High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4

Per Karlström
Department of Electrical Engineering
Linkping University
Email: perk@isy.liu.se

Andreas Ehliar
Department of Electrical Engineering
Linkping University
Email: ehliar@isy.liu.se

Dake Liu
Department of Electrical Engineering
Linkping University
Email: dake@isy.liu.se

Abstract— **Since the invention of FPGAs, the increase in their size and performance has allowed designers to use FPGAs for more complex designs. FPGAs are generally good at bit manipulations and fixed point arithmetics but has a harder time coping with floating point arithmetics. In this paper we describe methods used to construct high performance floating point components in a Virtex-4. We have constructed a floating point adder/subtracter and multiplier which we then used to construct a complex radix-2 butterfly. Our adder/subtracter can operate at a frequency of 361 MHz in a Virtex-4SX35 (speed grade -12).**

I. Introduction

Modern FPGAs are a great asset as hardware components in small volume projects or as hardware prototyping tools. More features are added to the FPGAs every year, making it possible to perform computations at higher clock frequencies. Dedicated carry chains, memories, multipliers and in the most recent FPGAs, larger blocks aimed at DSP computations and even processors have been incorporated into the otherwise homogenous FPGA fabric. All of these improvements accelerate fixed point computations but it is harder to implement high performance floating point computations on FPGAs. One of the major bottlenecks is the normalization required in a floating point adder.

A floating point number consists of a mantissa ($\mathbf{M}$) and an exponent ($\mathbf{e}$) as shown in equation (1). The sign of the mantissa must be represented in some way. One way is to use a two's-complement representation, another common approach is to use a sign magnitude representation where a sign bit ($\mathbf{S}$) decides the sign and mantissa holds the magnitude of the number. The sign of the exponent must also be represented. A common approach is to store the exponent in an excess representation, where the exponent is treated as a positive number from which a constant is subtracted to form the final exponent. Since the mantissa in a normalized binary floating point number using the sign bit representation always will have a single one in the MSB position, this bit is normally not stored together with the floating point number. The IEEE 754, a standard for floating point numbers [4], dictates the format presented in equation (2). The IEEE 754 single precision format is 32 bit wide and uses a 23 bit fraction, an eight bit exponent represented using excess 127, and one bit is used as a sign bit.

$$x = \mathbf{M} \cdot 2^{\mathbf{e}} \tag{1}$$

$$x = (-1)^{\mathbf{S}} \cdot 1.\mathbf{M} \cdot 2^{\mathbf{e}-excess} \tag{2}$$

The overall goal of our design was to balance throughput and latency. Low latency is important if the floating point unit is to be used as building blocks for systems where the algorithms are not known beforehand, e.g. if the units are to be used in a processor. In theory, if latency was not a constraint, pipeline stages could have been added until a higher clock frequency could not be achieved.

We chose to implement the most commonly used operations, addition, subtraction, and multiplication. In order to test these components in a realistic environment we constructed a complex radix-2 butterfly kernel using our components.

We have tested the floating point units on an FPGA from the Virtex-4 family (Virtex-4 SX35-10). For further details about the Virtex-4 FPGA, see the Virtex-4 User Guide [2]. The Virtex-4 contains a number of blocks targeted at DSP computations, these blocks are called DSP48-blocks and are thoroughly described in the XtremeDSP user guide [3].

Floating point arithmetics is useful in applications where a large dynamic range is required or in rapid prototyping for applications where the required number range has not been thoroughly investigated. Our floating point format is similar to IEEE 754 [4]. An implicit one is used and the exponent is excess-represented. However, we do not handle denormalized numbers, nor do we honor NaN or Inf.

The reason for excluding denormalized numbers is due to the large overhead in taking care of these numbers, especially for the multiplier. These are commonly excluded from high performance systems, i.e. the CELL processor does not use denormalized numbers for the single precession format in its SPUs [6].

Our implementation has no rounding, therefore the results after the addition and multiplication are truncated to fit the mantissa size. It is usually easier to add en extra mantissa bit to handle the same precision as achieved when using more elaborate rounding schemes.

II. Related Work

A number of attempts at constructing floating point arithmetics in FPGAs have been done and presented in the academia. Although many of the papers are a bit old and few target modern FPGAs such as the Virtex-4.

High-performance floating point arithmetics on FPGA is discussed in [5] Although the paper has some interesting figures about the area versus pipeline depth tradeoff, their design seems to be a bit to general to utilize the full potential of the FPGA. I.e. to reach 250 MHz for the adder they have to use 19 pipeline stages on a Virtex2Pro speed grade -7.

To be fully IEEE 754 compliant the FPU needs to, in one way or another, support denormalized numbers, be it either with interrupts, letting the processor deal with these uncommon numbers or having direct support for denormals in hardware. For a good discussion on different strategies to handle denormals see [7]. Although it is a good general discussion the paper does not cover any FPGA specific details.

An interesting approach to tailor floating point computations to FPGAs are to use higher than 2-radix floating points since this maps better to the FPGA fabric. This is better described in [11].

Full IEEE 754 rounding requires the FPU to support, round to nearest even, round to minus infinity, round to positive infinity, and round to zero. A more detailed discussion about rounding is presented in [8]. This paper does not deal with any FPGA specific implementations.

Since the invention of FPGAs and their increase in performance, IP cores for FPGAs has started to appear in the market. Both Nallatech [9] and Xilinx [10] has IP cores for double and single precision floating point format. Neither of these companies publish low level techniques used in their IP cores.

III. Methodology

As a reference for the RTL code we implemented a C++ library for floating point numbers. The number of bits in the mantissa and exponents could be configured from 1 to 30 bits. The C++ model was later used to generate the test vectors for the RTL test benches. Using a mantissa width of 23 and an exponent width of 8 the C++ model was tested against the floating point implementation used in the development PC. The only differences occurred due to the different rounding modes.

Initial RTL code was written using Verilog adhering to the C++ model. The performance of the initial RTL model was evaluated and the most critical parts of the design were optimized to better fit the FPGA. This was repeated until the performance was satisfactory and no bugs were discovered by the test benches.

IV. Implementation

The implementation was written in Verilog and ISE 8.2i was used to synthesize, place, and route the design.

A. Multiplier

A floating point multiplier is conceptually easy to construct. The new mantissa is formed as a multiplication of the old mantissas. In order to construct a good multiplier some FPGA specific optimizations were needed. The 24×24 bit multiplication of the mantissa is constructed using four of the Virtex-4's DSP48 blocks to form a 35×35 bit multiplier with a latency of five clock cycles. For a thorough explanation of how to construct such a multiplier the reader is referred to [3]. The new exponent is even easier to construct, a simple addition will suffice. The new sign is computed as an exclusive-or of the two original signs. The result of the multiplication has to be normalized, this is a simple operation since the most significant bit of the mantissa can only be located at one out of two bit positions given normalized inputs to the multiplier. The exponent is adjusted accordingly in an additional adder.

B. Adder/Subtracter

A floating point adder/subtracter is more complicated than a floating point multiplier. The basic adder architecture is shown in Figure 1. The first step compares the operands and swaps them if necessary so that the smallest number enters the path with the alignment shifter. If the input operands are non-zero, the implicit one is also added in this first step. In the next step, the smallest number is shifted down by the exponent difference so that the exponents of both operands match. After this step, an addition or subtraction of the two numbers are performed. A subtraction can never cause a negative result because of the earlier comparison and swap step.

The normalization step is the final and most complicated step. It is implemented using three pipeline stages. Figure 2 depicts the architecture of the normalizer. The following is done in each pipeline stage:

1) The mantissa is processed in parallel in a number of modules, each looking at four bits of the mantissa. The first module operates on the first four bits and outputs a normalized result assuming a one was found in these bits. An extra output signal, shown as dotted lines in Figure 2, is used to signal if all four bits were zero. The second module assumes that the first four bits were all zero and instead operates on the next four bits, outputting a normalized result. This is repeated for the remaining bits of the mantissa. Each module also generates a value needed to correct the exponent, this is marked as dotted lines in Figure 2.
2) One of the previous results, both mantissa and exponent offset value, is selected to be the final output. If all bits were zero, a zero is generated as the final result.
3) The mantissa is simply delayed to synchronize with the exponent. The exponent is corrected with the offset selected in the previous stage.

Our normalization uses a rather hardware expensive approach a less hardware expensive architecture could be used if deeper pipelines were allowed. The modules in the first stage of the normalizer looks at four bits each, the choice to look at

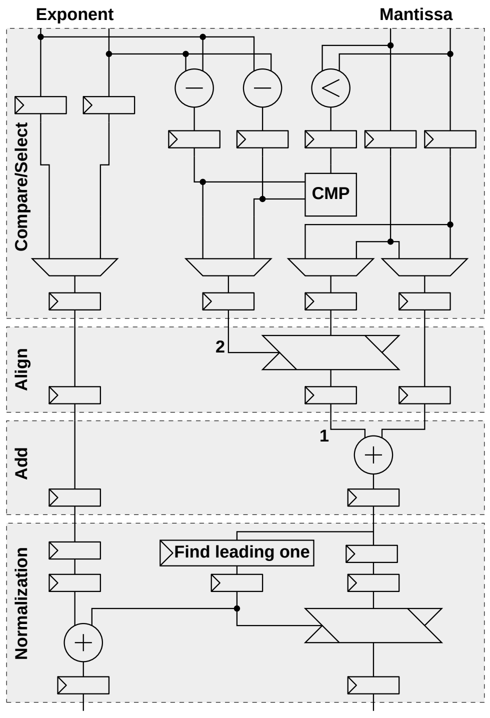

Fig. 1. The overall adder architecture

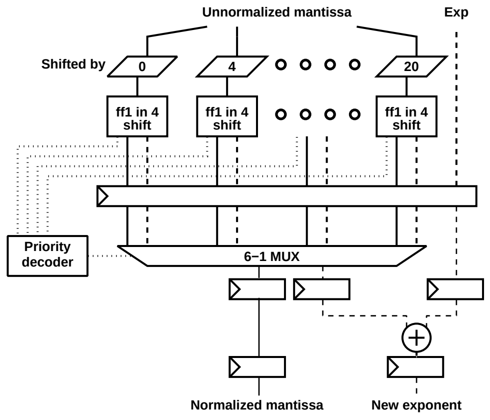

Fig. 2. The normalizer architecture

four bit was done since it maps well to the four input LUTs of the Virtex-4.

C. Low Level Optimizations

Initially the adder met timing at 250 MHz. It did not achieve this performance once it was inserted into a complex radix-2 butterfly. At this point further optimizations were required. One FPGA specific optimization was to make sure that the adder/subtracter was implemented using only one LUT per bit. Figure 3 shows a bit cell of the optimized adder. An additional input signal is used to zero out the mantissa from the pre-alignment step, marked with 1 in Figure 1. This is done so that the shifter in the align step only has to consider the five least significant bits in the exponent difference, marked with 2 in Figure 1. If one of the more significant bits is one, the mantissa is shifted so much that all its bits are zero. This is handled by the *Set to zero* signal in Figure 3. A similar way to achieve the same result is by using the reset input of the flip flops, although this will limit the maximum clock frequency.

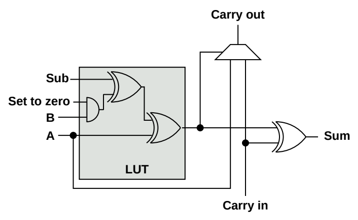

Fig. 3. Combined adder and subtracter

D. Testing

In addition to testing the RTL implementation against the C++ model, we have also tested a radix-2 complex butterfly, using a 15 bit mantissa and 10 bit exponent format, for real in a Virtex-4SX35 speed grade -10. This design was successfully run at a clock frequency of 250 MHz.

V. Results

Table I lists the final resource utilization in the FPGA for various components. The numbers in the table is the maximum frequency the place and route tool could achieve with a a Virtex-4 speed grade -10. These speeds also assumes a clock with no jitter. All clock frequencies are rounded down to the nearest integer from the results reported by the place and route tools. We have focused our measures and comparisons on the adder since it is the bottleneck module in our current design.

	Adder	Multiplier
LUTs	557	88
Flip Flops	375	244
DSP48	0	4
Speed	275 MHz	327 MHz
Stages	7	6

TABLE I

Component Statistics

Table II list various performance metrics over different devices and speed grades.

	Latency	Speed in device (MHz)		
Module		XC4VSX-10	XC4VSX-11	XC4VSX-12
23 bit $\mathbf{M}$, 8 bit $\mathbf{e}$				
Adder	7	275	318	361
Multiplier	6	327	400	451
15 bit $\mathbf{M}$, 10 bit $\mathbf{e}$				
Adder	6	285	330	369
Multiplier	3	338	375	418

TABLE II

Performance in various devices.

Table III compares the performance of the 23 bit format floating point adder using the best speed grades from a number of FPGA families from Xilinx.

	XC4VSX-12	XC2VP-7	XC2V-6	XC3SE-5	XC3S-5
Freq.	361 MHz	288 MHz	250 MHz	202 MHz	174 MHz

TABLE III

FAMILY COMPARISON

Table IV list the resource utilization of the steps in Figure 1. To avoid the extra delays associated with the FPGA I/O pins two extra pipeline stages before and and one stage after were inserted into the top module. These extra flip flops are not included in the resource utilization metrics.

Table V compares our results from the adders (DA) against some other results published, although we do not handle denormalized numbers or all rounding modes in our design we are confident that no more than three pipeline stages needs to be added to make the units fully IEEE 754 complaint. USC [5] does not consider NaN or denormals and Nallatech [9] uses an alternative internal floating point format and can thus also avoid handling denormals. Thus the comparisons here are not completely fair they still give a good picture of how the performance of our floating point units compare to other FPGA implementations.

	23 bit $\mathbf{M}$, 8 bit $\mathbf{e}$		15 bit $\mathbf{M}$, 10 bit $\mathbf{e}$	
	LUT	FF	LUT	FF
Compare/Select	113	149	99	129
Align	97	57	100	44
Add	31	33	26	33
Normalization	326	222	225	145
Total	567	461	450	351

TABLE IV

ADDER RESOURCE UTILIZATION

	XC2VP-6			XC2VP-7	
	DA	Nallatech	Xilinx	DA	USC
	Adder				
Speed	248 MHz	184 MHz	269 MHz	288 MHz	250 MHz
Latency	7	14	11	7	19

TABLE V

COMPARISON WITH OTHER IMPLEMENTATIONS

VI. DISCUSSION

There are a number of opportunities for further optimizations in this design. For example, instead of using CLBs for the shifting, a multiplier could be used for this task by sending in the number to be shifted as one operand and a bit vector with a single one in a suitable position as the other operand.

If the application of the floating point blocks are known it is possible to do some application specific optimizations. For example, in a butterfly with an adder and a subtracter, operating on the same operands, the first compare stage could be shared between these. If the application can tolerate it, further pipelining could increase the performance significantly. If the latency tolerance is very high, bit serial arithmetics could probably be used as well. In this project we limited the pipeline depth to compare well with FPUs used in CPUs.

According to a post on comp.arch.fpga it is possible to achieve 400MHz performance in a XC4VSX55-10 for IEEE 754 single precision floating point arithmetics [1]. Few details are available but a key technique is to use the DSP48 block for the adder since an adder implemented with a carry chain would be too slow. The post normalization step is supposed to be implemented using both DSP48 and Block RAMs. The pipeline depth of this implementation is not known, although what is known is that the normalization consists of 11 pipeline stages.

It would also be interesting to look at the newly announced Virtex-5 FPGA. The 6-LUT architecture should reduce the number of logic levels and routing all over the design. As an example, one could investigate if the parallel shifting modules in the normalizer should take six bits as input since it could map well to the six input LUT architecture of the Virtex-5 or if the fact that a 4-to-1 mux can be constructed in a six input LUT still favors the current four bits per module architecture.

A final step of this research would be to implement all rounding modes and at least generate flags so a software solution can deal with denormals and the other special numbers defined in IEEE 754.

VII. CONCLUSION

We have shown that it is possible to achieve good floating point performance with low latency in modern FPGAs. To make maximal use of an FPGA it is important to take into account the specific architecture of the targeted FPGA. The most important optimization we did was to perform the normalization in a parallel fashion.

The parallel normalization approach proved to be efficient since it reduced the number of pipeline stages needed to perform the normalization operation.

REFERENCES

[1] *Andraka, Ray*; Re: Floating point reality check, news:comp.arch.fpga, 14 May 2006

[2] *Xilinx*; Virtex-4 User Guide

[3] *Xilinx*; XtremeDSP for Virtex-4 FPGAs User Guide

[4] *ANSI/IEEE Std 754-1985* IEEE Standard for Binary Floating-Point Arithmetic

[5] *Govindu G., Zhuo L., Choi S, and Prasanna V.* Analysis of High-performance Floating-point Arithmetic on FPGAs

[6] *Oh H., Mueller S. M. Jacobi C. et al* A Fully Pipelined Single-Precision Floating-Point Unit in the Synergistic Processor Element of a CELL Processor

[7] *Schwarz M. E., Schmookler M., and Trog S.* Hardware Implementations of Denormalized Numbers

[8] *Santoro M. R., Bewick G., and Horowitz M. A.* Rounding Algorithms for IEEE Multipliers

[9] *Nallatech* http://www.nallatech.com

[10] *Xilinx Floating-point Operator v2.0* http://www.xilinx.com/bvdocs/ipcenter/data_sheet/floating_point_ds335.pdf

[11] *Cantanzaro R. and Nelson B.* Higher Radix Floating-Point Representation for FPGA-Based Arithmetic