431
D2	24	10	13	217	275
E0	32	5	26	328	766
E1	32	8	23	291	674
E2	32	11	20	284	536

The quantities for the area of each instance are expressed in slices of the Xilinx XCV1000 FPGA. Results for the `fp_add` module in Table 2 also represent the `fp_sub` module, which has the same amount of logic. The results in Table 2 show growth in area with increasing total bitwidth, for both modules. This growth is represented graphically in Figure 3.

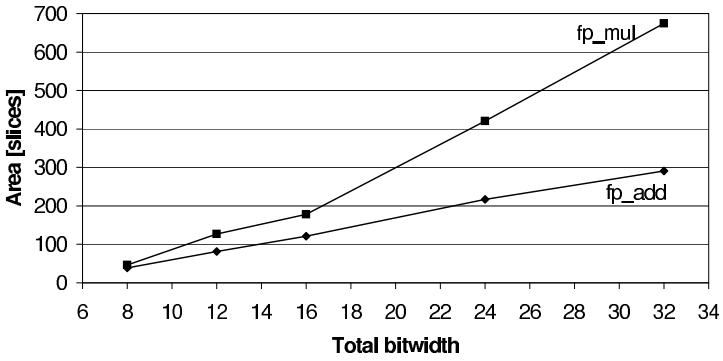

Fig. 3. Growth of area with increasing bitwidth

To implement a single precision IEEE adder, the designer would use three parameterized modules: `denorm`, `fp_add` and `round_norm`. The floating point format is 1 sign bit, $exp\_bits = 8$ and $man\_bits = 23$. Design E1 in Table 2 corresponds to the adder. The total design mapped to a Xilinx Virtex 1000 takes up 305 slices, or just under 2.5% of the chip.

3 Application

We have used the parameterized hardware modules to implement a floating point implementation of k-means clustering on multispectral satellite images. The K-means clustering algorithm consists of assigning each pixel in the image to one of K clusters, as well as accumulating the values of all pixels assigned to each cluster. These arithmetic operations can be implemented in fixed point format, floating point format, or a combination of the two, with format converters in suitable positions. A fixed point implementation of K-means clustering on reconfigurable hardware [3] exhibits significant speedup over a software implementation.

The k-means algorithm processes images one pixel at a time, calculating the distance between the pixel and each of the K cluster centers. All K distances are then compared and the pixel is assigned to the closest cluster center. An accumulator and a counter are associated with each cluster. Once a pixel is assigned to a certain cluster, the value of the pixel is added to the cluster's accumulator and the cluster's counter is incremented.

The distance calculation is performed in 1-norm (Manhattan norm). The operations performed are: (1) subtract each pixel from each cluster center, each channel (dimension) processed separately, (2) take the absolute value of each difference, and (3) sum all the absolute values (dimensions). The distance of the pixel to the cluster center in the 1-norm is the resulting sum. The remaining operations are comparison of distances and accumulation.

The algorithm has been partitioned so the distance calculation is performed in floating point arithmetic with 5 exponent and 6 fraction bits (1-5-6 format), while the comparison and accumulation operations are performed in 12-bit unsigned fixed point format. Input data is in 12-bit unsigned fixed point format, so it needs to be converted to the 1-5-6 floating-point representation. Similarly, the distance signal in 1-5-6 floating-point format needs to be converted into 12-bit unsigned fixed point representation to be used by the comparison and accumulation circuits.

3.1 Results

The hybrid design of the K-means clustering algorithm resulted in a successful implementation using our library of parameterized modules. The design occupied 10,883 slices, or 88% of one of the three processing elements on the Wildstar board. This hybrid fixed and floating point implementation was tested against an existing, purely fixed point implementation and yielded satisfactory results, shown in Figure 4.

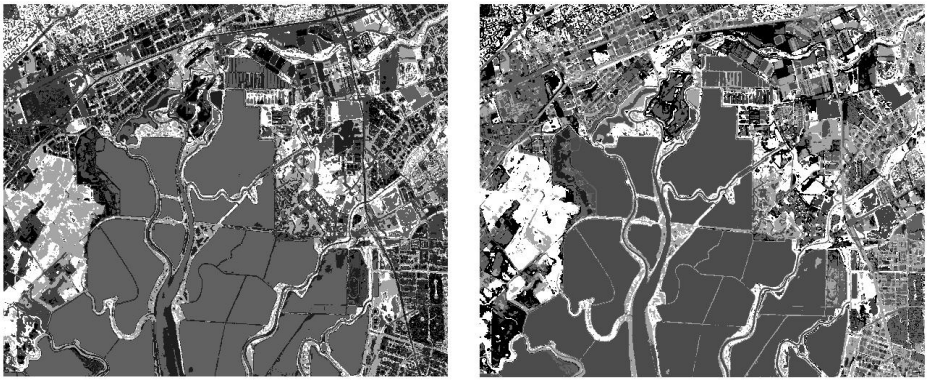

Fig. 4. Clustered image: pure fixed point (left) and hybrid (right)

The floating point K-means distance calculation is significantly larger than the equivalent fixed point implementation. In the fixed point implementation, Manhattan distance was used to save space. In the floating point version, 2-norm (Euclidean) distance will not take up significantly more space than the 1-norm calculation. The advantage of using the 2-norm distance is that the results match more closely with those achieved in software.

The hardware modules described in Section 2 lend themselves to the creation of finely customized datapaths. First, they give the hardware designer full freedom to implement various sections of the algorithm in the most suitable arithmetic form, be it in fixed or floating point representation. Thus, the input data format does not influence the way the data is processed. Similarly, the results of processing can be returned in any format.

Also, all bitwidths in the datapath, whether in fixed or floating point representation, can be optimized to the precision required for that signal. Hence, designing with our library of parameterized modules avoids expensive inefficiencies that are inherent in designs that operate only in the IEEE standard formats. In fact, such inefficiencies occur in any design that is restricted to using a small set of particular formats, even if these are custom. Using our library of parameterized modules provides the finest-grain control possible over datapath bitwidths. Finally, when using floating point arithmetic, the designer has full control to trade off between range and precision. Because our modules are fully parameterized, the boundary between the exponent and the fraction for the same total bitwidth is flexible. Thus, with a wider exponent field, the designer has larger range for a value while sacrificing precision. Similarly, to increase the precision of a signal at the cost of reduced range, the designer chooses a narrower exponent and wider fraction field.

This research was funded in part by a grant from Los Alamos National Laboratory.

References

1. IEEE Standards Board and ANSI. IEEE Standard for Binary Floating-Point Arithmetic, 1985. IEEE Std 754-1985.
2. J. Dido, N. Geraudie, L. Loiseau, O. Payeur, Y. Savaria, and D. Poirier. A Flexible Floating-Point Format for Optimizing Data-Paths and Operators in FPGA Based DSPs. In *International Symposium on Field-Programmable Gate Arrays*, pages 50–55. ACM, ACM Press, February 2002.
3. M. Estlick, M. Leeser, J. Theiler, and J. Szymanski. Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware. In *International Symposium on Field-Programmable Gate Arrays*, pages 103–110. ACM, February 2001.
4. B. Fagin and C. Renard. Field Programmable Gate Arrays and Floating Point Arithmetic. *IEEE Transactions on VLSI Systems*, 2(3), September 1994.
5. W. B. Ligon III, S. McMillan, G. Monn, K. Schoonover, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998.
6. L. Louca, T. A. Cook, and W. H. Johnson. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In K. L. Pocek and J. Arnold, editors, *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 107–116, April 1996.
7. I. Sahin, C. S. Gloster, and C. Doss. Feasibility of Floating-Point Arithmetic in Reconfigurable Computing Systems. In *2000 MAPLD International Conference*, 2000.
8. N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, April 1995.
9. I. Stamoulis, M. White, and P. F. Lister. Pipelined Floating-Point Arithmetic Optimized for FPGA Architectures. In *9th International Workshop on Field Programmable Logic and Applications*, volume 1673 of *LNCS*, pages 365–370, August-September 1999.