Floating Point Unit Generation and Evaluation for FPGAs

Jian Liang and Russell Tessier
Department of Electrical and Computer Engineering
University of Massachusetts
Amherst, MA. 01003.

Oskar Mencer
Computer Sciences Center
Bell Labs, Lucent
Murray Hill, N.J. 07974

Abstract

Floating point units form an important component of many reconfigurable computing applications. The creation of floating point units under a collection of area, latency, and throughput constraints is an important consideration for system designers. Given the range of possible tradeoffs, most commercial or academic floating point libraries for FPGAs provide a small fraction of possible floating point units. In contrast, the floating unit generation approach outlined in this paper allows for the creation of more than 200 different floating point units, with differing area, throughput, and latency characteristics. These variations are supported through selection of a floating point architecture and the use of floating point unit pipelining. Each of these floating point units can be generated with a variable number of bits for the mantissa and the exponent.

Given requirements on throughput, area and latency, our generation flow automatically chooses the proper algorithm and architecture to create a floating point unit which fulfills design requirements. Our approach is fully integrated into standard C++ using **ASC***, a stream compiler for FPGAs, and the underlying PAM-Blox II module generation environment [13]. The floating point units created by our approach are competitive in size and performance with ones created by commercial vendors.*

1 Introduction

With gate counts approaching ten million gates, FPGAs are quickly becoming suitable for major floating point computations. However, to date, few comprehensive tools to allow for floating point unit tradeoffs have been developed. Most commercial or academic floating point libraries provide only a small number of floating point modules with fixed parameters of bit-width, area, and speed. With this limitation, user designs must be modified to meet the available units.

The balance between FPGA floating point unit resources and performance is influenced by subtle context and design requirements. Generally, implementation requirements are influenced by throughput, latency and area.

1. FPGAs are often used in place of software due to inherent parallelism and specialization. For data-intensive applications, data *throughput* is critical.

2. If floating point computation is in a dependent loop, computation *latency* could be an overall performance bottleneck.

3. In typical FPGA designs, only a few floating point units will be on the critical path. For these non-critical units, it may be possible to limit unit performance in an effort to reduce resource area.

Although bitwidth variation provides some flexibility, this parameter alone cannot address all possible tradeoffs.

The VLSI design community has developed a variety of floating point algorithms, architectures, and pipelining approaches. For example, a *two-path* floating point adder [6] was introduced to trade area for latency and other architectures [1, 18, 3] were designed to reduce critical path delay. With modification, these techniques can be applied to FPGAs. To better evaluate the floating point unit design space on FPGAs, we have developed a floating point unit generator which can generates more than 200 different floating point adders, subtractors, multipliers and dividers. Three trade-off levels can be explored: the architectural level, the floating point algorithm level, and the floating point representation level. Each of these require examination of FPGA-specific features and techniques including:

1. Different floating point units can be built by using various combinations of carry chains, LUTs, tri-state buffers and flip flops to obtain different area, latency, and throughput values. These features lead to parallel and serial versions of units.

2. The implementation of a variety of well-known floating point algorithms can be considered. These include

standard 3-stage floating point addition [17], two-path addition [6], Leading-One-Detection (LOD) [18], and Leading-One-Prediction (LOP) [3].

3. The floating point representations can be customized by using different sign modes for both mantissa and exponent. Each generated floating point unit can also contain a mantissa and exponent with custom bit widths.

Our floating point unit generation tool is integrated into ASC [15], and a friendly and fully automatic design flow.

Section 2 provides background on floating point unit design and shows the trade-offs of floating point computation on FPGAs. The design flow and the algorithms are introduced in Section 4. The area and performance of different trade-off options are presented in Section 5. The application of our generator on a Wavelet filter is presented in Section 6. We summarize the paper in Section 7.

2 Background

2.1 Floating Point Representation

Standard floating point numbers are represented using an exponent and a mantissa in the following format:

$$(sign-bit)mantissa \times base^{exponent+bias}$$

The *mantissa* is a binary, positive fixed-point value. Generally, the decimal point is located after the first bit, m_0, as $mantissa = \{m_0.m_1m_2...m_n\}$, where the m_i is the i_{th} bit of the mantissa. The floating point number is “normalized” when m_0 is one.

The *exponent* combined with a *bias* sets the range of representable values. The common value for the bias is -2^{k-1}, where k is the bit width of the exponent.

The base of the representation sets the granularity of shifting and rounding. In the IEEE754 standard [9], the base is two. Other units use a larger number to reduce the latency of the shift operation, which is a critical part of most floating point arithmetic units. For example, 2^4 is chosen in the IBM S/370 [23].

The IEEE standard floating point representation makes floating point unit implementation portable and the precision of the results predictable. However, application specific circuits are not required to use a uniform representation of numbers at the bit level. Adapting the floating point representation of the algorithm, architecture, and bit-level offers significant potential for optimization.

2.2 Floating Point Implementations in FPGAs

A few efforts have made to build floating point units using FPGAs. Several bit-width-scalable floating point arithmetic architectures [12, 11, 21] have been evaluated. Fagin and Renard [5] implemented a floating point adder and multiplier that met IEEE754 [9] floating point standards. Most commercial floating point libraries provide units that comply with the IEEE754 standard [4, 16]. Luk [7] showed that in order to cover the same dynamic range, a fixed point design must be five times larger and 40% slower than the floating point design. A floating point library containing units with parameterized bit width units was described in [2]. In this library, the mantissa and exponent bit width can be customized. This library also includes a converter which can convert between fixed point numbers to floating point numbers. In [10], a floating point library with variable bit width units is presented. A square root unit has been included in this library. The floating point units are arranged in fixed pipeline stages.

2.3 Floating Point Algorithms

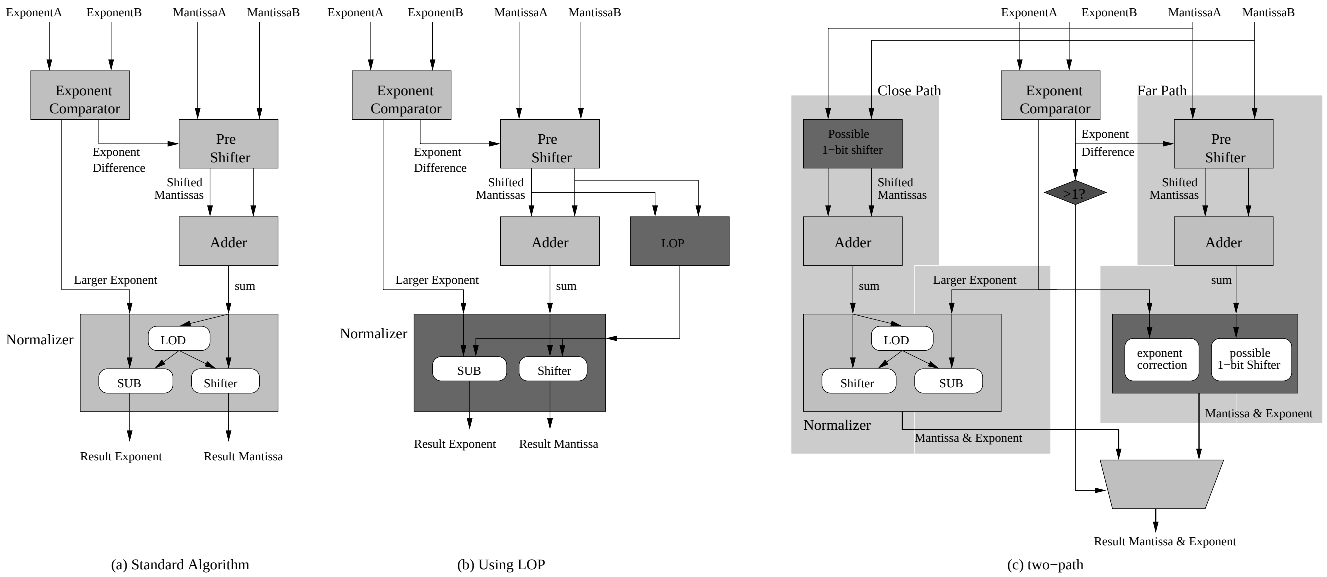
Although the multiplication and division of floating point numbers are straightforward with respect to their corresponding fixed point units, addition and subtraction are significantly more complex operations in the floating point domain. Three floating point add/sub algorithms are briefly introduced in this section: standard [17], 2-path [6] and leading-one predictor (LOP) [3]. Since floating point addition is substantially more difficult than floating point multiplication we concentrate our discussion on addition.

The standard pipelined floating point addition algorithm [17] consists of five steps:

1. Exponent difference
2. (Pre) Shift for mantissa alignment
3. Mantissa add/subtraction
4. (Post) Shift for normalization
5. Rounding

The implementation of these steps defines the area, latency, and performance of the floating point unit. To illustrate comparisons, we consider the block diagrams of the floating point adders shown in Figure 1.

The area-efficient standard floating adder is implemented as shown in Fig.1(a). The exponents of the two input operands, *exponentA* and *exponentB* are fed into the *exponent comparator*. In the *pre-shifter*, a new mantissa is created by right shifting the smaller exponent by the difference of the exponents so that the resulting two mantissas are aligned and can be added. An overflow will result when both mantissas have *one*s as most significant bits. In this case, the resulting mantissa will be shifted one bit to the right and the exponent will be decreased by one.



(a) Standard Algorithm

(b) Using LOP

(c) two−path

Figure 1. Floating Point Addition Algorithms

The *normalizer* transforms the mantissa and exponent into normalized format. It first uses a Leading-One-Detector (LOD) circuit to locate the position of the most significant *one* in the mantissa. Based on the position of the leading one, this resulting mantissa is left-shifted by an amount subsequently deducted from the exponent.

In this algorithm, the exponent comparator is a subtractor and multiplexer. It requires about $2 \times n$ LUTs, where n is the exponent bit width. The pre-shifter is implemented using a barrel shifter and its size is about $m \times log(m)$ LUTs, where m is the bit width of the mantissa. The size of mantissa adder depends on the architecture and sign mode. Using a ripple adder for the unsigned mantissa, it is about m LUTs. In the normalizer, the LOD is nearly the same size as the mantissa adder. The shifter is equal to the pre-shifter and the SUB is about the same size as the exponent comparator. Overall, the size of the normalizer is about the size of the sum of the other three components.

Figure 1(b) shows the floating point adder using the Leading-One-Predictor (LOP) [8, 20, 22]. This floating point adder implementation requires larger area than the standard adder but exhibits improved latency. The primary difference is the replacement of the leading-one detector (LOD) with a leading-one predictor (LOP). Since the LOP circuit can be executed in parallel with mantissa addition, overall latency can be reduced.

The two-path adder [6], shown in Fig.1(c), has two parallel data paths. This implementation exhibits the smallest latency of the three adders, due to the elimination of a shifter from the critical path, at the cost of additional mapping area. When the exponents of the two input numbers are larger than 1, the *far* path, on the right in Fig.1(c), is taken. Otherwise, the *close* path on the left is taken. After the alignment, one of the mantissas will be reduced and the position of the leading one will change at most one bit. This implementation eliminates the long shifter of the normalizer. In the close path, the mantissa will be shifted at most one bit for alignment. A long shifter is not necessary either. In Fig.1(c), the shaded blocks indicate the different components to the standard algorithm. The 2-path algorithm uses two data paths which almost doubles area usage.

3 FPGA Floating Point Tradeoffs

In this section, floating point unit implementation tradeoffs using FPGA architectural features are evaluated. Some floating point unit parameters (sign mode, normalization, and rounding) offer special opportunities for area and latency reduction.

3.1 Standard Floating Point Adder

The exponent comparator of the adder shown in Fig. 1(a) is a subtractor. The pre-shifter is implemented using a barrel shifter. The normalizer requires a second shifter to move the resulting number into normalized format. The mantissa adder forms the kernel of this unit.

A ripple adder using an FPGA carry chain (e.g. Virtex) is an efficient mantissa adder implementation. Alternately, a serial adder can be used to minimize area.

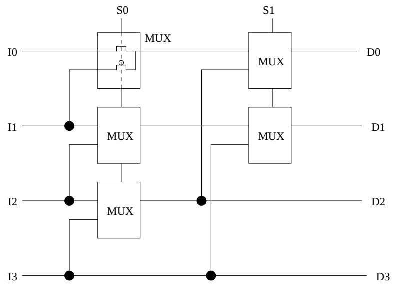


Figure 2. Barrel Shifter

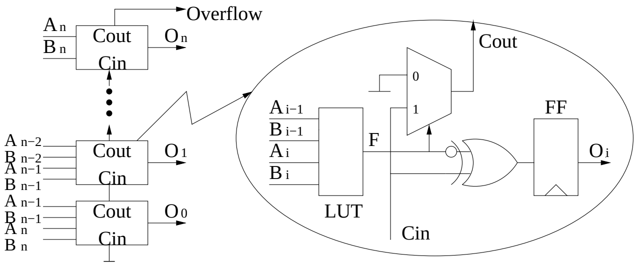


Figure 3. Mapping LOP into Virtex CLB

3.1.1 2-Path Floating Point Adder

As mentioned in Section 2, 2-path [6] and LOP [8] adder implementations shorten the critical path of floating point adders by requiring extra shifter logic. Fig. 2 shows a 4-bit barrel shifter implementation where the $I_{0,1,2,3}$ are the input bits, $D_{0,1,2,3}$ are the output bits and the $S_{0,1}$ are the shift index. In Virtex FPGAs, each multiplexer bit occupies a 4-input LUT. In the case of a 32-bit shifter, shifting will require 5 LUT delays.

3.2 LOP Floating Point Adder

This algorithm [3] requires use of the leading-one predictor. A subtractor can be used because the leading one position in addition can be determined from the larger operand. Fig. 3 shows the implementation of a LOP unit in a Virtex CLB. A_i and A_{i-1} are two consecutive bits of the minuend, and the B_i and B_{i-1} are consecutive bits of the subtrahend. The required LUT function is

$$F = (A_i \oplus B_i)\&\overline{(\overline{A_{i-1}}\&B_{i-1})}$$

This implementation requires an output ripple from the most significant bit to the least significant bit. Using the carry chain for this function, the LOP has nearly the same delay as the mantissa adder.

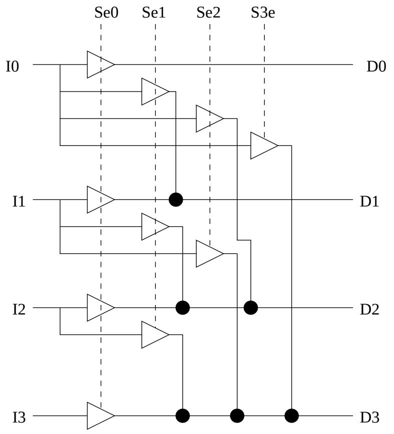


Figure 4. Tri-state Buffer Shifter

3.3 Pipelining

The pipelining of the floating point adders can be changed to realize area and throughput tradeoffs. Removing registers results in a shorter pipeline and lower throughput, but simpler overall control circuitry.

All components were carefully hand-pipelined at the block level. For all three adders, the exponent comparator requires one stage, the pre-shifter requires $log(bitwidth)$ stages, the mantissa add/sub unit requires one stage, the LOP/LOD requires one stage, and the normalizer requires $log(bitwidth)$ stages. The $log(bitwidth)$ stages of the pre-shifter and the normalizer shown in Fig. 2 require a pipeline register after every multiplexer. The number of pipeline stages per unit can be tuned in our generator through the use of input parameters.

3.4 Tri-state Buffer Usage

Tri-state buffers can used instead of LUTs to efficiently build long shifters. As shown in Fig. 4, a tri-state buffer shifter has approximately constant delay. When fully pipelined, the tri-state buffer has only one stage, compared with the $log(bitwidth)$ stages of the barrel shifter. The availability of tri-state buffers are generally limited in contemporary FPGAs.

3.5 Floating Point Improvements

FPGA specialization allows for potential tradeoffs in floating point unit sign mode, normalization, and rounding implementation. These tradeoffs are in addition to bitwidth tradeoffs commonly found in floating point libraries.

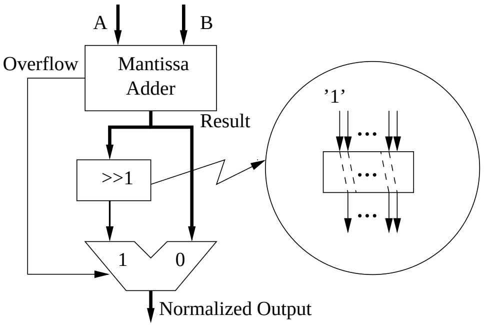


Figure 5. Normalization of Unsigned Adder

3.5.1 Sign Mode

The IEEE754 standard requires that floating point numbers be in sign-magnitude format. However, if operations can be limited to unsigned numbers, floating point adder implementation can be made smaller and faster. The result of unsigned addition is always positive and larger than either input number. Functionally, a normalizing post shifter needs only right shifters. To normalize the result, at most 1-bit right shifting is required. A multiplexer is used to replace the LOP and normalizer in the unsigned adder. The architecture after the mantissa addition is shown in Figure 5. The overflow bit of the mantissa is used to control the multiplexer. If the mantissa overflows, the result is right shifted by 1-bit and a *1* is shifted into the most significant bit.

3.5.2 Normalizer

In standard floating point units, output data must be in normalized format. This requirement maintains floating point number precision. Some operators require normalized input data to work properly. On FPGAs, output precision depends on operator bitwidth. When all operator output data bits are maintained, precision-based normalizing is unnecessary. In this case, the normalizer can be skipped to speed up circuit operation and to save resources. For LOP and 2-path adders, the LOD/LOP can also be eliminated.

3.5.3 Rounding

Five options are provided by our generator for rounding: (1) IEEE754 default, (2) biased round-to-near, (3) random rounding, (4) global random rounding, and (5) truncation. The IEEE754 default rounding scheme [9] rounds up remainders that are greater than or equal than 0.5. The rounding unit consists of an adder and a wide OR gate. The biased round-to-near approach rounds up when the remainder is greater than 0.5 and eliminates the wide OR. Random rounding algorithms provide random up/down rounding to increase the numerical stability of some algorithms [15, 19]. A random bit generator is required for this implementation. In global random rounding, a global random bit generator provides the random bit for all FPGA rounding operators. The truncation scheme is the simplest approach since it discards the remainder.

4 Intelligent Floating Point Library

Given a full spectrum of implementation trade-offs, it can be difficult for users to manually pick the best parameters for a specific floating point unit with defined operating characteristics. A floating point unit library and generation flow was developed to automate the parameter selection. This flow has been integrated into ASC [15], a C++-based stream compiler tool developed at Bell Laboratories.

4.1 ASC: A Stream Compiler for FPGAs

ASC requires a series of steps to convert C++ code to an FPGA bitstream. Initially, the ASC programmer selects a piece of the original program and transforms it to ASC code. In performing this transformation the user can trade-off silicon area for latency and throughput to explore the implementation space. ASC semantics are implemented as a C++ class library consisting of user defined types and operators for custom types. Custom hardware type operators are mapped to the PAM-Blox II module generation environment [13]. PAM-Blox II uses Compaq PamDC [14], a gate level design library, to generate hardware net-lists.

ASC uses types and attributes to hook the programmers algorithm description to the architectural features of the stream architecture data path. These custom types allow the application programmer to specify number representation size and form for each program variable. Each variable has a set of attributes, such as an architectural attribute and a sign attribute, for specifying negative number representations. Number representation types are custom types implemented in C++. The architecture attribute and the sign attribute are parameters stored in the hardware variable class state. In the case of floating point variables, the type is `HWfloat`, and the attributes are the bitwidth of the mantissa, the bitwidth of the exponent, etc.

For example, the following ASC code creates a stream of floating point numbers (`a`) as input and produces an incremented stream of output numbers (`b`). Each number is incremental *SIZE* times:

ASC code:

```
STREAM_START;
  // variables and bitwidths
  HWfloat a(IN, 24, 8);
  HWfloat b(OUT, 24, 8);
  for (i=0; i < SIZE; i++)
     b = a + 1.0;
STREAM_END;
```

This program is compiled with a conventional C++ compiler and generates an FPGA net-list. Note that the floating point format in the example has a 24 bit mantissa and an 8 bit exponent.

4.2 Floating Point Unit Selection

For each C++ expression in ASC code, users can choose `AREA`, `THROUGHPUT`, or `LATENCY` optimization options. By selecting one of these three modes, the generator selects the appropriate algorithm and architecture for each floating point operator and generates the appropriately-sized unit. Given area limitations, the tool provides the proper implementation choice based on a pre-defined cost function.

Parameter `galg` selects the floating point algorithm from IEEE standard, 2-path, or LOP floating point addition algorithms. The 2-path algorithm gives the smallest latency.

From experimentation, it was determined that area usage can be estimated with a linear function:

$$F(x) = a \cdot x + b \tag{1}$$

where $F(x)$ gives the number of LUTs and x is the mantissa bitwidth. The exponent is 8-bit by default. The linear equation of the curve indicates the linear growth of subcomponents such as ripple adders, comparators, and shifters.

In addition to area constraints, five additional parameters are used to control the design of floating operators in this library.

```
pipelining: gpipe={PIPE,ALIGN,NORM,NONE};
tbuf:       gsh  ={TBUF,BARREL};
normalize:  gnorm={YES,NO};
rounding:   ROUNDING_CHOICE;
```

Parameter `gpipe` specifies if pipelining occurs at all stages, at the alignment stage, at the normalization stage, or not at all.

Parameter `tbuf` specifies the low level implementation of the normalizer, using either Xilinx tri-state buffers (TBUF) or using a LUT-based barrel shifter.

Parameter `gnorm` indicates the use of selective normalization. (normalization after every operation).

Parameter `ROUNDING_CHOICE` indicates one of the six rounding modes: (1) none, (2) truncation, (3) IEEE, (4) nearest, (5) random, (6) global-random. The default is *truncation*.

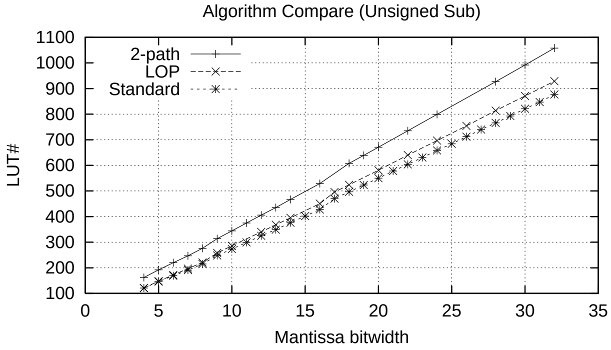


Figure 6. Area of Different FPGA Addition Algorithms

The other three parameters are automatically determined based on the required throughput, latency and area. Full pipelining is used if high throughput is desired. TBUFs are used to reduce area. Normalization, if required, is inserted. For advanced users, some parameters can be set manually to better assist the trade-off options.

5 Results

To evaluate the performance of our floating point unit library and generator, ASC was used to evaluate a spectrum of floating point unit designs. Resource usage and performance numbers were obtained using the Xilinx ISE4.1 [24] placement and routing tool set. All results are for the Xilinx XCV300E-6. Unless otherwise noted, all exponents follow the 8-bit IEEE754 standard.

5.1 Floating Point Algorithms

Fig. 6 and Fig. 7 present the area and unpipelined latency of the three algorithms across a range of mantissa bit widths. As expected, the 2-path algorithm has the smallest latency and uses the most programmable logic resources. With a 24-bit mantissa and 8-bit exponent, the 2-path floating point adder is 12% faster and 28% larger than the standard floating point adder. The size and performance of the LOP algorithm falls between the two algorithms.

As previous discussed in Section 2, the size of the the mantissa adder, exponent comparator, and LOD/LOP, in FPGA floating point adders grow linearly with bit width. In contrast, shifter size is O($mlog(m)$), where m is the mantissa bit width. This is approximately linear for small m.

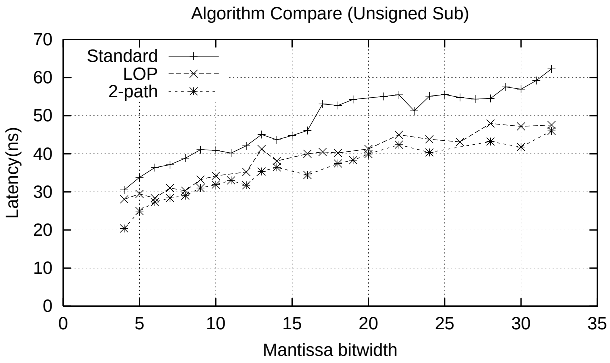


Figure 7. Latency of Different FPGA Addition Algorithms

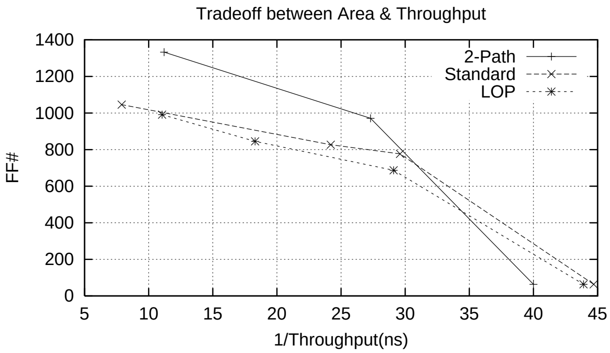


Figure 8. Trade-off Between Throughput and Area

Fig. 8 presents the trade-off between throughput and flip flop counts for the three algorithms. The floating point units support a 32-bit mantissa with an 8-bit exponent. In Fig. 8, the points at the right end of the lines are the modules optimized for latency. These modules have no internal flip flops. Fully block-level pipelined units correspond the points at the left. The intermediate points represents the modules obtained by selecting the partial pipelining options of `gpipe`. The 2-path tradeoff curve drops off more quickly since there are two data paths which require additional control circuitry.

5.2 Implementation of Sign Mode

Fig. 9 and Fig. 10 present the area and latency comparison of standard signed and unsigned floating point adders.

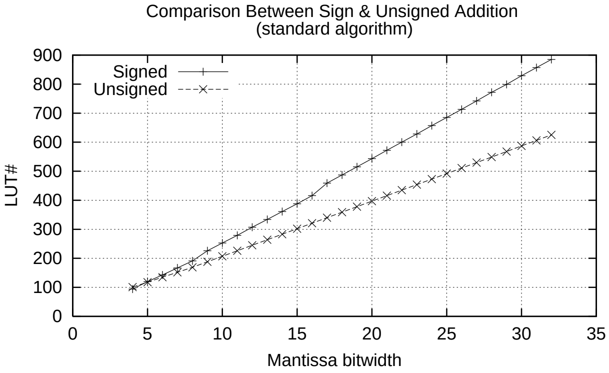


Figure 9. Sign Vs. Unsigned

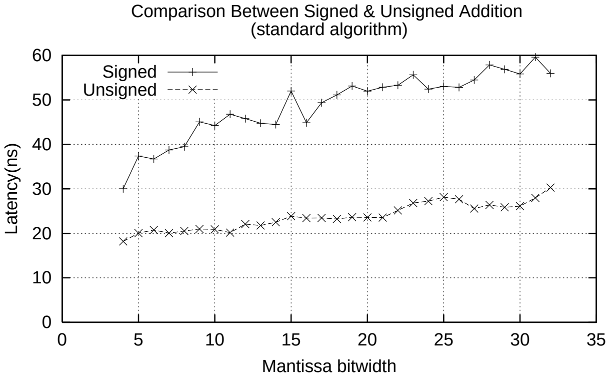


Figure 10. Sign Vs. Unsigned

The numbers in Fig. 9 and Fig. 10 indicate that the unsigned adder is about 39% smaller and 94% faster than the sign-magnitude adder with a 24-bit mantissa. These improvements are a result of the elimination of the normalizer for unsigned operation. The slope is a result of the linear growth of the mantissa adder and pre-shifter.

5.3 Normalizer

Fig. 11 and Fig. 12 show the area and latency of the sign-magnitude standard floating point adders with and without the normalizer. For designs with a 24-bit mantissa and 8-bit exponent, the adder which excluded the normalizer and associated multiplexer is 42% smaller and 76% faster.

5.4 Rounding

The affect of rounding on LUT area and design performance is shown in Fig.13. The results were calculated for

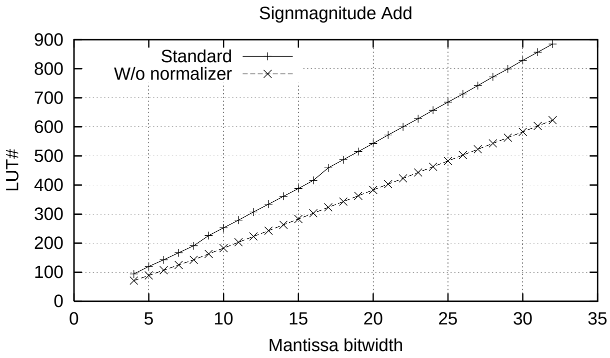


Figure 11. Normalized Vs. Un-normalized Area

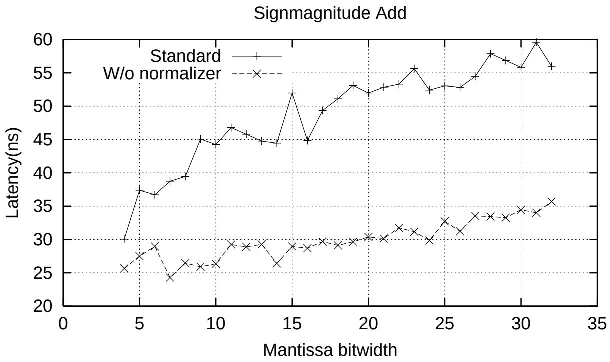


Figure 12. Normalized Vs. Un-normalized Latency

a floating point ader with a 32-bit floating point multiplier (4-bit exponent). The resulting 24-bit mantissa rounds to a 12-bit mantissa after multiplication.

5.5 Comparative Result with Commercial Floating Point Units

To evaluate the performance of our generator, the created modules are compared with modules taken from Xilinx [4] and Nallatech [16] FPGA floating point libraries. A IEEE754 standard compatible module with fixed frequency, bit-width, and pipeline stages was chosen.

For comparison, IEEE standard two 24-bit mantissa and 8-bit exponent modules were generated using our flow. Module *A* is optimized for latency and module *B* is optimized for throughput. Both modules use the same bit width and sign mode as the commercial libraries. All the parameters are obtained for the Xilinx XCV300E-6.

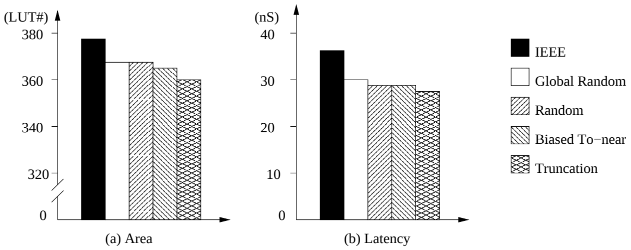


Figure 13. Trade-off of Rounding Schemes

It can be seen from Fig.14 that both *A* and *B* consume fewer resources compared with the commercial modules. Module *A* has half the latency of the Xilinx core, which is the fastest of the two commercial cores examined. Module *B* has a higher throughput than the Xilinx module, but is slower than the Nallatech module.

6 Performance of a Wavelet Application

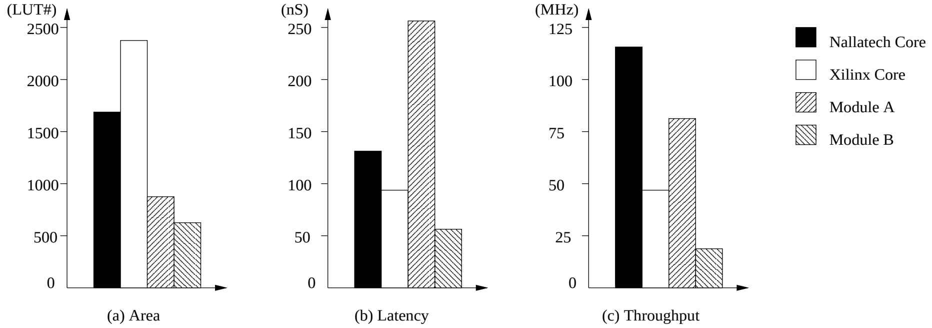
To show the utility of our generation system, floating point units generated with our system were integrated into the design flow of a wavelet filter. As shown in Eq. 2, for this application an input sequence, *x*, is convolved with nine coefficients. This application consists of 9 floating point multipliers and 8 floating point adders.

$$y[i] = \sum_{j=1}^{9} \alpha[j] * x[j+i] \tag{2}$$

ASC code was created for this application. All floating point units were based on IEEE754 standard floating point format and were generated automatically using our flow. The application allows for pipelined operation by sequentializing operations. Application latency is the latency sum of 9 floating point multipliers and 8 adders. The module generator automatically picked the parameters for the units. Performance numbers are shown in Table 1.

7 Conclusion

This paper presents a floating point unit generator for FPGAs, which, based on the requirements on throughput, area and latency, is able to create a range of floating point units. Our approach implements three floating point algorithms, the 2-path, LOP and the IEEE standard algorithms. Through experimentation, it has been shown that the area usage is roughly linear in bit width. By choosing different



(a) Area

(b) Latency

(c) Throughput

Figure 14. Comparison with Commercial Cores

Optimization Choice	Clock Period(ns)	Cycle Latency	Slices	LUTs	Flip Flop
Throughput	31.7	156	9638	16,125	12,154

Table 1. Performance of Wavelet Filter Application

algorithms, it is possible to trade off latency and area. Furthermore, customized sign modes, normalizing and rounding schemes can be selected to further optimize the area and performance. Our flow has been integrated into the ASC design environment. The library module generators are built to automate algorithm and architecture selection. Given area limitations, the module generator is able to select the unit with the appropriate precision.

We plan to continue this work by applying it to additional applications. Another future step includes the integration of on-chip block memories into the design flow.

References

[1] E. Antelo, M. Bóo, J. Bruguera, and E. Zapata. Design of a novel circuit for two operand normalization. *IEEE Transactions on VLSI Systems*, 6(1):173–176, 1998.

[2] P. Belanović and M. Leeser. A library of parameterized floating point modules and their use. In *the International Conference on Field Programmable Logic and Application*, Lecture Notes in Computer Science. Springer, 2002.

[3] J. D. Bruguera and T. Lang. Leading–one prediction with concurrent position correction. *IEEE Transactions on Computers*, 48(10):1083–1097, 1999.

[4] Digital Core Design. Alliance Core: DFPADD Floating Point Adder. In *http://www.dcd.pl*, 2001.

[5] B. Fagin and C. Renard. Field programmable gate arrays and floating point arithmetic. *IEEE Transactions on VLSI Systems*, 2(3):365–367, Sept. 1994.

[6] M. Farmwald. *On the Design of High Performance Digital Arithmetic Units*. PhD thesis, Stanford University, Aug. 1981.

[7] A. A. Gaffar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang. Automating customisation of floating-point designs. In *Field-Programmable Logic and Applications*, springer, 2002.

[8] E. Hokenek and R. Montoye. Leading-Zero Anticipator (LZA) in the IBM RISC System/6000 Floating-Point Execution Unit. *IBM Journal Research and Development*, 34(1):71–77, 1990.

[9] IEEE. IEEE Std 754-IEEE Standard for Binary Floating-Point Arithmetic. In *http://standars.ieee.org/reading/ieee/std/busarch/754-1984.pdf*, 1984.

[10] G. Lienhart, A. Kugel, and R. Mãnner. Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations. In *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2002.

[11] W. B. Ligon, S. McMillan, G. Monn, F. Stivers, and K. D. Underwood. A Re-evaluation of the Practicality of Floating-Point Operations on FPGAs. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa, CA, Apr. 1998.

[12] L. Louca, W. H. Johnson, and T. A. Cook. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 107–116, Napa, CA, Apr. 1996.

[13] O. Mencer. PAM-Blox II: Design and Evaluation of C++ Module Generation for Computing with FPGAs. In *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, Napa, CA, Apr. 2002.

[14] O. Mencer, M. Morf, and M. J. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa, CA, Apr. 1998.

[15] O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. Object-oriented domain-specific compilers for programming fpgas. *IEEE Transactions on VLSI, special issue on Reconfigurable Computin*, Feb. 2001.

[16] Nallatech. IEEE754 Floating Point Core. In *http://www.nallatech.com*, 2001.

[17] S. Oberman, H. Al-Twaijry, and M. Flynn. The SNAP Project: Design of Floating Point Arithmetic Units. In *Proceedings of Arith-13*, Pacific Grove, July 1997.

[18] V. G. Oklobdzija. An algoritmic and novel design of a leading zero detector circuit: Comparison with logic synthesis. *IEEE Transactions on VLSI Systems*, 2(1):124–128, 1993.

[19] D. S. Parker, B. Pierce, and P. R. Eggert. Monte carlo arithmetic: How to gamble with floating point and win. *Computing in science and engineering*, 2(4):58–68, July/Aug 2000.

[20] N. Quach and M. Flynn. Leading-one prediction, implementation, generalization and application. *Technical Report CSL-TR-91-463. Stanford University*, 1991.

[21] N. Shirazi, A. Walters, and P. Athanas. Quantitative Analysis of Floating Point Arithmetic on FPGA-based Custom Computing Machines. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 155–162, Napa, CA, Apr. 1995.

[22] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi. Leading-zero anticipatory logic for high speed floating point addition. *IEEE Journal of Solid-State Circuits*, 31(8):1157–1164, 1996.

[23] S. Waser and M. J. Flynn. *Introduction to Arithmetic for Digital Systems Designers*. Holt, Rinehard & Winston, New York, 1982.

[24] Xilinx. ISE Logic Design Tools. In *http://www. xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title= Design+Tools*, 2002.