

# ITERATIVE FLOATING POINT COMPUTATION USING FPGA DSP BLOCKS

*Fredrik Brosser, Hui Yan Cheah, Suhaib A. Fahmy*

School of Computer Engineering  
Nanyang Technological University, Singapore  
frebro@student.chalmers.se, hycheah1@e.ntu.edu.sg

## ABSTRACT

This paper presents a single precision floating point unit design for multiplication and addition/subtraction using FPGA DSP blocks. The design is based around the DSP48E1 primitive found in Virtex-6 and all 7-series FPGAs from Xilinx. Since the DSP48E1 can be dynamically configured and used for many of the sub-operations involved in IEEE 754-2008 binary32 floating point multiplication and addition, we demonstrate an iterative combined operator that uses a single DSP block and minimal logic. Logic-only and fixed-configuration DSP block designs, and other state-of-the-art implementations, including the Xilinx CoreGen operators are compared to this approach. Since FPGA based systems typically run at a fraction of the maximum possible FPGA speed, and in some cases, floating point computations may not be required in every cycle, the iterative approach represents an efficient way to leverage DSP resources for what can otherwise be costly operations.

## 1. INTRODUCTION

DSP blocks in modern FPGAs have increased in capability over time. Concurrently, FPGAs have found use in a range of application domains that now demand floating point computation. However, floating point arithmetic operators consume a significant amount of resources. Some existing work has sought to use DSP blocks within such operators. Indeed, the Xilinx CoreGen operators do use DSP blocks for parts of the floating point arithmetic. DSP blocks are not just flexible at design time, rather, their operation can be reconfigured at runtime.

In this paper, we investigate the use of DSP blocks in an iterative manner for single precision floating point adders and multipliers. We show how the flexibility can be leveraged to create a combined adder and multiplier unit which is efficient, and offers reasonable performance. We compare the design to other floating point operators, both open-source and proprietary, such as the Xilinx LogiCore operators or those from the open source FloPoCo project [1], which can be optimised for latency or speed. Existing operator implementations do sometimes make use of DSP blocks,

but use a fixed DSP configuration. The flexibility and low logic usage of our approach comes at the cost of a higher latency and lower throughput, but this may be feasible for some applications.

For floating point numbers, addition is especially costly due to mantissa alignment and normalisation – two operations that each require shifting, as well as shift amount calculations. These operations can all be carried out in a DSP48E1. Existing IEEE 754 floating point operators in FPGAs are subject to some limitations. Alignment and normalisation are used by all floating point operators [2], and are costly operations to implement in logic. Furthermore, the width of embedded multipliers commonly found in current FPGAs are not well suited for the multiplications used in IEEE 754 operations. The proposed design reduces the logic consumption of floating point operations significantly, by trading area for latency.

The remainder of this paper is organised as follows. Section 2 gives a summary of related work, Section 3 introduces the floating point format used and gives a brief overview of alternatives. In Sections 4 and 5 we describe two conventional floating point operator designs for addition and multiplication. Section 6 describes the iterative DSP48E1-based floating point unit implementation. Section 7 presents implementation results. Finally, we conclude and discuss future work.

## 2. RELATED WORK

Floating point operator design is a well explored area in the ASIC community, and to a lesser extent, in the FPGA community. A notable example for FPGAs is the (open-source) FloPoCo project [1]. This toolset provides a versatile generator for floating point operators and datapaths, generating synthesisable VHDL code. However, the project is not strictly focused on using DSP slices, instead focusing on flexibility and specialised operators such as those described in [3]. Previous papers have examined tradeoffs involved in designing floating point operators for FPGA ([4], [5]). Xilinx CoreGen can generate floating point operators that make effective use of the DSP48E1 slice, and these are widely

used. In [6], the authors describe a Block Processing Element for block floating point numbers, adopting a similar approach using DSP slices.

More advanced DSP blocks like the DSP48E1 are highly flexible and can be reconfigured on a cycle-by-cycle basis at runtime [7]. However, in the vast majority of cases in the literature, this flexibility is not leveraged, and DSP blocks are inferred through the synthesis tool, that generates a static configuration. Relying on vendor tools to infer DSP blocks from general RTL code can be inefficient. [8]

[9] presents a soft processor built around a single DSP48E1 that uses this flexibility to support a full instruction set. The work presented in this paper builds on the same idea, of changing the DSP block configuration at run time, as applied to floating point operators.

### 3. FLOATING POINT FORMAT

To first ensure we have a handle on floating point operator implementation, and to better understand how the DSP block might be used, we implement fully-parallel versions of floating point addition and multiplication with and without DSP blocks. We show in Section 7 that these implementations are comparable to the state of the art.

The floating point format used for the operators presented in this paper is based on IEEE 754-2008 binary32, with some simplifications. First, only one type of NaN (quiet NaN) is used and only the default rounding mode (round to nearest, tie to even) is supported. Denormalised numbers are also not handled. Full support for IEEE 754 requires these to be supported [10], therefore these operators are compatible with, but not completely following the IEEE 754-2008 binary32 standard.

IEEE 754 Floating point numbers are represented as

$$F_{E,F} = (-1)^{-s} * 2^{(\omega_E - 127)} * (1 + \omega_F), \quad (1)$$

where  $E$  and  $F$  are the number of bits in the exponent and mantissa fields, respectively. A single precision floating point number uses 32 bits and its bit fields can be written

$$F_{E,F} = \{S, E_{[7:0]}, F_{[22:0]}\}, \quad (2)$$

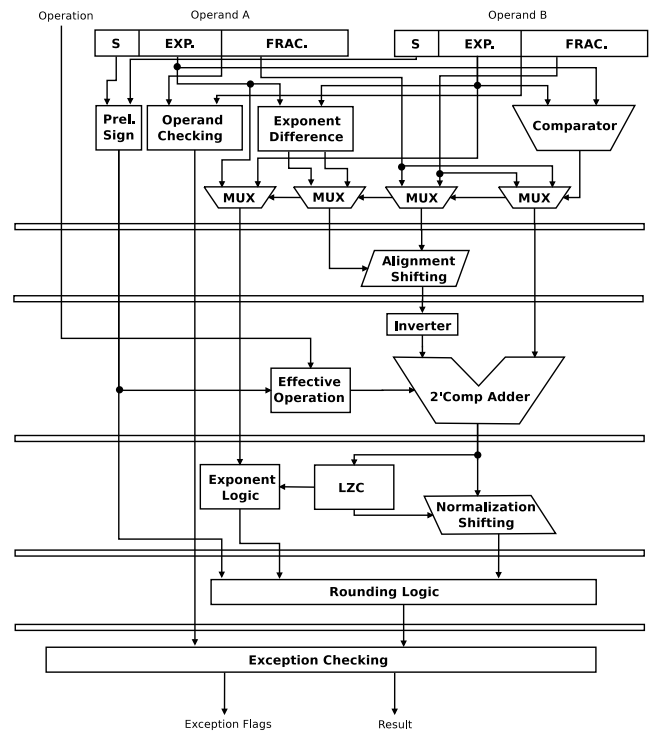
where  $S$  is a single sign bit,  $E_{[7:0]}$  is an 8-bit exponent field, and  $F_{[22:0]}$  is a 23-bit mantissa field. The 8-bit exponent uses an Excess-127 format to represent signed numbers. An implicit 1 is included as the MSB of the mantissa field, making the effective fraction part  $1.f_{[22:0]}$ .

All five exception flags defined in the IEEE 754 standard are tested for and set accordingly. These are *Invalid Operation*, *Division by Zero*, *Overflow*, *Underflow* and *Inexact*. In our proposed designs *Division by Zero* is unused, but still provided for compatibility with IEEE 754. Previous papers on the subject have described alternative approaches. [6] uses the block floating point format to avoid the added

hardware complexity introduced by full IEEE 754-2008 binary32 compatibility. We have decided to use IEEE 754-2008 binary32 because of the large range of representable numbers and compatibility with other existing hardware. We consider single-precision operators in this paper, with double-precision left to future work.

### 4. LOGIC-ONLY FIXED CONFIGURATION FLOATING POINT OPERATORS

In this section, we present LUT-only implementations of addition and multiplication. These operators are later used for comparison with the iterative design. The purpose is to explore the potential for mapping each suboperation to the DSP48E1 by introducing the logic-only alternative first.



**Fig. 1.** Fixed Configuration Floating Point Adder Architecture for Logic-Only implementation.

The logic-only adder design is based on a standard (naive) implementation of addition for IEEE 754. This standard algorithm consists of three main stages, namely alignment, mantissa addition and normalisation. In the proposed design, presented in Figure 1, these steps are further pipelined for performance. The individual pipeline stages are organised as follows.

1. **Pre-alignment and Exponent Logic.** In the first pipeline stage, the input operands are checked for exception conditions, and the operands are split into bit fields for computing

the preliminary sign and exponent difference. Exponent difference is computed using two 8-bit subtractions, which can effectively be done in a single DSP48E1 addition.

2. **Alignment Shift.** A barrel shifter is used to perform the shifting, using the Guard, Round and Sticky bits ( $G$ ,  $R$ ,  $S$ ). The 25x18 multiplier in the DSP48E1 slice is not wide enough for a single DSP slice to perform the alignment shifting, requiring multiple iterations or two cascaded DSP blocks instead[11].

3. **Addition.** The effective operation is computed from the preliminary sign and the input operation. If the effective operation is subtraction, the lesser operand is negated. This addition can be performed in a single DSP48E1.

4. **Normalisation.** The normalisation step is usually the most resource demanding step in a floating point addition, as it involves both shifting and leading-zero counting. The leading zero-counter in our design is simply implemented by a series of multiplexers and the shifter is implemented using a barrel shifter. This would require either two DSP blocks or multiple iterations.

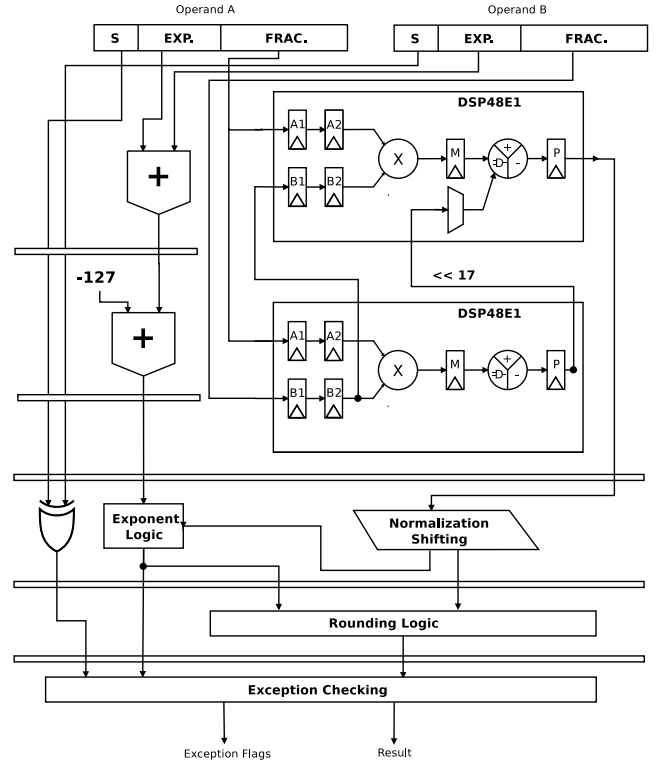
5. **Rounding.** The rounding mode used is the default IEEE 754 rounding mode, *round to nearest, tie to even*, computed using  $G$ ,  $R$ ,  $S$  and  $norm(F_{sum})$ . The final sign is calculated, taking into account special cases such as  $round(F_{sum}) = 0$ .

6. **Exception Condition Checking.** In the exception condition checking stage  $round(F_{[22:0]})$  and  $round(E_{[7:0]})$  are checked for overflow (which could arise from rounding), underflow, invalid result, division by zero and inexact result.

The logic-only multiplier implementation is algorithmically simpler than the adder and uses fewer pipeline stages. The Alignment Shifting stage is not needed for multiplication, but a LUT-only implementation of the required 25x25 bit multiplication is expensive in terms of logic.

## 5. FIXED CONFIGURATION FLOATING POINT OPERATORS WITH DSP BLOCKS

In this section, we present implementations of addition and multiplication using DSP blocks in fixed configurations. The adder design uses the DSP48E1 slice to perform the mantissa addition, with all other pipeline stages unchanged. The multiplier design involves more modifications that we explain to highlight the use of fixed-configuration vectors with the DSP48E1. The floating point multiplier makes use of two DSP48E1 blocks for the mantissa multiplication, making the circuit considerably smaller than the logic only implementation. The DSP48E1 blocks are configured as shown in Figure 2, where two DSP48E1 blocks are cascaded together to accommodate the required 24x24-bit mantissa multiplication. This is needed because a single DSP block supports only a 25x18-bit multiplication [12]. Here, the run



**Fig. 2.** Fixed Configuration Floating Point Multiplier Architecture Using DSP Blocks.

time dynamic configurability of the DSP block is not leveraged, instead using fixed configuration vectors 0000101 and 1010101 as OPMODE for DSP blocks 1 and 2, respectively. This gives the following effective outputs from the DSP blocks.

$$P_1 = A_{[23:0]} * B_{[17:0]} \quad (3)$$

$$P_2 = (A_{[23:0]} * B_{[23:18]}) + (P_1 \ll 17) \quad (4)$$

The common exponent is calculated by adding the exponents of the operands and subtracting the bias (-127). Normalisation of a floating point product is simpler than for a floating point sum, involving only a potential 1-bit right normalisation shift required when overflow occurs. The sign calculation is the XOR of the operands' signs. Using Guard, Round and Sticky bits, the result is rounded using the default mode (round to nearest, tie to even). Similarly to the adder, the correctly rounded result is checked for exception conditions. The exception flags are output as a bit vector.

## 6. ITERATIVE DSP-BASED FLOATING POINT UNIT

In this section we present an iterative design for a DSP48E1-based floating point unit. The design takes as input two IEEE 754 single-precision inputs. As opposed to the logic-only and fixed DSP-configuration designs presented, the iterative design is capable of both addition/subtraction and

multiplication selected via a control input. The dynamic programmability of the DSP48E1 block allows the datapath to be configured at runtime to perform the required operation.

The DSP48E1 is used for as many of the sub-computations as possible in order to conserve logic resources. Figure 3 gives a block diagram description of the design. Algorithmically the iterative operator is equivalent to the logic-only and fixed-configuration DSP implementations. Since the design is iterative and uses a single DSP block, the initiation interval will be longer than fixed-datapath operators. Currently, the initiation interval is 20 and 23 cycles for multiplication and addition/subtraction, respectively. Pre-alignment and Exception stages can be overlapped between instructions, as they do not make use of the DSP block. The high initiation interval results in an overall lower throughput for the iterative design, compared to the fixed datapath implementations such as the Xilinx CoreGEN operators.

The Control Unit is implemented as a state machine and coordinates inputs from the RAM32M with the control vectors to the DSP48E1. Other functions, such as pre-align logic (checking for input exceptions and splitting operands into bit fields) and exception checking for the final results, are implemented in logic.

Because floating point adders and multipliers share some common sub-operations, the iterative design gives greater flexibility by providing both operations in the same module. The DSP48E1 slice works as an execution unit, with the input configuration parameters ALUMODE, OPMODE and INMODE [12] set dynamically by the Control Unit. The DSP48E1 is fully pipelined for increased frequency, giving a latency of 3 clock cycles for each iteration round. An extra register for the C input is implemented in fabric logic outside the DSP block (See Figure 3). For storing intermediate results between iterations through the DSP48E1, 16 instances of (2-bit) RAM32M blocks are used, storing 32 bit values. The synthesis tool (Xilinx ISE) implements this as LUTs. The output from the DSP block is written back to the RAM32M Block. Below follows a step-by-step description of the operations.

1. **Pre-alignment.** In the first pipeline stage, the input operands are checked for exception conditions such as non-normalised numbers, and the operands are split into bit fields for computing the preliminary sign and exponent difference. Exponent difference is computed using two 8-bit subtractions, which can effectively be done in a single DSP48E1 addition. The exponent difference computation uses one iteration in the DSP (3 clock cycles).

2. **Align.** The alignment stage requires two iterations through the execution core. Shifting in the DSP48E1 is implemented as a multiplication, with the 18 bit multiplier input set as  $2^k$  (as described in [11]), where  $k$  is the shift amount calculated in the pre-alignment stage. The smaller mantissa is shifted

to fit the new, shared, exponent. Intermediate results and rounding information bits are stored in the memory block. The maximum shift amount is 25 (as the smaller mantissa would be shifted out to become all-zero).

3. **Execution.** Addition can be carried out in a single iteration, exactly as for the logic-only implementation of the adder. Multiplication, again with the limitation of the 25x18 bit multiplier in the DSP48E1, must be carried out iteratively over two iterations through the DSP slice. As an example of the dynamic configuration of the DSP block mentioned earlier, in the execution stage the OPMODE input from the control unit to the DSP block is set as 0000011 for addition and as 0000101 for multiplication, giving the corresponding computation in the multiplier.

4. **Normalisation.** Normalisation requires two iterations in the DSP slice and is implemented in a similar manner to alignment shifting. Leading zero-counting is implemented in logic and provides the shift amount input to the normalisation shift stage.

5. **Rounding.** The default IEEE 754 rounding mode, *round to nearest, tie to even*, is implemented using the DSP slice to calculate the  $norm(F_{sum}) + 1$  rounding alternative. With  $norm(F_{sum})$  already stored as an intermediate value in memory, the rounding stage then uses additional logic to select the correctly rounded value based on the  $G, R, S$  bits and the normalised mantissa.

6. **Exception Condition Checking.** Checking for exception conditions (overflow, all-zero, etc.) is done in logic, in the same way as in the logic-only implementations.

## 7. RESULTS

All designs are implemented in Verilog and synthesised using Xilinx ISE 14.4, targeting a Virtex-6 XC6VLX75T-1 FPGA. No manual Place-and-Route effort is done. Our own designs have been tested using automatic test generation and checking for a wide range of test vectors. The FloPoCo operators are generated using FloPoCo 2.4.0 targeting the same device. The CoreGen operators are generated using Xilinx CoreGen 14.4 [13] and are configured for full DSP-usage.

The post-place-and-route results for the LUT-only and fixed-DSP adder and multiplier designs are presented in Table 1. The designs are compared against the FloPoCo and Xilinx CoreGen operators. The Xilinx CoreGen and FloPoCo operators are tool-generated. We also present the results for our iterative DSP-based design. These tools are convenient for designers to use and can be configured to suit the application by optimising for some set of user-defined constraints. For comparison, we consider both speed- and latency-optimised FloPoCo operators.

The CoreGen adder with DSP makes efficient use of the DSP block and gives a lower logic usage. As the man-

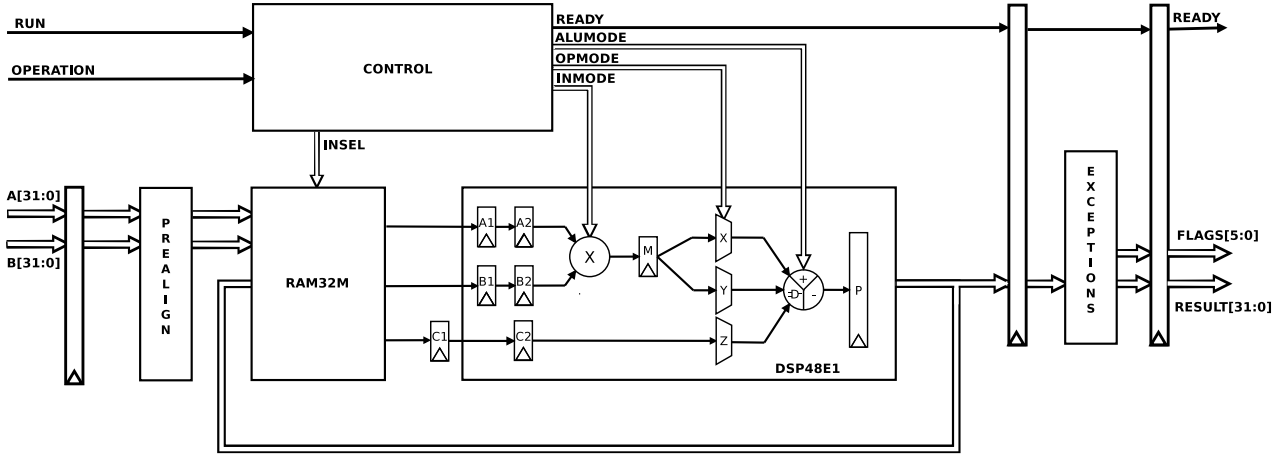


Fig. 3. Block Diagram of Iterative DSP-Based Floating Point Operator.

	Slice LUTs	Slice Registers	DSP48E1s	Latency	Frequency
<b>Adder/Subtractor (Logic Only)</b>					
CoreGen	398	551	0	12	440 MHz
FloPoCo <sup>1</sup>	633	745	0	13	336 MHz
FloPoCo <sup>2</sup>	293	437	0	6	310 MHz
Ours	467	494	0	11	408 MHz
<b>Adder/Subtractor (DSP-based)</b>					
CoreGen	221	330	2	11	408 MHz
Ours	469	524	1	10	408 MHz
<b>Multiplier (Logic Only)</b>					
CoreGen	681	629	0	8	306 MHz
FloPoCo <sup>1</sup>	799	412	0	6	320 MHz
FloPoCo <sup>2</sup>	716	195	0	3	254 MHz
Ours	759	311	0	4	109 MHz
<b>Multiplier (DSP-based)</b>					
CoreGen	125	170	2	8	376 MHz
FloPoCo <sup>1</sup>	160	165	2	4	337 MHz
FloPoCo <sup>2</sup>	127	50	2	2	221 MHz
Ours	116	223	2	8	405 MHz
<b>Iterative Combined (DSP-based)</b>					
Ours	242	161	1	22, 25 <sup>3</sup>	340 MHz

<sup>1</sup>Optimised for speed, <sup>2</sup>Optimised for low latency, <sup>3</sup>For multiplication (22) and addition/subtraction (25)

**Table 1.** PAR results for logic-only and fixed-configuration DSP floating point adder/subtractor and multiplier implementations.  $\omega_E, \omega_F = 8, 23$  (Single Precision)

tissa multiplier is expensive to implement in logic, the logic-only multipliers all show a high LUT usage. In the case of the multiplier implementations using DSP blocks, there is a large resource usage improvement compared to the logic-only designs, owing to the use of the hard multiplier in the DSP block.

Finally, comparing the iterative design to the operators described above, the iterative design is smaller than the logic-only designs and the adder designs using DSP blocks. In

the adder, shifting is the most expensive sub-operation to implement in logic, but in the iterative design this is done at no extra logic cost using multiple iterations through the DSP block. The iterative design uses 48% fewer LUTs and 67% fewer Registers than our logic-only adder, but runs at a slower clock frequency and with a higher latency. From the result tables we also see that for multiplication, using fixed-configuration DSP-slices gives better area savings over logic-only than for addition. This is mainly because the normali-

		Logic Only		Fixed-DSP		Iterative
		Add	Mult	Add	Mult	Both
1	Prealign	1	0	1	0	1
2	Align Shift	3	n/a	3	n/a	9
3	Execute	1	1	1	5	3-6
4	Normalise Shift	3	1	3	1	6
5	Rounding	1	1	1	1	2
6	Exceptions	1	1	1	1	1

**Table 2.** Comparison of latency in the individual suboperation stages in operators with different DSP-utilisation

sation operation in a multiplier is considerably simpler than for addition.

Table 2 shows a breakdown of the computational stages of our logic-only, fixed DSP, and iterative designs, showing how the iterative design uses multiple iterations, giving increased latency to lower the resource usage for the stages that are the most expensive to implement in logic, namely shifting.

The speed-optimised FloPoCo multiplier runs at a similar frequency and uses an almost equal number of registers, but uses 34% fewer LUTs. Again, the latency of the iterative design is higher, in this case 4-5 times higher. As noted earlier, the throughput of the iterative operator is lower than that of the fixed-datapath operators, as these can initiate a new floating point operation every clock cycle, whereas the iterative operator has an initiation interval of 20-22 cycles. The multipliers using DSP blocks are smaller than the iterative design, but make use of 2 DSP blocks. However, the iterative design provides increased flexibility as it is a combined multiplier and adder, as well as being smaller than the adder implementations. This makes the iterative design a good, flexible choice for applications requiring floating point capability for both multiplication and addition, where logic and DSP block usage is critical.

## 8. CONCLUSION AND FUTURE WORK

In this paper, we have described an iterative floating point combined multiplier and adder, and compared it to commercial, open source and custom designs for floating point multipliers and adders in FPGA. We have shown that the DSP48E1 slice allows for much flexibility as its input and operation modes can be configured dynamically and that this can be used to create a lean and flexible floating point operator, at the cost of higher latency. We have also noted some of the limitations of the iterative design and of floating point operators in FPGA in general.

Following on from this proof of concept, future work will aim to further develop the idea presented in this paper. This includes implementing a more mature and fully IEEE 754-2008 binary32 compliant floating point unit using a single DSP48E1 slice as an execution core, with support

for division and square root operations. We plan to release the design as open source in the future, and to look at the possibilities of integrating this work into the iDEA soft core processor [9]. We will also explore how using two or more DSP blocks in this iterative manner might help mitigate the latency cost and increase throughput. Application to double-precision floating point numbers will also be explored. A simpler floating point format with smaller mantissa bit size, better suited to the port width of the DSP48E1 block can also be investigated in future work.

## 9. REFERENCES

- [1] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.
- [2] F. de Dinechin, J. Detrey, O. Cret, and R. Tudoran, "When FPGAs are better at floating-point than microprocessors," in *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2008, p. 260.
- [3] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Proc. International Conference on Field Programmable Technology (FPT)*, 2010, pp. 110–117.
- [4] A. Malik and S.-B. Ko, "A study on the floating-point adder in FPGAs," in *Proc. Canadian Conference on Electrical and Computer Engineering*, 2006, pp. 86–89.
- [5] P.-M. Seidel and G. Even, "On the design of fast IEEE floating-point adders," in *Proc. IEEE Symposium on Computer Arithmetic (ARITH)*, 2001, pp. 184–194.
- [6] G. Conde and G. W. Donohoe, "Reconfigurable block floating point processing elements in Virtex platforms," in *Proc. International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp. 509–512.
- [7] H. Y. Cheah, S. A. Fahmy, D. L. Maskell, and C. Kulkarni, "A lean FPGA soft processor built using a DSP block," in *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, 2012, pp. 237–240.
- [8] B. Ronak and S. A. Fahmy, "Evaluating the efficiency of DSP block synthesis inference from flow graphs," in *Proc. International Conference on Field Programmable Logic and Applications (FPL)*, Oslo, Norway, 2012, pp. 727–730.
- [9] H. Y. Cheah, S. A. Fahmy, and D. L. Maskell, "iDEA: A DSP block based FPGA soft processor," in *Proc. International Conference on Field-Programmable Technology (FPT)*, 2012, pp. 151–158.
- [10] "IEEE standard for floating-point arithmetic," *IEEE Std. 754-2008*, 2008.
- [11] *UG193: Virtex-5 FPGA XtremeDSP Design Considerations*, Xilinx Inc., 2012.
- [12] *UG369: Virtex-6 FPGA DSP48E1 User Guide*, Xilinx Inc., 2011.
- [13] *LogiCORE IP Floating-Point Operator v6.0*, Xilinx Inc., 2012.