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
{jliang,tessier}@ecs.umass.edu

Oskar Mencer
Department of Computing
Imperial College
London, England SW7 2AZ
oskar@doc.ic.ac.uk

Abstract

Most commercial and academic floating point libraries for FPGAs provide only a small fraction of all possible floating point units. In contrast, the floating point unit generation approach outlined in this paper allows for the creation of a vast collection of floating point units with differing throughput, latency, and area characteristics. Given performance requirements, our generation tool automatically chooses the proper implementation algorithm and architecture to create a compliant floating point unit. Our approach is fully integrated into standard C++ using **ASC***, a stream compiler for FPGAs, and the PAM-Blox II module generation environment. The floating point units created by our approach exhibit a factor of two latency improvement versus commercial FPGA floating point units, while consuming only half of the FPGA logic area.*

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 that allow for floating point unit trade-offs have been developed. Most commercial and academic floating point libraries provide only a small number of floating point modules with fixed parameters of bit-width, area, and speed. Due to these limitations, user designs must be modified to accommodate 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 characterized by throughput, latency and area:

1. FPGAs are often used in place of software to take advantage of 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 non-critical path units, it may be possible to trade off unit performance for reduced resource *area*.

Although bit-width variation provides some flexibility, this parameter alone cannot address all possible trade-offs.

The VLSI design community has developed a variety of floating point algorithms, architectures, and pipelining approaches. For example, a *2-path* floating point adder [6] was introduced to trade area for latency and other architectures [1, 3, 19] 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 create a large space of floating point adders, subtractors, multipliers and dividers based on a variety of parameters. Three trade-off levels can be explored: the architectural level, the floating point algorithm level, and the floating point representation level. Each of these levels requires examination of FPGA-specific features:

1. Floating point units can be built using various combinations of carry chains, LUTs, tri-state buffers and flip flops to obtain different throughput, latency, and area 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 [18], 2-path addition [6], Leading-One-Detection (LOD) [19], and Leading-One-Prediction (LOP) [3].
3. Floating point representations can be customized to use different sign modes for the mantissa and the exponent. Each generated floating point unit can support custom bit-width operands.

To ease design implementation, our floating point unit generation tool has been integrated into ASC [16], an easy-to-use and fully automatic design tool. This stream compiler allows for design specification using C++. The accuracy of our approach has been verified through the design and implementation of a wavelet filter which uses floating point arithmetic.

The remainder of this paper is structured as follows. Section 2 provides background on floating point unit design. Section 3 describes trade-offs in implementing floating point computation on FPGAs. The design flow and the algorithms used by our approach are introduced in Section 4. Area and performance results 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 and describe opportunities for future work.

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 fixed point is located after the first bit, m_0, so that $mantissa = \{m_0.m_1m_2...m_n\}$, where 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. A common value for the bias is -2^{k-1}, where k is the bit-width of the exponent [10].

The base sets the granularity of shifting and rounding. In the IEEE754 standard [9], the base is set to two. Other units use a larger number to reduce the latency of the shift operation, a critical part of most floating point arithmetic units. For example, the base for the IBM S/370 [23] is 16.

The IEEE floating point standard makes floating point unit implementation portable and the precision of the results predictable. A variety of different circuit structures can be applied to the same number representations, offering flexibility. The floating point unit algorithm, architecture, and bit-width adaptation offer significant potential for optimization.

2.2. Floating Point Implementations in FPGAs

Several efforts to build floating point units using FPGAs have been made. These approaches have generally explored bit-width variation as a means to control precision. A floating point library containing units with parameterized bit-width was described in [2]. In this library, mantissa and exponent bit-width can be customized. The library includes a unit that can convert between fixed point and floating point numbers. In Lienhart et al. [11], a floating point library with variable bit-width units is presented. The floating point units are arranged in fixed pipeline stages.

Several researchers [5, 12, 13] have implemented FPGA floating point adders and multipliers that meet IEEE754 floating point standards. Most commercial floating point libraries provide units that comply with the IEEE754 standard [4, 17]. 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 a corresponding floating point design. In contrast to earlier approaches, our floating point unit generation tool automates floating point unit creation.

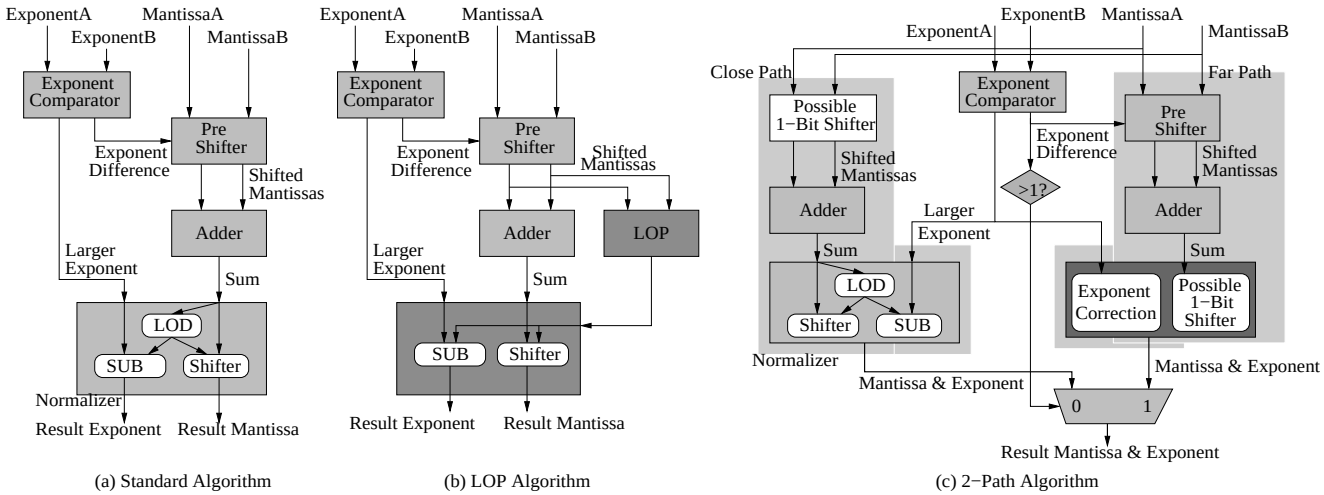
2.3. Floating Point Algorithms

Our floating point unit generation tool can create floating point multipliers, dividers, adders, and subtractors. Although floating point multipliers and dividers are costly in terms of logic resources, their implementation is relatively straightforward. Both units require a fixed point multiplier/divider for the mantissa and a fixed point adder/subtractor for the exponent. The algorithms for addition and subtraction require more complex operations due to the need for operator alignment. Three floating point add/subtract algorithms are briefly introduced in this section: standard [18], leading-one predictor (LOP) [3], and 2-path [6]. The implementation of these steps defines floating point unit throughput, latency, and area. To illustrate comparisons, we consider the block diagrams of the floating point adders shown in Figure 1.

Standard floating point addition requires five steps [18]:

1. Exponent difference
2. Pre-shift for mantissa alignment
3. Mantissa addition/subtraction
4. Post-shift for result normalization
5. Rounding

The area-efficient standard floating point adder is shown in Figure 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 mantissa corresponding to the smaller exponent by the difference of the exponents so that the resulting two mantissas are aligned and can be added. If the mantissa adder generates a carry output (e.g. when both mantissas have *one*s as most significant bits), the resulting mantissa is shifted one bit to the right and the exponent is increased



(a) Standard Algorithm (b) LOP Algorithm (c) 2−Path Algorithm

Figure 1. Floating Point Addition Algorithms

by one. 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, the resulting mantissa is left-shifted by an amount subsequently deducted from the exponent.

For the standard algorithm, the exponent comparator is implemented with a subtractor and a multiplexer. The comparator requires about $2 \times n$ LUTs, where n is the exponent bit-width. The size of the pre-shifter is about $m \times log(m)$ LUTs, where m is the bit-width of the mantissa. The size of the mantissa adder depends on the adder architecture and sign mode. If a ripple-carry adder is used for an unsigned mantissa, about m LUTs are required. The normalizer LOD is nearly the same size as the mantissa adder. The shifter is equal in size to the pre-shifter and the subtractor (SUB) is about the same size as the exponent comparator. Overall, the size of the normalizer is about the sum of the sizes of the other three components.

Figure 1(b) shows a block diagram of a Leading-One-Predictor (LOP) floating point adder [8, 21, 22]. This adder implementation requires more area than a standard adder, but exhibits reduced latency. The primary difference between the adders is the replacement of the leading-one detector (LOD) circuit with a leading-one predictor (LOP) circuit. Since the LOP circuit can be executed in parallel with mantissa addition, overall latency can be reduced.

The 2-path adder [6], shown in Figure 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 values are larger than 1, the *far* path, on the right in Figure 1(c), is taken. Otherwise, the *close* path on the left is taken. After alignment, one of the mantissas is reduced and shifted by at most one bit. This close path implementation eliminates the pre-shifter.

3. FPGA Floating Point Unit Trade-offs

In this section, floating point unit implementation trade-offs 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 Figure 1(a) contains a subtractor. The normalizer requires a second shifter to convert the resulting number into normalized format. The mantissa adder forms the kernel of this unit. A ripple-carry adder using an FPGA carry chain is an efficient mantissa adder implementation. Alternately, a serial adder can be used to minimize area.

3.2. LOP Floating Point Adder

The LOP algorithm requires the use of a leading-one predictor. Figure 2 shows the implementation of a LOP unit using Virtex CLBs. A_i and A_{i-1} are consecutive bits of the minuend, and 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. The LOP

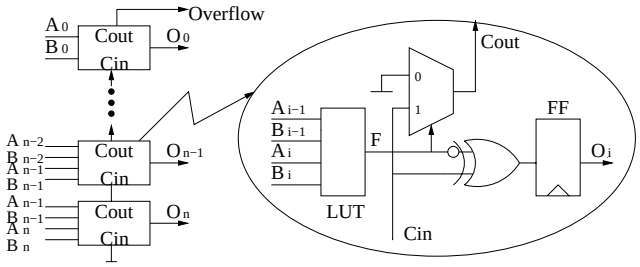


Figure 2. LOP Circuitry Implemented in Virtex CLBs

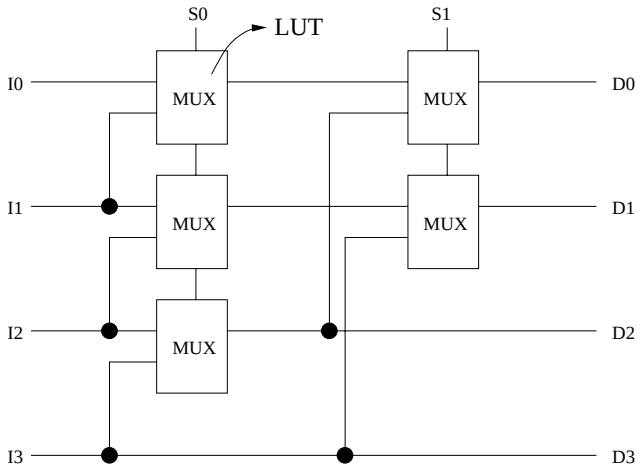


Figure 3. A LUT-Based Shifter

has nearly the same delay as the mantissa adder if an embedded FPGA carry chain is used. The LOP may lead to a one-bit error in the position of the leading one. This error is detected in the normalizer and the leading one is shifted one position to the left during normalization.

3.3. 2-Path Floating Point Adder

As mentioned in Section 2, 2-path adder implementations shorten the critical path of floating point addition by eliminating shift logic. Figure 3 illustrates that this critical path savings can be substantial. The figure shows a 4-bit shift implementation with input bits, $I_{0,1,2,3}$, output bits, $D_{0,1,2,3}$, and shift index, $S_{0,1}$. 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.

3.4. Pipelining

The pipelining of floating point adders can be adjusted to realize area and throughput trade-offs. The removal of registers often results in a shorter pipeline with lower

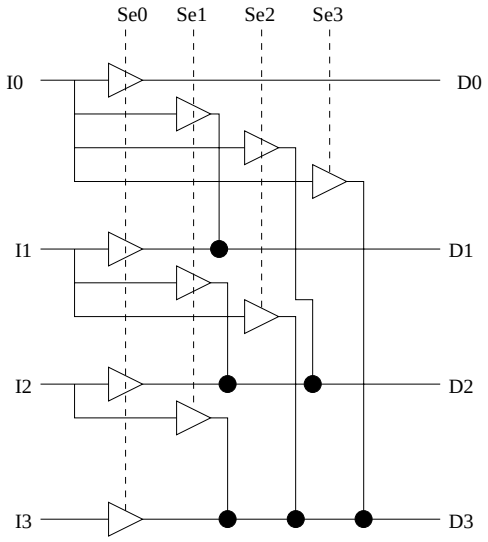


Figure 4. A Tri-state Buffer Shifter

throughput. This reduction generally leads to simpler overall control circuitry. All adders generated by our system can be pipelined at the block level. The exponent comparator requires one pipeline stage, the pre-shifter requires $log(bitwidth)$ stages, the mantissa add/subtract unit requires one stage, the LOP/LOD requires one stage, and the normalizer requires $log(bitwidth)$ stages. The $log(bitwidth)$ stages of the shifter shown in Figure 3 require a pipeline register after every multiplexer. In our generator, the number of pipeline stages per unit can be tuned through the use of input parameters.

3.5. Tri-state Buffer Usage

Tri-state buffers can be used instead of LUTs to efficiently build long shifters. As shown in Figure 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 LUT-based shifter. The availability of tri-state buffers is generally limited in contemporary FPGAs.

3.6. Parameter Specialization

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

3.6.1. Sign Mode

The IEEE754 standard requires that floating point numbers be represented in signed-magnitude format. How-

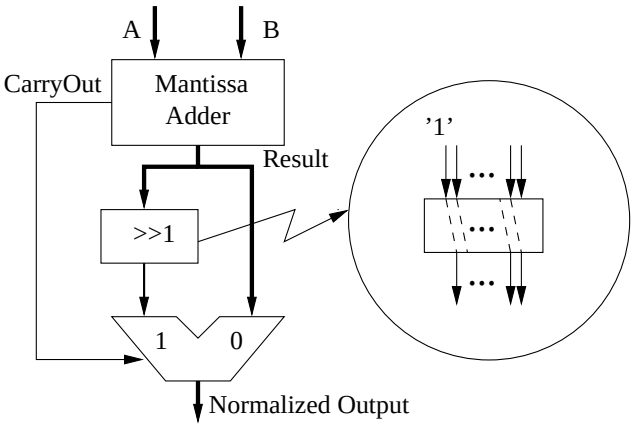


Figure 5. Normalization of Unsigned Addition

ever, if addition can be restricted to unsigned values, floating point adder implementations can be made smaller and faster through specialization. The result of unsigned addition is always positive and larger than either input number. Functionally, a normalizing post shifter requires only right shifters. To normalize the result, at most 1-bit right shifting is required. A multiplexer can be used to replace the LOP and the normalizer in the unsigned adder. The optimized architecture is shown in Figure 5. The $CarryOut$ bit of the mantissa is used to control the multiplexer. If the mantissa has a carry output, the mantissa addition result is right shifted by 1-bit and a *1* is shifted into the most significant bit.

3.6.2. Normalization

Output data must be represented in normalized format in standard floating point units. This requirement maintains the precision of floating point numbers. Some operators require normalized input data to work properly. In FPGAs, output precision depends on operator bit-width. When all operator output data bits are maintained, normalization 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.

3.6.3. Rounding

Five rounding options are provided by our generator: (1) IEEE default, (2) biased round-to-nearest, (3) random, (4) global random, and (5) truncation. The IEEE754 default rounding scheme [9] rounds up remainders that are greater than or equal to 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 techniques provide random up/down rounding to increase the numerical stability of some addition algorithms [16, 20]. A random bit generator is required for this approach. In global random rounding, a global random bit generator provides the random bit for all FPGA rounding operators. A truncation scheme is the simplest rounding approach since it discards the remainder eliminated by rounding.

3.7. Bit-Width Variation

Each floating point unit can have a variable mantissa and exponent bit-width. It was found that adders with odd bit-widths are slower than adders with even bit-widths. This effect occurs because Virtex CLBs consist of an even number of LUTs. If the input bit-width is odd, the last bit will not be placed in the carry chain, which results in a longer delay.

3.8. Exception Handling

Floating point units created by our generation tool indicate overflow via a dedicated output signal. This signal is propagated as an input to successive units in the floating point unit chain. Currently, no hardware support for not-a-number (NaN) or underflow detection is provided.

4. Floating Point Unit Generation

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 generation flow was developed to automate parameter selection. This flow has been integrated into ASC [16], 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 into an FPGA bitstream. Initially, the ASC programmer selects a piece of the original program and transforms it into 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 [14]. PAM-Blox II uses Compaq PamDC [15], a gate level design library, to generate hardware netlists.

ASC uses types and attributes to hook the programmer's algorithm description to the architectural features of the stream architecture data path. These custom types allow the

application programmer to specify both the number representation size and the type for each program variable. Each variable has a set of attributes, such as an architectural attribute and a sign attribute, to specify 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 include the bit-width of the mantissa and the bit-width of the exponent, among others.

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 incremented $SIZE$ times:

```
STREAM_START;
  // variables and bit-widths
  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 netlist. 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 the IEEE standard, LOP, and 2-path floating point addition algorithms.

From experimentation, it was determined that area usage for each of the three types of adders can be estimated using a linear function:

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

where $F(x)$ gives the number of LUTs, x is the mantissa bit-width, and a and b are scaling constants. The exponent can be set to a default value. The linear equation of the curve indicates the linear growth of subcomponents such as ripple-carry adders, comparators, and shifters. In addition to area constraints, four additional parameters are used to control the design of floating point operators in this library.

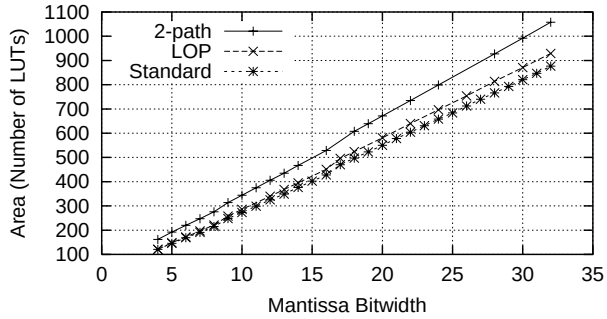


Figure 6. Area of FPGA Addition Algorithms

```
pipelining: gpipe={PIPE,ALIGN,NORM,NONE};
tbuf:       gsh  ={TBUF,LUT};
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 `gsh` specifies the low level implementation of the normalizer, using either Xilinx tri-state buffers (TBUF) or a LUT-based shifter. Parameter `gnorm` indicates the use of selective normalization (e.g. normalization after every operation). Parameter `ROUNDING_CHOICE` indicates one of the five rounding modes listed in Section 3.6.3. The default rounding mode is truncation.

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, parameters can be set manually to better assist the investigation of trade-offs.

5. Results

To evaluate the performance of our floating point unit generator, ASC was used to create 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 Virtex XCV300E-6. Unless otherwise noted, all exponents follow the 8-bit IEEE754 standard.

5.1. Floating Point Algorithms

Figures 6 and 7 present the area and unpipelined latency of the three addition algorithms across a range of mantissa bit-widths. As expected, the 2-path algorithm uses the most

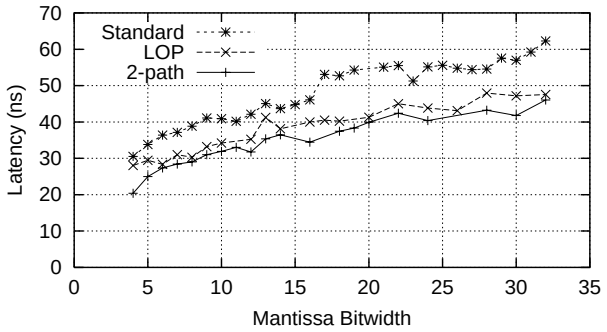


Figure 7. Latency of Addition Algorithms

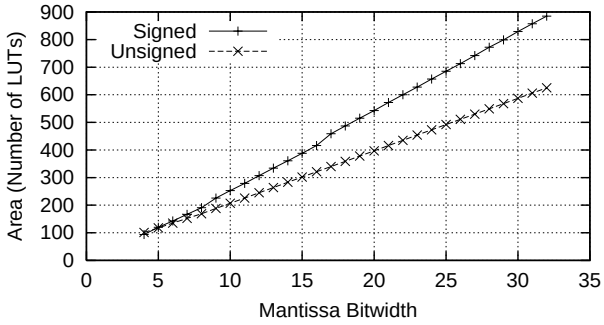


Figure 9. Signed vs. Unsigned Area

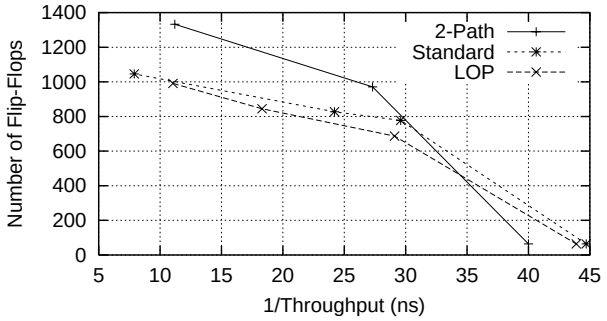


Figure 8. Throughput and Area Trade-off

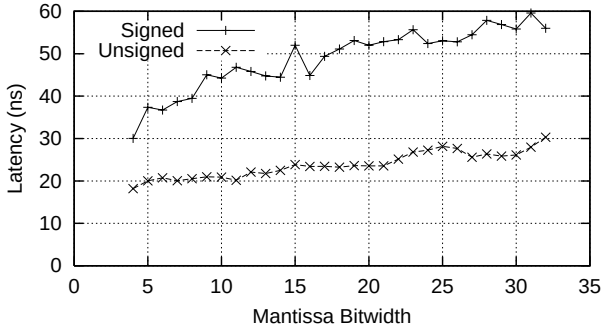


Figure 10. Signed vs. Unsigned Latency

programmable logic resources and has the smallest latency. With a 24-bit mantissa and 8-bit exponent, the 2-path floating point adder is 19% larger and 28% faster than the standard floating point adder. The size and performance of the LOP algorithm falls between the two algorithms.

As previously discussed in Section 4, the size of the mantissa adder, exponent comparator, and LOD/LOP in FPGA floating point adders grow linearly with bit-width. In contrast, shifter size complexity is O($m\ log(m)$), where m is the mantissa bit-width. This value is approximately linear for small m.

Figure 8 presents the trade-off between throughput and flip flop counts for the three algorithms. Floating point units with 32-bit mantissas and 8-bit exponents were used to generate these results. In Figure 8, the points at the right end of the plots are for modules optimized for latency. These modules have no internal flip flops. Fully block-level pipelined units correspond to the points at the left. The intermediate points represent the modules obtained by selecting the partial pipelining options of `gpipe`. The 2-path trade-off curve drops off more quickly than the others since there are two data paths which require additional control circuitry.

5.2. Sign Mode Implementation

Figures 9 and 10 present area and latency comparisons of standard signed and unsigned floating point adders. The numbers in these figures indicate that the unsigned adder with a 24-bit mantissa is about 26% smaller and 46% faster than the signed-magnitude adder. These improvements are a result of the elimination of the normalizer for unsigned operation. The slope of the curves is a result of the linear growth of the mantissa adder and pre-shifter.

5.3. Normalization

Figures 11 and 12 show the area and latency of signed-magnitude standard floating point adders with and without

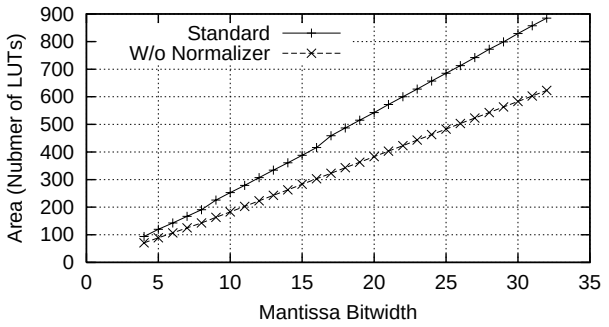


Figure 11. Normalized vs. Un-normalized Area

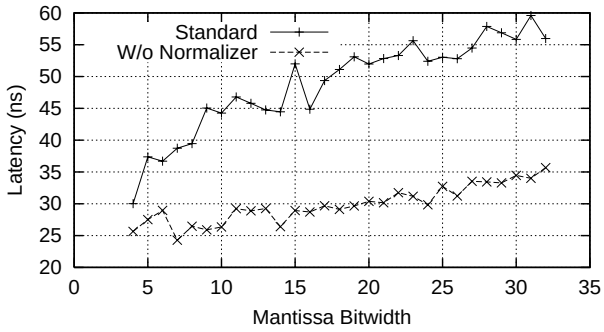


Figure 12. Normalized versus Un-normalized Latency

normalization. For designs with a 24-bit mantissa and 8-bit exponent, the un-normalized adder and associated multiplexer are 28% smaller and 44% faster than the corresponding normalized circuits.

5.4. Rounding

The affect of rounding on LUT area and design performance is shown in Figure 13. The results were calculated for floating point multipliers with 16-bit input values (12-bit mantissa, 4-bit exponent). The resulting 24-bit mantissa is rounded to a 12-bit mantissa.

5.5. Comparison with Commercial Floating Point Units

To evaluate the performance of our generator, the created modules were compared with modules taken from Xilinx [4] and Nallatech [17] FPGA floating point libraries. A IEEE754 standard compatible module with fixed frequency, bit-width, and pipeline stages was chosen.

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

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

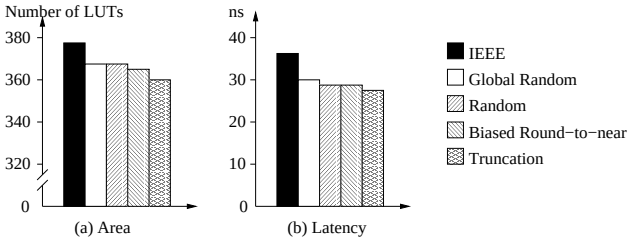


Figure 13. Area and Delay Comparison of Rounding Approaches

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 Equation 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 floating point adders. The module generator automatically selected the parameters for the units. Performance numbers for the application are shown in Table 1.

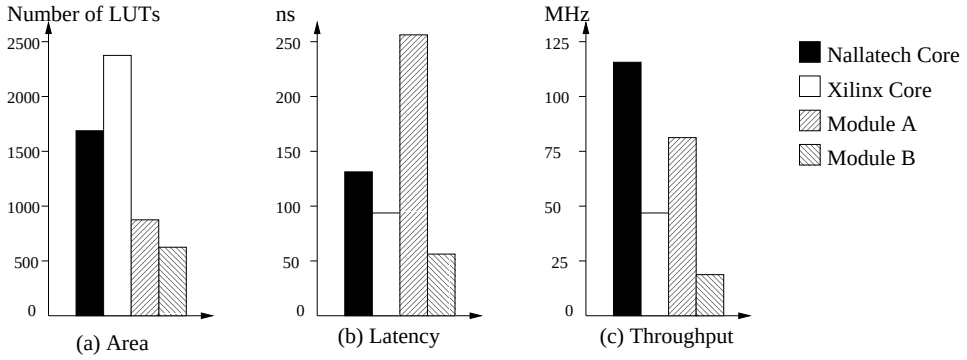


Figure 14. Comparison with Commercial Floating Point Addition Cores

Optimization Choice	Clock Period (ns)	Slices	LUTs	Flip Flops
Throughput	31.7	9638	16,125	12,154

Table 1. Performance of the Wavelet Filter Application

7. Conclusion

This paper presents a floating point unit generation tool for FPGAs, which, based on throughput, latency, and area requirements, is able to create a range of floating point units. Our approach implements three floating point addition algorithms, the IEEE standard, LOP, and 2-path algorithms. Through the selection of different algorithms, latency and throughput can be traded for area. Customized sign modes and normalization and rounding schemes can be selected to further optimize floating point unit area and performance. To ease design implementation, our flow is integrated into the ASC stream compilation environment. Experimental results for floating point addition show superior latency performance versus commercial core offerings.

We plan to continue this work by applying our generator to additional applications. Other future steps include the integration of on-chip block memories into the design flow and improved hardware support for underflow and NaN detection.

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 *Proceedings, International Conference on Field Programmable Logic and Applications*, Montpelier, France, Aug. 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, Inc. *Alliance Core Data Sheet: DFPADD Floating Point Adder*, 2001. http://www.dcd.pl/dcdpdf/xil/dfpadd_ds.pdf.

[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, Department of Electrical Engineering, Aug. 1981.

[7] A. A. Gaffar, W. Luk, P. Y. Cheung, N. Shirazi, and J. Hwang. Automating Customisation of Floating-point Designs. In *Proceedings, International Conference on Field-Programmable Logic and Applications*, Montpelier, France, Aug. 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] Institute of Electrical and Electronics Engineers. *IEEE 754 Standard for Binary Floating-Point Arithmetic*, 1984. http://standards.ieee.org/reading/ieee/std/busarch/754-1984.pdf.

[10] I. Koren. *Computer Arithmetic Algorithms*. Brookside Court Publishers, Amherst, MA, 1998.

[11] G. Lienhart, A. Kugel, and R. Mǎnner. Using Floating-Point Arithmetic on FPGAs to Accelerate Scientific N-Body Simulations. In *Proceedings, IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 182–191, Napa, CA, Apr. 2002.

[12] 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 *Proceedings, IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 206–215, Napa, CA, Apr. 1998.

[13] L. Louca, W. H. Johnson, and T. A. Cook. Implementation of IEEE Single Precision Floating Point Addition and Multiplication on FPGAs. In *Proceedings, IEEE Workshop on*

FPGAs for Custom Computing Machines, pages 107–116, Napa, CA, Apr. 1996.

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

[15] O. Mencer, M. Morf, and M. J. Flynn. PAM-Blox: High Performance FPGA Design for Adaptive Computing. In *Proceedings, IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 167–174, Napa, CA, Apr. 1998.

[16] O. Mencer, M. Platzner, M. Morf, and M. J. Flynn. Object-oriented domain-specific compilers for programming FPGAs. *IEEE Transactions on VLSI Systems*, 9(1):205–210, Feb. 2001.

[17] Nallatech, Inc. *IEEE754 Floating Point Core*, 2001. http://www.nallatech.com/products/ip/floating_point_virtex_1/index.asp.

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

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

[20] 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.

[21] N. Quach and M. Flynn. Leading-one prediction, implementation, generalization and application. In *Technical Report CSL-TR-91-463*, Department of Electrical Engineering and Computer Science, Stanford University, 1991.

[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, N.Y., 1982.

[24] Xilinx Corporation. *ISE Logic Design Tools*, 2002. http://www.xilinx.com/.