Low-Latency Double-Precision Floating-Point Division for FPGAs

Björn Liebig, Andreas Koch
Embedded Systems and Applications Group (ESA), Technische Universität Darmstadt
Email: {bl,ahk}@esa.cs.tu-darmstadt.de

***Abstract*—**
With growing FPGA capacities, applications requiring more intensive use of floating-point arithmetic become feasible candidates for acceleration using reconfigurable logic. Still among the more uncommon operations, however, are fast double-precision divider units. Since our application domain (acceleration of custom-compiled convex solvers) heavily relies on these blocks, we have implemented low-latency dividers based on the Goldschmidt algorithm that are accurate up to 1 bit of least precision (1-ULP). On Virtex-6 devices, our units operate at 200 MHz and significantly outperform other state-of-the-art 1-ULP dividers. We evaluate our blocks both stand-alone, as well as on the application-level when used for the high-level synthesis of the convex solver cores.

I. Introduction

Floating point arithmetic both in software as well as in hardware is often implemented following the standard IEEE 754 [1]. While for many applications single-precision computation suffices, and is preferable both for the reduced storage size as well as increased memory bandwidth, some applications require double-precision operations for numerical stability. On general-purpose processors (GPPs), which use commonly use the same FPU for both single and double precision, arithmetic operations often show similar performance regardless of the precision used. On FPGAs, however, double precision operations often have a significantly higher latency than single precision [2]. Also, with the intensive use of FPGAs to accelerate classical DSP algorithms, optimization efforts have concentrated on multiplication, addition, and multiply-add units. High-speed division, which is relevant for novel applications such as convex solvers or model-predictive control, has received less attention. Since our research concentrates on these fields, we have worked to alleviate this deficiency.

Multiplicative division methods such as Newton-Raphson and Goldschmidt [3] offer quadratic convergence and thus a reduced latency when compared to the frequently used digit-recurrence methods, e.g. the SRT algorithm. An initial approximation, often retrieved from a lookup table, can be used to decrease the number of iterations, and further increase the performance. The higher performance of multiplicative division methods usually comes at the expense of an increased area, which may become acceptable with growing FPGA capacities.

Another way to further reduce latency and area consumption is to perform Faithful Rounding (FR) instead of IEEE754 Conforming Rounding (CR). The latter requires the selection of the *single* representable value closest to the infinitely accurate result, while the former can arbitrarily return any one of the *pair* of closest representable values bracketing the accurate result. In literature the error is often expressed as a multiple of a unit of least precision (ULP), which has the CR error at up to 0.5 ULP, while FR has an error of up to 1.0 ULP.

The improved accuracy of CR comes at a significantly increased hardware cost, however: In multiplicative division methods, it is often not possible to directly compute the CR result. Instead, an FR result is returned and post-processing is required to actually perform CR. Thus, if FR suffices for a given application, as it does for our convex solver synthesis, it is worthwhile to design specialized arithmetic units for FPGA implementation.

The next section will give an overview of related work on division units in general, and on FPGA division units in particular. Section III covers two FPGA implementations of 1-ULP accurate double-precision division units, both based one recent publications [4], [5], but originally targeting the ASIC domain. The two division strategies are compared while putting special focus on latency and area efficiency of the FPGA implementation of the mantissa division. In Section III-A, we apply FPGA-specific optimizations to the superior unit to further minimize latency. The proposed divider is then compared to state-of-the-art FPGA dividers in Section IV-A. Additionally, the performance and area impact of using the high-speed divider is examined on the use-case of a number of convex solver accelerators. Section V concludes the discussion looks forward to future work.

II. Related Work

A. Floating-Point Representation

In IEEE 754 [1], a finite number R is represented by three components named mantissa (M), exponent (E) and sign (S), so that $R = M * 2^{E-b} * (-1)^S$,where the bias b is a positive integer. The standard specifies a number of fundamental number formats with pre-defined widths of the M and E fields and bias values.

As an example, Figure 1 shows the structure of the widely used *binary64* format, more commonly known as *double-precision*.

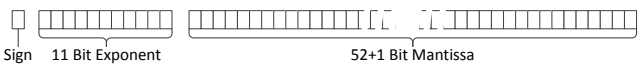

Fig. 1. IEEE 754 double-precision number format

The format also guarantees unique representations of each number, thus avoiding ambiguity. This is achieved by scaling the mantissa so that its most-significant **1** bit actually becomes the most significant bit (msb) of the M field in the standardized binary representation. Since this leads to all numbers (with the exception of Zero) having an M field beginning with a **1** bit,

this bit is no longer explicitly stored (implied **1**), thus ensuring that the mantissa will be in the range $1 \leq M < 2$.

For very small values, scaling the mantissa as described above may lead to an exponent that cannot be represented $E < E_{min}$. In this case, the IEEE 754 standard defines the mantissa to be stored *without* the implied leading **1** as a *subnormal* number. Since this irregularity provides only a minor increase of the range of representable numbers, but carries a significant area and latency overhead, it is commonly omitted in FPGA implementations of arithmetic operators (e.g., [2]). In our work, we also follow this approach and treat subnormals as zero, which does not affect the operation of the convex solvers using the blocks.

B. Hardware Division

Methods for (floating point) division can be divided into three categories: digit recurrence (or subtractive), lookup table based, and multiplicative (or iterative) methods.

Digit recurrence algorithms compute one digit of the result per clock cycle and thus achieve linear convergence. For higher performance, a radix higher than two can be chosen, allowing to gather to more than a single bit of the result per cycle at cost of increased area requirements. One example for a digit recurrence algorithms is the SRT algorithm [6].

Lookup table based methods are used to compute the reciprocal of the divisor, and use a later multiplication to actually compute the quotient. For small word widths, it is possible to use the entire divisor as address for a single lookup. However, the exponential growth of the table size prevents larger lookups. For larger word widths, bipartite tables [7] or piecewise polynomial approximations can be used. In piecewise polynomial approximations, the reciprocal function is divided into 2^m equally sized partitions. For each partition, a polynomial approximation is performed, with the polynomial coefficients required for each partition being stored in a lookup table with m entries. The m most significant bits of the divisor are then used as address to retrieve the coefficients. Afterwards, the polynomial approximation of the division can be computed using only multiplications and additions.

Two examples for popular **multiplicative methods** are the Newton-Raphson and the Goldschmidt [3] algorithms. While Newton-Raphson computes the reciprocal of the divisor, again requiring a final multiplication, Goldschmidt performs a direct computation of the quotient. Both methods offer quadratic convergence, making them especially attractive for computations that require a high accuracy (e.g., double precision), but usually requiring multiple iterations to reach the desired accuracy. This number of iterations can be reduced significantly by providing a good initial approximation, which can be computed using another division method, e.g., a lookup table based method. Using Goldschmidt's approach for the computation of $Q = Y/X$, a sequence is computed such that

$$\frac{A_{i+1}}{B_{i+1}} = \frac{A_i * Z_i}{B_i * Zi}$$

with $A_0 = Y, B_0 = X$ and $Z_i = 2 - B_i$. In this sequence, A_i converges to Q.

C. FPGA Implementations

A number of these approaches have been adapted for FPGA implementation of double-precision division in the past. The FloPoCo floating point library uses a Radix-4 digit recurrence method [8]. As most digit recurrence methods, it allows the computation of the CR result. However, performance for larger word widths is limited due to the linear convergence of digit recurrence methods.

The VFLOAT library implements a lookup table based Taylor approximation to compute the reciprocal of the divisor [9], which is then multiplied with the dividend. The result may differ from the CR result computed by IEEE754 conform dividers (such as a Xilinx IP core), which lead the authors to classify their unit as 1-ULP accurate. VFLOAT division was shown to be faster than the corresponding Xilinx IP core.

A faster FPGA division unit was presented by Pasca [10]. It uses a piecewise polynomial approximation of second degree as seed value for a single Newton-Raphson iteration, which allows the complete double-precision divider to compute a FR result. The block offers superior performance when compared to other state-of-the-art division units. However, as the Newton-Raphson method only computes the reciprocal, an additional multiplication must be performed afterwards. In contrast, the Goldschmidt method allows this multiplication to be performed in parallel and thus could reduce the overall latency further.

D. Truncated Multiplication

To reduce the size of a division unit's internal wide fixed-point multipliers, the use of *truncated* multipliers is suggested in [11]. The truncation makes use of the fact that in fixed-point multiplication, the result is often rounded to avoid a growth in word length. A truncation of the least significant bits during the multiplication is proposed, which reduces the hardware cost by $25\ldots35\%$ introducing only a small computation inaccuracy. To compensate for this and limit the maximum error, a correction-constant is added to the final result.

In [12], Banescu adapts this concept for DSP tiling on FPGAs. Instead of truncating all bits in the adder tree less significant than position x, complete DSP blocks are truncated/omitted, or replaced by smaller LUT-based multipliers with reduced input word width. Such truncated multipliers were also used in [10].

E. Goldschmidt Division for ASICs

Our work is based on two prior efforts on implementing Goldschmidt division for ASICs. In the first approach [4], which we will refer to as **TripleGS**, a small lookup table *reciproc* containing the reciprocal of X is used to determine a good seed value for A_0. Afterwards, a modified Goldschmidt algorithm is iterated two times to compute a 1-ULP accurate *single-precision* result. To reach *double-precision* accuracy, an extension using a third Goldschmidt iteration is proposed. The modified version of the Goldschmidt [13] algorithm is used to keep the required multiplications narrow. The difference between the traditional Goldschmidt method and the modified version used in the paper is shown in Table I. The computation of A_i leads to the same result in both versions. However, with growing i, the required word width of R_i is less in the modified version than in the original. Furthermore, the squaring operation performed for R_i requires fewer resources than a full multiplication.

Figure 2 shows the complete computation performed for a double-precision division.

A second approach, which we will refer to as **PolyGS**, is proposed in [5], a work which covers the high-speed computation of double-precision floating point reciprocal, division,

method	iteration step	initial value
Traditional Goldschmidt	$A_{i+1} = A_i * (2 - B_i)$ $B_{i+1} = B_i * (2 - B_i)$	$A_0 = Y * reciproc[X]$ $B_0 = X * reciproc[X]$
Modified Goldschmidt	$A_{i+1} = A_i * (1 + R_i)$ $R_{i+1} = R_i^2$	$A_0 = Y * reciproc[X]$ $R_0 = 1 - X * reciproc[X]$

TABLE I. TRADITIONAL VS. MODIFIED GOLDSCHMIDT METHOD

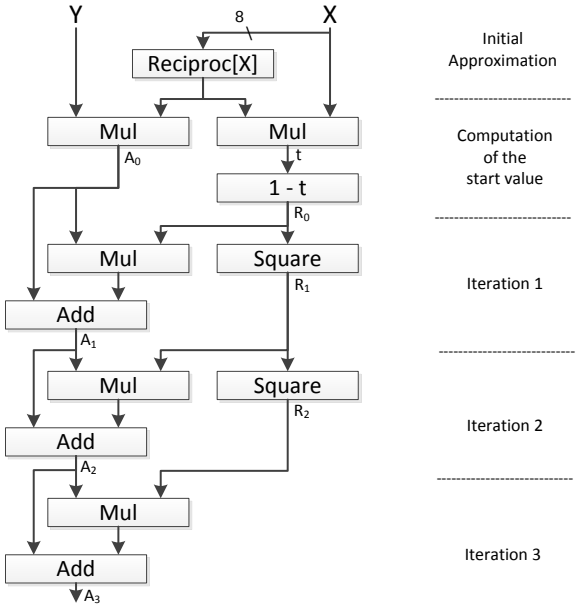

Fig. 2. TripleGS: Compute $Q = Y/X$ using a small lookup table and three Goldschmidt iterations

square root, and inverse square root operations. We just concentrate on division, which has the structure shown in Figure 3. First, a piecewise second-degree polynomial approximation is performed. The first 9 bits of mantissa of X are used for the table lookup of the three polynomial coefficients C_0, C_1, C_2. The result R_d of this approximation is then used as input for a single iteration of the Goldschmidt algorithm. As R_d is already accurate to 30 bits, only a single Goldschmidt iteration is required to compute the final result. As in *TripleGS*, the modified Goldschmidt algorithm [13] is used to reduce area and latency.

As shown in Figures 2 and 3, both approaches have a chain of 4 multiplier/squaring units in their critical path (divisor to quotient). In the next section, we examine how they can actually be mapped to the Virtex-6 FPGA architecture.

III. FPGA IMPLEMENTATION, OPTIMIZATION, AND ACCURACY EVALUATION

In this section, we examine initial FPGA implementations of the TripleGS and PolyGS approaches, suggest optimizations, and evaluate the accuracy of the units. Both units target the Virtex-6/-7 architecture using Block RAM (BRAM) for the lookup tables and DSP blocks for composing the multipliers and squaring units.

The small lookup table used in the **TripleGS** approach takes the 8 most significant bits of X as input for addressing and returns a 9 bit wide approximation of $1/X$. The total memory usage is only 2304 bits, easily fitting into a block RAM of type RAMB18E1.

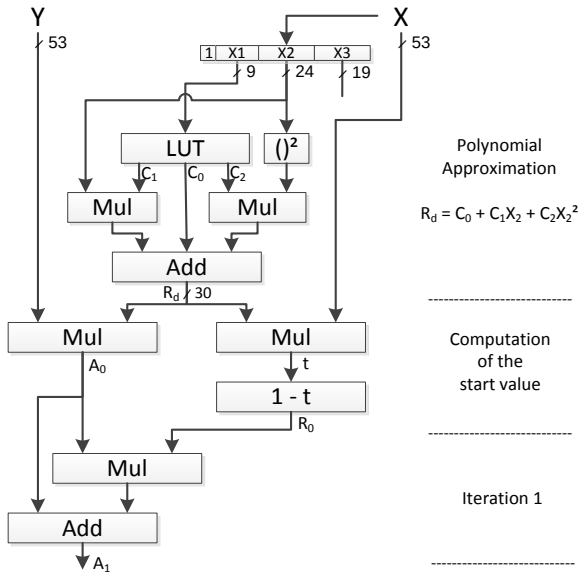

Fig. 3. PolyGS: Compute $Q = Y/X$ using a polynomial approximation and a single Goldschmidt iteration

The initial lookup is followed by three iterations of the (modified) Goldschmidt method. Although Han et al. [4] claim to aim for reduced area, and although the size of the multiplier has already been reduced by applying the modified Goldschmidt algorithm instead of the original one, some wide multiplications remain in the data path, especially in the first and second iteration (49x49, 42x42).

The traditional tiling of the multiplications into several DSP instances thus leads to a large DSP count. To some extent, this problem is due to the specifics of the target architecture, as some of the multipliers are ill-matched to the 17x24 bit DSP Slices of the devices (e.g., the first two multipliers after the lookup table require input bit widths of 9x53).

Note that while Virtex-5/-6/-7 DSP blocks offer 18x25 bit signed multiplication, only 17x24 bit are available for unsigned inputs (which are used for floating-point division). Furthermore, for FPGAs, the area savings in TripleGS due to using squaring units instead of two-operand multipliers are not as large as they would be in an ASIC design.

Figure 4 shows the resulting division unit, pipelined for 200 MHz operation as shown by the red dashed horizontal lines. For adding more than two operands, Carry Save Adders (CSA) are used to sum the DSP block outputs. In total, 30 DSP blocks have been instantiated to achieve a total latency of 12 cycles for the division.

The **PolyGS** approach [5] partitions the divisor X into three parts: $X_1 = X[51:43]$, $X_2 = X[42:19]$, $X_3 = X[18:0]$. X_1 is used to perform a lookup of the polynomial coefficients C_0, C_1, and C_2 of the piecewise polynomial approximation. The total bit width of C_0, C_1, and C_2 is $30+20+12 = 42$ bits, requiring $42*2^9$ bits of memory that can be held in single block RAM RAMB36E1. X_2 is then used as input value to the polynomial $C_2 \cdot X_2^2 + C_1 \cdot X_2 + C_0$. As a further optimization, it suffices to use just the uppermost 16 bits of X_2 for the squarer (please see [5] for details). The entire polynomial approximation requires only relatively narrow multipliers: $C_1 \cdot X_2$ is a 20b x 24b unit, $C_2 \cdot X_2^2$ uses 12b x 16b, and the squarer X_2^2 is organized as 16b x 16b.

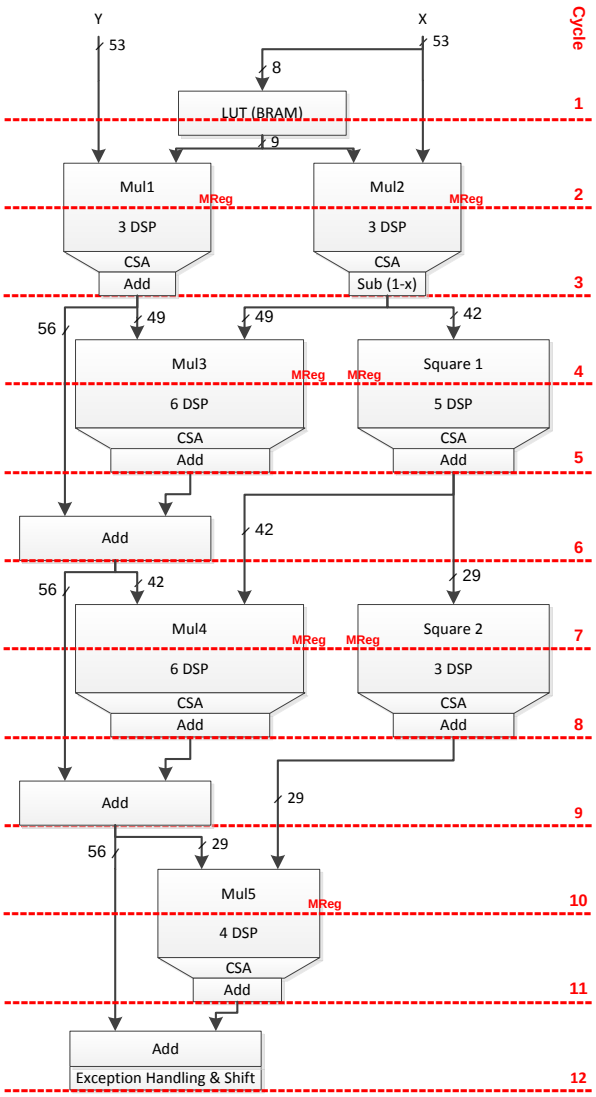

Fig. 4. FPGA implementation of TripleGS (12 cycles, 30 DSPs)

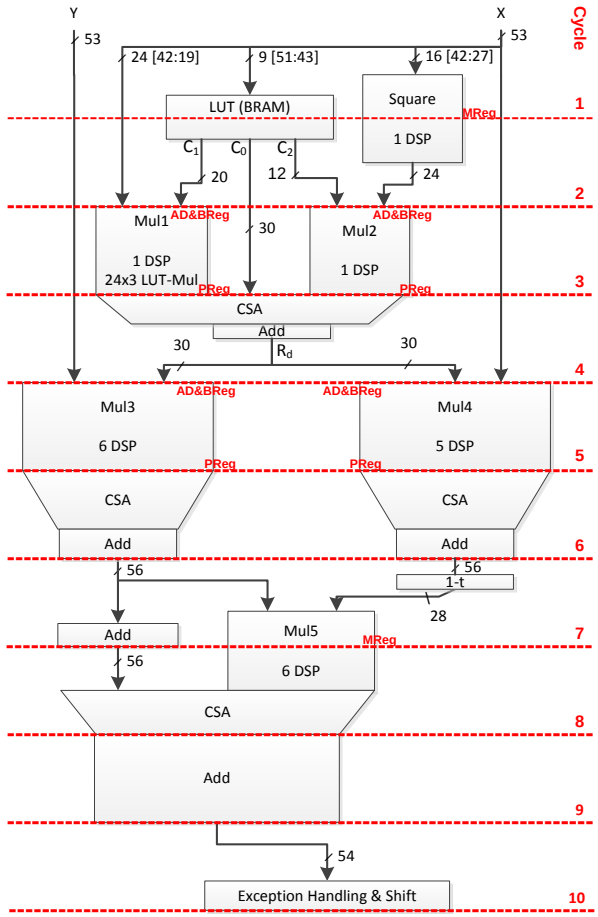

Fig. 5. FPGA implementation of the PolyGS approach (10 cycles, 20 DSPs)

All of these require just a single DSP block (assisted by a few slices of logic for $C_1 \cdot X_2$). The multipliers after the approximation are wider and must be composed from several DSP blocks (up to six). In total, 20 DSP blocks are required after performing DSP tiling (using LUT-based sub-multipliers only if one operand is less than 4 bits wide).

Figure 5 shows the pipelined implementation of the second division unit. The narrow arithmetic for the polynomial approximation allows a tight pipelining of these multipliers. In summary, PolyGS requires only 10 cycles, but the critical path length (four multipliers) remains similar to that of TripleGS.

Since TripleGS is both larger and slower than PolyGS already at the microarchitecture level (as well as on the layout level, as will be shown later in Section IV-A), further lower-level optimizations will be applied only to PolyGS.

A. *FPGA specific optimization of latency and area*

After deciding to use PolyGS as the baseline for further FPGA-specific optimization, we first consider the critical path of Figure 5, specifically the two multipliers Mul3 and Mul4. On closer examination, it is obvious that for each one, only a single one of the inputs, driven by the 30 bit adder, is actually timing critical. This specific configuration allows the exploitation of the DSP blocks' integrated pre-adder feature.

However, the direct approach of connecting the individual sum and carry outputs of the Carry Save Adder (CSA, in Cycle 4 of Figure 5) to the pre-adders led to excessive routing delays. These are avoided by replacing the CSA with a discrete conventional binary adder (computing $C_1X_2+C_0$, as shown in Figure 6), which is then connected to the pre-adders of Mul3 and Mul4.

For clarity, Figure 6 omits the following detail: Since R_d is now computed inside of the DSPs (using the pre-adders, as R_{d1} in Mul3 and R_{d2} in Mul4), rounding has to be performed differently than in Figure 5 (where it occurred before the DSPs). Rounding each of the two R_d computations individually, however, would lead to double the previous maximum rounding error: The two paths through Mul3 and Mul4 converge later and accumulate each of their individual rounding errors. As a solution, R_{d1} is rounded, R_{d2} is truncated, and a condition bit is asserted if $R_{d1} - \text{round}(R_{d1}) + R_{d2} - \text{trunc}(R_{d2}) > 0.5$, i.e., an addition of **1** is required to correctly round *up*. This bit is evaluated in the small LUT-based sub-multipliers assisting the DSPs in Mul3 and Mul4 and leads to the addition of the extra 1 if required for rounding.

Another cause for delay is the large addition required to compute the 56-bit result of Mul3 at the end of Cycle 6. In contrast to the final addition of the beginning of Cycle 9 (required for an IEEE 754 compatible result in binary representation),

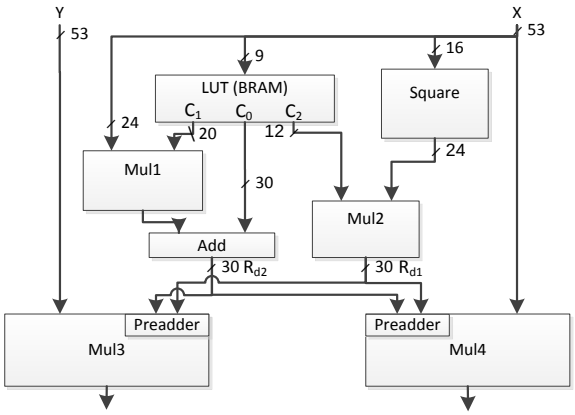

Fig. 6. Using the DSP pre-adders in Mul3 and Mul4

internal additions can be partitioned into narrower (and thus faster) operations using Carry Save Arithmetic. Here, the result of Mul3 is not completely computed in binary, but in a partial Carry Save representation, with extra carry bits inserted at bit positions 30 and 42. This breaks the original 56b computation into three narrower additions that can run in parallel, yielding a result consisting of 56 bits plus 2 carry bits. The carry bits will be handled separately in the LUT-based sub-multipliers of Mul5. The widening of the result of Mul4 from 28 to 29 bits will be explained below.

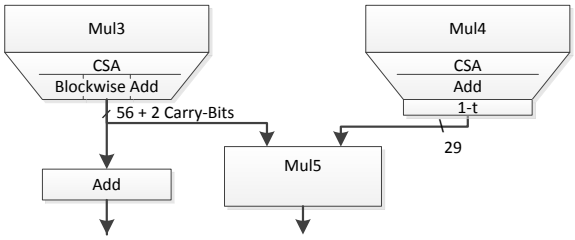

Fig. 7. Using a partial carry save format for the result of Mul3

Based on the work of [12], the multiplier design and tiling can be reworked to show better results in both performance and area consumption. As stated earlier, truncated multipliers reduce resources, delay, or power consumption. In the division unit under discussion, the bit width of the multiplier output is always much smaller than the sum of its input bit widths (see Figure 5). This indicates that the full output width is not required in the next step. All multipliers are candidates for a truncated multiplication. However, Mul2 and the squaring unit are already smaller than a single DSP and therefore not considered.

Using truncated multipliers generally increases the maximum error of an arithmetic unit. However, we still require 1-ULP accuracy at the final and thus need to carefully compensate for the accuracy loss due to tiling. We achieve this by selectively increasing the width of some operations. The error introduced by rounding the result of Mul4 to 56 bits has been shown to be the most significant error term [12]. To compensate, its bit width was increased here to 57 bits (56 fractional bits), reducing the rounding error at Mul4 by 50%. Also according to [12], the most significant 29 bits of 1-Mul4 are known to be all zero (or all one), so only $57 - 29 + 1 = 29$ bits are actually required for the two's complement representation of 1-Mul4.

With these constraints, we can use the tiling for Mul1, Mul3, Mul4, and Mul5 shown in Figure 8 for the truncated multiplications. Empty areas in the multiplication rectangle represent the truncated parts. As another measure to compensate for the truncation error, half of the maximum result of the truncated areas is added to the result. E.g., if a 2x5 bit multiplier is truncated, $11_b \cdot 11111_b/2 = 101110_b$ is added to the product as compensation. This addition is performed either in the adder tree of the multiplication, or using an empty C-input of a DSP block.

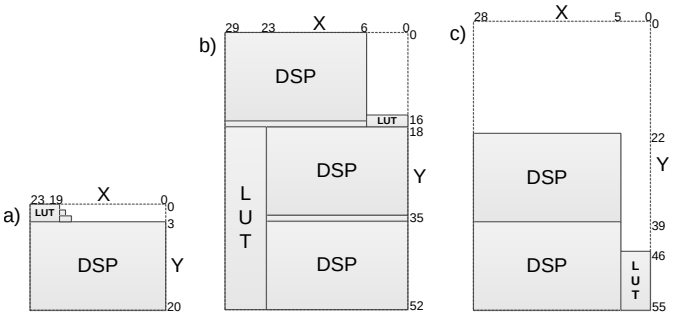

Fig. 8. Truncated tiling for multipliers Mul1(a), Mul3+Mul4(b) and Mul5(c)

Mul1 is a 20x24 multiplier, with a maximum truncation error as shown in Figure 9. As an integer, the sum would be the value 10111111111111111001_b. However, since the 44 bit result of Mul1 is actually a fixed-point number less than one (due to $X_2 < 2^{-9} \wedge C_1 < 1$, details in [5]), the introduced error is quite small and does not lead to a violation of our 1-ULP accuracy requirement (see Section III-B for an error analysis).

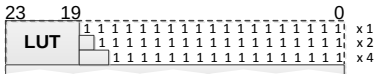

Fig. 9. Maximum error scenario for truncated multiplication in Mul1

Mul3 (and Mul4) are 30x53 wide. In both inputs, all but one bit are fractional bits, leading to a full precision result with 83 bits, of which 81 bits are fractional bits. Furthermore, the highest bit of the result is known to be zero because R_d is known to be less or equal to one. The remaining 82 result bits are rounded to 56 (57, respectively) bits so the last 26 (25) bits are dropped after rounding. A 6x16 bit wide part of the computation is truncated, with a maximum truncation error of

$$(2^6 - 1) \cdot (2^{16} - 1) < 2^{22}.$$

To compare the errors introduced by rounding and truncated multiplication, the maximum truncation error (after the compensation addition described above) is 2^4 (2^3, respectively) times *smaller* than the error due to rounding. Also, note that for Mul4, the large LUT multiplier shown in Figure 8(b) is partially removed by logic optimization, since the higher 28 bits of the subsequent computation of $1 - t$ (in Cycle 7 of Figure 5) are not used.

Mul5 has been changed from 56x28 to 56x29 bits due to the modification described above. However, the 29 bit input was originally 57 bits wide, but had the first 28 bits removed (see discussion above). The result could therefore be regarded

as a $57 + 56 = 113$ bit wide value, of which 111 bits are fractional bits. However, the final result of the mantissa result just requires the 53 fractional bits as defined by IEEE 754 double precision (in the worst case, if $Y/X < 1$). Thus, rounding for this final computation can remove $111 - 53 = 58$ bits. Truncation is performed on Mul5 as shown in Figure 8(c) removing large parts of the computation. The error introduced by the truncation shown is less than 2^{52}:

$$(2^{29} - 1) \cdot (2^{22} - 1) + (2^5 - 1) \cdot (2^{24} - 1) \cdot 2^{22} < 2^{52}$$

After the compensation addition, the error due to truncated multiplication is 2^6 times smaller than the maximum error due to the rounding of the final result.

Figure 10 shows the complete division unit optimized using these measures, named **PolyGSopt**. Its latency is reduced to 8 clock cycles at 200 MHz operation frequency. Of the 20 DSP blocks used in the first implementation, only 11 are remaining here.

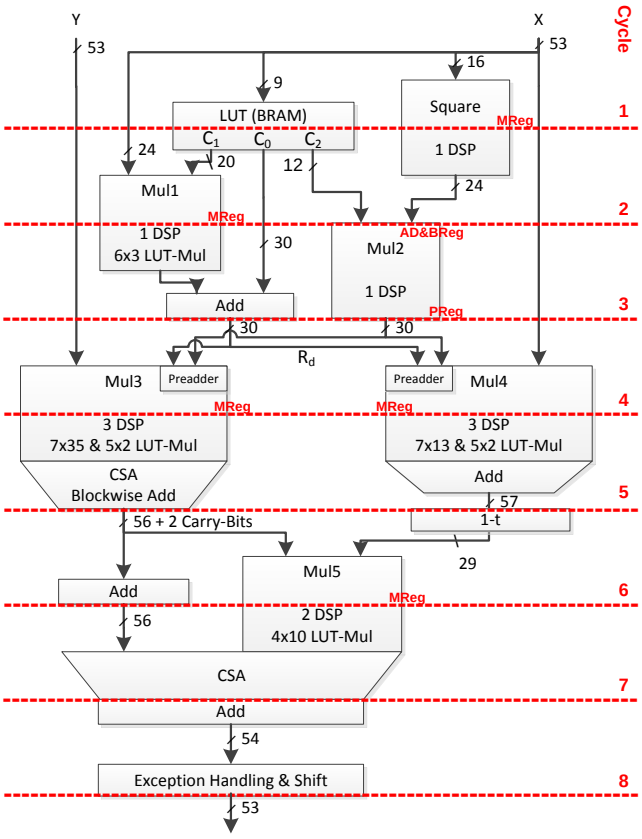

Fig. 10. PolyGSopt implementation with reduced latency and area (8 cycles, 11 DSPs)

B. Error Analysis

As the use of truncated multipliers increases the maximum error, it is necessary to perform a new error analysis for the **PolyGSopt** implementation. A maximum error of 1-ULP in a double-precision floating-point mantissa implies an upper bound of 2^{-52}. The result of the mantissa division will be between 0.5 and 2, requiring a normalization shift for results smaller than 1. Here, ϵ_Z must be smaller or equal to 2^{-53} to ensure an 1-ULP accurate final result.

The overall maximum error ϵ_Z is composed of the error of the method (minimax approximation and Goldschmidt iteration), and the error introduced by using finite precision arithmetic in the computation.

$$\epsilon_Z = \epsilon_{method} + \epsilon_{comp}.$$

According Pineiro et al. [5], the error of the method applying a single Goldschmidt iteration is

$$\epsilon_{method} = X \cdot Y \cdot \epsilon_{Rd}^2.$$

This also applies to the modified Goldschmidt method. The computation error ϵ_{comp} is calculated as

$$\epsilon_{comp} = \epsilon_{M3} + Y \cdot Rd \cdot \epsilon_{M4} + Y \cdot \epsilon_{M4} \cdot \epsilon_{R_d}$$
$$+X \cdot \epsilon_{M3}\epsilon_{R_d} + \epsilon_{M3} \cdot \epsilon_{M4} + \epsilon_{M5Z}$$

with ϵ_{M5Z} denoting the error introduced by final rounding to the 54 bit result and truncated multiplication in Mul5 and ϵ_{Mx} denoting the error introduced by rounding and truncated multiplication in multiplier x.

For the mantissa division, there are 2^{52+52} possible inputs, so a complete simulation of all possible inputs is generally not practical. However, the computation of R_d only depends on X_1 and X_2, resulting in only 2^{9+24} possible combinations. The maximum error ϵ_{R_d} can therefore be determined experimentally. The computed value of R_d must be compared to the exact value $1/X$ with $X = \{X_1, X_2, X_3\}$. For the comparison, X_3 must be set both to all **1**s and all **0**s to maximize the error. As the maximum error decreases with growing X, the value of $X \cdot \epsilon_{R_d}^2$ was also determined experimentally. This resulted in the following upper bounds for the 8-cycle division unit shown in Figure 10:

$$\epsilon_{Rd} \leq 2^{-28.969}$$
$$X \cdot \epsilon_{Rd} \leq 2^{-28.331}$$
$$X \cdot \epsilon_{Rd}^2 \leq 2^{-57.644}$$

With $Y < 2$, an upper bound for the method error can be computed:

$$\epsilon_{method} \leq 2^{-56.644}$$

The computation error at each multiplier consists of the rounding error and the (compensated) error of the truncated multiplication. The result of Mul3 is 56 bits wide, of which 55 bits are fractional bits. Rounding to nearest therefore causes an maximum error of 2^{-56}. Furthermore, the error introduced by truncated multiplication was shown to be 2^4 times smaller.

$$\epsilon_{M3} \leq 2^{-56} + 2^{-60}$$

The result of Mul4 is 57 bits wide, of which 56 bits are fractional bits. However, the leading 28 bits are not used because they are known to be all zero after subtraction. Rounding to nearest causes a maximum error of 2^{-57}, while the error introduced by truncated multiplication is the same as at Mul3.

$$\epsilon_{M4} \leq 2^{-57} + 2^{-60}$$

The result of Mul5 is passed to the final addition without rounding. But after the addition, it is rounded to 54 bits in worst case (if the result is smaller than 1). 53 of these 54 bits are fractional bits, so the error of rounding to nearest is 2^{-54}.

Furthermore, the truncated multiplication error is known to be 2^6 times smaller:

$$\epsilon_{M5Z} \leq 2^{-54} + 2^{-60}$$

Substituting these inequalities into the inequality for ϵ_{comp} results in

$$\begin{aligned}\epsilon_{comp} \leq \quad & 2^{-56} + 2^{-60} + 2 \cdot (2^{-57} + 2^{-60}) \\ + \quad & 2 \cdot (2^{-57} + 2^{-60}) \cdot 2^{-28.969} \\ + \quad & 2 \cdot (2^{-56} + 2^{-60}) \cdot 2^{-28.969} \\ + \quad & (2^{-56} + 2^{-60}) \cdot (2^{-57} + 2^{-60}) \\ + \quad & 2^{-54} + 2^{-60}\end{aligned}$$

$$\epsilon_{comp} \leq \quad 2^{-54} + 2^{-55} + 2^{-58} + 2^{-83.269}$$

Further substitution into the inequality for ϵ_Z proves, that the divider is actually accurate within 1-ULP:

$$\epsilon_Z \leq 2^{-54} + 2^{-55} + 2^{-56.644} + 2^{-58} + 2^{-83.269} < 2^{-53}$$

IV. Experimental Results

A. Post-Place&Route Performance

The division units presented in this paper have been successfully tested using randomized mantissas. The results were compared to a 75 bit Xilinx IP Core divider and did not violate the 1-ULP error constraint.

Table II shows the synthesis results for the division units presented for a Virtex-6 device, specifically a XC6VLX240T-1. Our optimized implementation of the polynomial approximation followed by a single Goldschmidt iteration clearly outperforms all other division units. The wall clock time of a single division is reduced by 62% compared to the Xilinx IP Core and by 58% compared to the VFLOAT division unit.

For comparison, the divider presented in [10] is also included in Table II, although it was not implemented for Virtex 6 but for Altera Stratix V. However it also uses polynomial approximation followed by a single Newton-Raphson iteration. Its similar structure thus makes it an interesting competitor. It also reaches much higher clock frequencies than the prior work on Virtex-6 dividers, even though the Virtex-6 and Stratix V generally perform comparably in practice for optimized designs. Our implementation manages to outperform even that very fast unit, reaching a 40% shorter wall clock time.

For completeness, we also present data when mapping our implementation to Virtex-7 XC7VX690T-2 and Virtex-5 XC5VFX200T-1 devices. Note that the Virtex-5 lacks the DSP pre-adder capability and can thus only support the 10-cycle PolyGS, but not the 8-cycle PolyGSopt implementation.

B. Impact on High-Level Synthesis

Our work on high-performance division was motivated by research into high-level synthesis tools for the automatic generation of custom convex solver accelerators on FPGAs. We use CVXGEN [14] to generate the actual solver as behavioral C, which is then processed by our Nymble hardware/software co-compiler [15], generating synthesizable Verilog and hardware/-software interface code. Nymble exploits domain knowledge of the nature of the solver code structure to perform a number of optimizations not available to general high-level synthesis tools.

As target platform, we employ a Xilinx ML507 board (Virtex-5 FX-based), using the hardware and software environment described in [16] to achieve high-throughput low-latency access to shared memory between the accelerator(s) and the general-purpose PowerPC 440 processor. As the XC5VFX70T device on the actual board is too small to hold the complete system-on-chip (processor buses, memory controller, network interface, etc.) for the larger solver examples, we also employ a simulated version of the board that virtually substitutes the larger XC5VFX200T device into the same architecture. For our measurements, we performed post-layout simulations, including cycle-accurate models for memories and caches. All Nymble-family compilers currently use LLVM 3.3 as front-end and for machine-independent optimization. Furthermore, we relied on Synopsys Synplify-I-2014.03-1 for logic synthesis, and Xilinx ISE 14.7 for physical mapping.

Four example solvers provided on the CVXGEN website were used as benchmarks (with the specified parameters): SQP ($m = 3, n = 10$), Lasso ($m = 100, n = 10$), SVM ($N = 20, n = 4$), Portfolio ($n = 25, m = 5$). In addition, we translate Stan5P, a significantly more complex example containing a complete model-predictive control problem for collision avoidance in autonomous ground vehicles. Note that most of these solvers actually require double-precision to achieve convergence.

Table III shows the floating-point operators used for the comparison. As a baseline, we employed only Xilinx Core-Gen units, which perform IEEE754 Conforming Rounding (CR). The cores were configured for the lowest latency that still allowed operation with 200 MHz on our target FPGA (XC5VFX200T-1). We then replace the Xilinx divider with a 10-cycle **PolyGS** core using faithful rounding (FR). Note that even higher performance is possible by also substituting the Xilinx multiplier core with a truncated mantissa multiplier as described in [12], but this lies outside the scope of this paper.

	Baseline	**Optimized**
Mul	6 cycles, 232 MHz, CR	
Add	3 cycles, 222 MHz, CR	
Div	57 cycles, 248 MHz, CR	10 cycles, 210 MHz, FR

TABLE III. Floating Point Units Used

Table IV shows the results of the high-level synthesis, using the performance of a placed-and-routed design. Wall-clock speed-ups of more than 2x are achievable using the new floating-point operators. Note that the division units are not the bottleneck for the clock frequencies here. Instead, large routing delays, caused by high fan-in of the instantiated floating point units (due to resource sharing) have been observed as the limiting factor.

	CR Division			**FR Division**			**Speed -up**
	Cycles	f_{max} **[MHz]**	**Time [ms]**	**Cycles**	f_{max} **[MHz]**	**Time [ms]**	
SQP	65313	91.1	0.72	33885	91.1	0.37	1.93
Lasso	49821	87.9	0.57	26769	91.2	0.29	1.93
SVM	95314	66.8	1.43	47548	83.5	0.57	2.51
Portfolio	127468	66.7	1.91	68907	80.2	0.86	2.22
Stan5P	1223450	65.9	18.56	678733	71.2	9.54	1.95

TABLE IV. Comparison of solver accelerators using CR and FR floating point operations

V. Conclusion and Future Work

Two low-latency 1-ULP accurate division units have been presented and compared. The approach using polyno-

Method	Acc.	Device	Latency [cycles]	Max Freq. [MHz]	Latency [ns]	Resources
IP Core from Xilinx [9]	CR	Virtex-6	20	192	104	3216 LUTs, 2035 Reg., 0 BRAM, 0 DSP
VFLOAT [8] (results as reported in [9])	1 ULP	Virtex-6	14	148	95	6957 LUTs, 934 Reg., 0 BRAM, 0 DSP
FloPoCo 2.5.0 FPDiv	CR	Virtex-6	17	136	125	4419 LUTs, 2509 Reg, 0 BRAM, 0 DSP
TripleGS	1 ULP	Virtex-6	12	201	60	1474 LUTs, 1294 Reg., 1 BRAM18K, 30 DSP
PolyGS	1 ULP	Virtex-6	10	230	44	1297 LUTs, 1244 Reg., 1 BRAM36K, 20 DSP
PolyGSopt	1 ULP	Virtex-6	8	202	40	1525 LUTs, 1094 Reg., 1 BRAM36K, 11 DSP
Polynomial Approx (d=2) + Newton-Raphson [10]	1 ULP	Stratix V	18	268	67	887 ALUTs, 823 Reg., 2 M20K, 9 DSP
FloPoCo Radix-4 [10]	CR	Stratix V	36	219	164	5209 ALUTs, 5473 Reg, 0 M20K, 0 DSP
PolyGS	1 ULP	Virtex-5	10	210	48	1251 LUTs, 1244 Reg., 1 BRAM36K, 20 DSP
PolyGSopt (speed grade -2)	1 ULP	Virtex-7	8	260	31	1853 LUTs, 1094 Reg., 1 BRAM36K, 11 DSP

TABLE II. PERFORMANCE OF PLACED&ROUTED DIVISION UNITS

mial approximation of second degree followed by a single Goldschmidt-iteration was shown to deliver superior performance in terms of latency and area when compared to the TripleGS approach. This superior design was then further optimized using truncated multipliers. To our knowledge, the resulting 8-cycle division unit is the lowest latency 1-ULP accurate division unit available for Virtex-6/-7 architecture that also reaches at least a 200 MHz clock frequency.

The immediate relevance of these improvements was demonstrated by integrating the divider into a high-level synthesis system for automatic generation of convex solver accelerators, leading to application-level speed-ups of more than 2x without adversely affecting numerical stability.

Future work could address the automatic generation of dividers for different word widths, using the truncation and tiling algorithms proposed in [12] to find optimal solutions. In addition, the design was optimized for a 200 MHz target frequency. Deeper pipelining should be possible and might allow much higher frequencies.

REFERENCES

[1] IEEE, "IEEE Standard for Floating-Point Arithmetic," pp. 1–70, 2008.

[2] Xilinx, "Xilinx LogiCORE IP Floating-Point Operator v5.0 Product Specification," Tech. Rep., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf

[3] R. E. Goldschmidt, "Applications of division by convergence," Ph.D. dissertation, Massachusetts Institute of Technology, 1964.

[4] K.-N. Han, A. F. Tenca, and D. Tran, "High-speed floating-point divider with reduced area," in *SPIE Optical Engineering + Applications*, M. S. Schmalz, G. X. Ritter, J. Barrera, J. T. Astola, and F. T. Luk, Eds. International Society for Optics and Photonics, Aug. 2009, pp. 74 440O–74 440O–8. [Online]. Available: http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=786791

[5] J.-A. Pineiro and J. Bruguera, "High-speed double-precision computation of reciprocal, division, square root, and inverse square root," *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1377–1388, Dec. 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=626534.627261

[6] I. Koren and O. Zinaty, "Evaluating elementary functions in a numerical coprocessor based on rational approximations," *IEEE Transactions on Computers*, vol. 39, no. 8, pp. 1030–1037, 1990. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=57042

[7] M. Schulte and J. Stine, "Symmetric bipartite tables for accurate function approximation," in *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*. IEEE Comput. Soc, 1997, pp. 175–183. [Online]. Available: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=614893

[8] J. Detrey and F. Dinechin, "A Tool for Unbiased Comparison between Logarithmic and Floating-point Arithmetic," *The Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 49, no. 1, pp. 161–175, May 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1295911.1295927

[9] X. Fang and M. Leeser, "Vendor agnostic, high performance, double precision Floating Point division for FPGAs," in *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Sep. 2013, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6670335

[10] B. Pasca, "Correctly rounded floating-point division for DSP-enabled FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, Aug. 2012, pp. 249–254. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6339189

[11] M. Schulte and E. Swartzlander, "Truncated multiplication with correction constant [for DSP]," in *Proceedings of IEEE Workshop on VLSI Signal Processing*. IEEE, 1993, pp. 388–396. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=404467

[12] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 4, p. 73, Jan. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1926367.1926380

[13] P. Markstein, "Ia-64 and elementary functions: Speed and precision, ser," *Hewlett-Packard Professional Books. Prentice-Hall, Englewood Cliffs, NJ*, 2000.

[14] J. Mattingley and S. Boyd, "CVXGEN: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, Nov. 2012. [Online]. Available: http://www.springerlink.com/index/10.1007/s11081-011-9176-9

[15] J. Huthmann, B. Liebig, J. Oppermann, and A. Koch, "Hardware/software co-compilation with the Nymble system," in *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, Jul. 2013, pp. 1–8. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6581538

[16] H. Lange and A. Koch, "Architectures and execution models for hardware/software compilation and their system-level realization," *IEEE Trans. Comput.*, vol. 59, no. 10, pp. 1363–1377, Oct. 2010. [Online]. Available: http://dx.doi.org/10.1109/TC.2009.180