Architecture Exploration of High-Performance Floating-Point Fused Multiply-Add Units and their Automatic Use in High-Level Synthesis

Björn Liebig, Jens Huthmann and Andreas Koch
Technische Universität Darmstadt
Embedded Systems and Applications Group
Email: {liebig,huthmann,koch}@esa.cs.tu-darmstadt.de

***Abstract*—Multiply-add operations form a crucial part of many digital signal processing and control engineering applications. Since their performance is crucial for the application-level speed-up, it is worthwhile to explore a wide spectrum of implementations alternatives, trading increased area/energy usage to speed-up units on the critical path of the computation.**

This paper examines existing solutions and proposes two new architectures for floating-point fused multiply-adds, and also considers the impact of different in-fabric features of recent FPGA architectures. The units rely on different degrees of carry-save arithmetic improve performance by up to 2.5x over the closest state-of-the-art competitor.

They are evaluated at the application level by modifying an existing high-level synthesis system to automatically insert the new units for computations on the critical path of three different convex solvers.

***Keywords*-FMA; fused; FPGA; multiply-add; carry save; floating-point;**

I. Introduction

Many signal processing and control engineering applications have large numbers of floating-point multiply-add operations at their core. When considering the use of reconfigurable compute units (RCU) to speed-up these algorithms, the implementation of fast multiply-add units often becomes crucial.

Orthogonal to the performance of individual units is the system-level performance vs. area vs. energy balance. To make system-level evaluations practical, we also have to consider the *automatic* use of the new units by system-level design tools, such as high-level language to hardware compilers [1]. In general, realizing all required multiply-add (MA) operations by very fast (low latency, high throughput) implementations is not efficient, as the area (and possibly energy overhead) can quickly become prohibitive. It is thus worthwhile to employ strategies that only employ the fast MA units on the critical path.

In this work, we will examine both topics in context of the highly relevant field of hardware acceleration of general solvers for convex optimization problems. Such solvers are used in systems relying on model-based/model-predictive control rules, which achieve much higher quality than simple proportional-integral-differential (PID) controllers. Specifically, we are using a tool-flow that accepts high-level descriptions of convex optimization problems in the CVXGEN language [2] and automatically generates a hardware-implementation of the specific solver. As concrete benchmarks for the system-level speed-up of the new MA units and the new compiler pass, we will consider three solvers of increasing complexity for trajectory planning during collision avoidance of autonomous ground vehicles.

A. Nature of convex solver computations

The solver computations have a high degree of instruction-level parallelism, but have also long chains of data-dependent operations (see example in Listing 1).

```
x[1] = a*b + c*d;
x[2] = e*f + g*x[1];
x[3] = h*i + k*x[2];
```

Listing 1: Solver computation structure

These dependency chains form (potentially long) path through the algorithm's control data flow graph (CDFG), shown in Fig. 1 for the previous example, with the critical path is marked by bold red edges. Reducing the computation latency on this path is crucial for achieving high application-levle speed-ups.

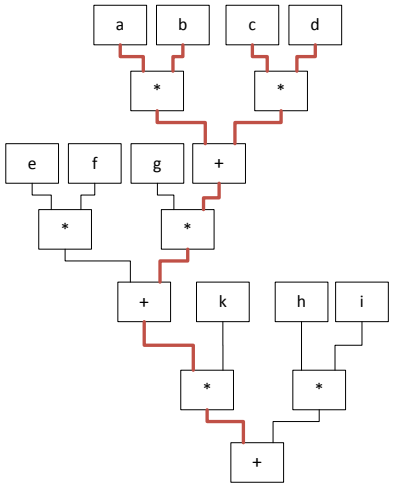


Figure 1: Critical path of code in Listing 1

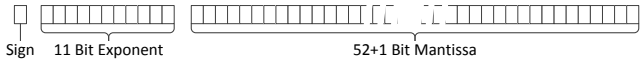


Figure 2: IEEE 754 Double-precision format

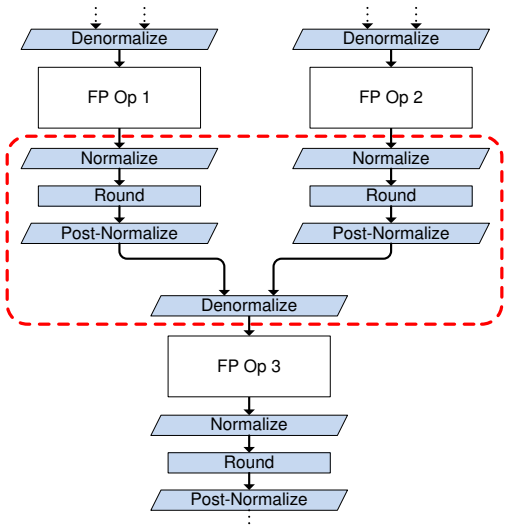


Figure 3: Normalization and denormalization between floating-point operations

B. Conventional floating-point representation

The IEEE standard 754 [3] defines the floating-point representations currently in widespread use.

A finite number R is represented in this format by the three components named mantissa (M), exponent (E) and sign (S), s.t. $R = M * 2^{E-b} * (-1)^S$ where the bias b is a positive integer. The standard defines a number of basic formats with specified widths of the M and E fields as well as the bias values.

As an example, Figure 2 shows the structure of the widely used binary64 format, more commonly known as *double-precision*.

The formats specified by the standard also ensure unique representations of each number, thus avoiding the ambiguity arising, e.g., from $1.5 * 2^3 = 0.75 * 2^4$. This is achieved by scaling the mantissa s.t. 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**). Another exception are numbers with a very small magnitude (having zero as exponent). These so-called *subnormals* do not have an implied **1** as MSB.

This scaling process is called *normalization*. It has to be performed after every computation for the result to be in valid IEEE 754 number representation. For high-performance computation, it can be worthwhile avoid normalization after every step, instead allowing the computations to be *fused* together and perform the normalization only at the end of the fused region (see Fig. 3).

This technique has often been used to increase the performance of MA operations, combining them into *fused MA* (FMA) operations. In this manner, the steps "normalization", "rounding", "post-normalization", and in some cases also "denormalization," can be avoided. Inside of these fused operators, non-standard floating-point formats can be used, generally allowing improved area / latency tradeoffs and a better match to the target technology of the specific implementation. Furthermore, if required, the intermediate results can also be represented in formats providing greater accuracy than the standard formats.

C. Contributions and structure

In this work, we improve upon the prior art by not only avoiding normalization between the *internal* addition and multiplication subcomputations of FMA operators, but also selectively, using high-level synthesis, between multiplication and addition *across* an entire chain of MA operations in a critical path of the CDFG.

Section 2 gives a brief overview of related work. Our own contributions will be presented in Section 3, specifically: A partial carry-save (PCS) number representation suitable for mapping to FPGAs, a fast FMA unit based on the PCS representation, an even faster FMA unit relying full carry-save (FCS) representation and exploiting features of recent FPGA architectures for area efficiency, and a high-level synthesis compiler pass for integrating the FMA units and the required non-standard $\leftrightarrow$ IEEE 754 data type conversions into scheduled CDFGs. Section 4 experimentally evaluates out approach by comparing it to current academic and industrial state-of-the-art implementations. Section 5 draws conclusion and looks out towards further work.

II. Related work

The multiply-add fused unit, which was later referred to as fused multiply-add (FMA), was first proposed in 1990 [4]. More recent works introduce improved FMA architectures, but often target stand-alone ASICs or units integrated into CPU pipelines. Thus, they use IEEE 754-conforming representations for all input operands as well as the result [5, 6, 7]. [8] gives a survey of the wide spectrum of FMA architectures developed from 1990 to 2007.

The principle of fused operators has also been applied to other computations, such as fused dot products [9, 10], again having standard-conforming interfaces.

The application-specific use of non-standard formats for improved numerical accuracy has been proposed for FPGAs, e.g., in [11]. The use of non-standard formats to improve performance is presented in [12] for the use of a multiply-accumulate (MAC) unit. It uses a PCS representation to achieve low latency at the addition stage but relies on application-specific knowledge of the input and output value ranges. Implementations of Radix 4 and 16 exponents

showed improved addition speed but slower multiplication [13].

Many existing floating-point libraries for FPGAs omit subnormals (which only marginally extend the representable number range) to improve performance [14, 15], an approach we will also follow. A detailed survey of the fundamentals of floating-point operations on FPGAs is given in [16]. To our knowledge, we are the first to use heterogeneous input formats (optimized mix of carry-save and IEEE 754 compliant operands) for FMA units.

In contrast to the publications discussed above, others focus on the assembly of complete datapaths from individual operators. FloPoCo [17] exploits a language mixing features from VHDL and C++ to describe pipelines of floating-point operators. However, it does not automatically perform operator fusion. Langhammer et al. developed a floating-point datapath compiler which can generate fused floating-point operations from a subset of C. The generated datapaths have standard IEEE 754 inputs and outputs [18, 19]. Our approach extends these prior works by the selective use of (partial) carry save number formats and by the integration in a C-to-HDL Compiler.

Carry Save Adders (CSA) have long been used for fast constant-time addition [20], especially inside multiplication units. Their carry save (CS) number format departs from conventional binary format by allowing the values **0, 1, 2** for each digit, but encodes this in a binary representation. The CS format, however, has to deal with non-unique representations for numbers, complicating, e.g., comparison operations. Please see Section III-E for a discussion of some of these details.

Automatic inference of CS arithmetic in synthesis has also been subject to prior research [21]. However, it has focused on the general synthesis of CS structures, not their selective use to accelerate floating-point operations. Other approaches use CS arithmetic internally to individual operations, but not between them [12, 22].

III. Fast Multiply-Add Units for Critical Path Acceleration

Entire *chains* of MA operations are typical for the solver datapaths we want to compile. For reducing the application-level latency, we need to reduce the latency through the complete FMA unit, starting at the multiplier input and ending at the adder result. This eliminates the MAC unit proposed in [12] from consideration, as it only exploits low latency addition. However, the idea of a mantissa in PCS format, which we exploit in our FMA designs, originates in that work.

In the following sections, we develop two FMA units calculating $R = A + B * C$ using CS representations: One using PCS, portable to older FPGAs (e.g., Xilinx Virtex-5), and one using FCS, exploiting special capabilities of recent FPGA generations (e.g., Xilinx Virtex-6 and later).

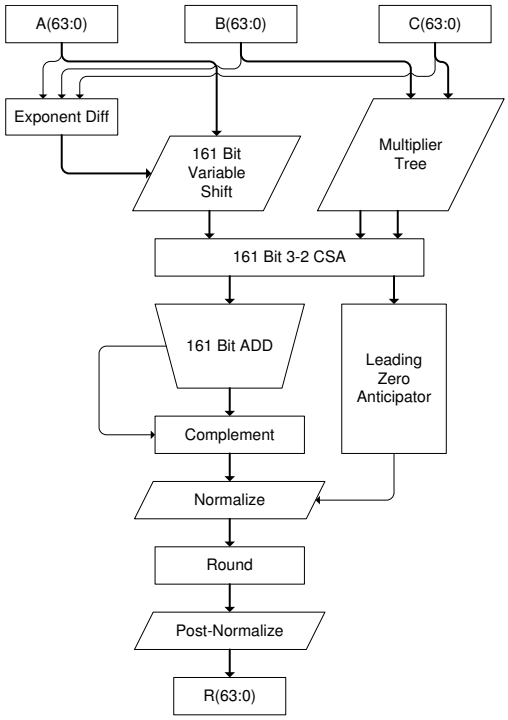


Figure 4: Classic FMA architecture [4] with IEEE 754-compliant operands and result

For brevity, we will be using $\{C, A, B\}_M$ to denote the mantissas of C, A, and B, respectively. Even though our architectures are freely parametrizable, we will examine double-precision operators matching or exceeding IEEE 754 accuracy here for comparison with prior solutions.

A. Reducing normalization latency

Since one of the major means of latency reduction in this work is the avoidance of unnecessary normalization steps, we begin the exploration by considering a classic FMA design [4] following this approach. This architecture, shown in Fig. 4, is used as a baseline for our own optimizations.

Adder and multiplier are fused into a single operation, without an intervening normalization step. The multiplier result is instead provided in CS format (please see Section III-E for an introduction to the CS representation). Furthermore, the performance of the adder is improved by performing the pre-shifting of the additive input A in parallel with the multiplication $B * C$.

Since the output of the classic FMA unit is in IEEE 754 format, the internal CS representation has to be converted to that plain binary format. This is achieved by a 161b adder followed by a conditional complement block to handle negative numbers. The actual normalization (left-shifting to achieve the implied **1**) is guided by a Leading Zero Anticipator (LZA) [23], which computes the shift-distance in parallel with the addition. Rounding to the required

precision, followed by a conditional one-bit right shift for post-normalization (to compensate for rounding overflow), is performed at the end.

B. Speeding-up post-normalization

Even in its original form (normalization only after the adder), the classic architecture has potential for improvement by just slightly deviating from IEEE 754 (still using binary format, but with modified field widths): By adding an extra bit at the most significant side of the mantissa, we can safely skip the post-normalization right shift at the end. Actually, this requires the use of two additional bits in the custom representation of the mantissa (now 54b) , as the leading **1** can no longer be just implied. In practice, if targeting FPGAs with embedded DSP48E blocks (such as the Xilinx Virtex-5, -6, and -7 devices), the slight widening of the internal computation (from 53b to 54b, both including the leading **1**) does not require additional DSP blocks. Furthermore, in our approach of selectively employing custom number formats just on the critical path, only the C input (which is the output of the previous FMA unit) needs to be widened. B can remain in standard format, as there is sufficient time for its proper post-normalization.

Orthogonal to these optimizations is the integration of IEEE 754 exception encoding. As already shown in FloPoCo [14], this can avoided by using two additional wires for explicitly signalling exceptions instead of encoding them in the number representation. We will apply the same technique.

C. More efficient rounding

It is tempting to eliminate the rounding step entirely. However, while truncation may be acceptable for some applications, others will suffer from the increased rounding error, which is the case for our solver accelerators. But by considering entire chains of FMA units during datapath assembly in high-level synthesis, we can move the rounding step from the output of an FMA unit through the C input into the succeeding unit. While this does not directly improve the latency, it allows the integration of the rounding for C into the CSA tree of the multiplier (Fig. 5), adding at most one logic level to the critical path.

As can be seen in Fig. 1, at most the second operand C of the multiplier and the first operand A of the adder are performance-critical and thus need to use the custom number format. We now add two rounding units: A dedicated one for A (running in parallel to the pre-shift distance computation), and a second one for C (integrated into the multiplier CSA tree). The second unit is on the critical path, however. To allow its execution in parallel with the multiplication, we perform the actual multiplication with the *unrounded* value of C_M and then correct an erroneous result afterwards by adding B_M to the product if rounding would have increased C_M by one (Fig. 6).

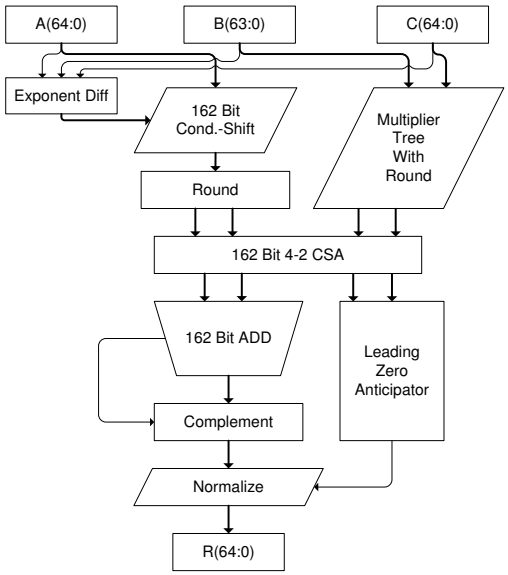


Figure 5: Modified FMA with rounding moved into the succeeding operation

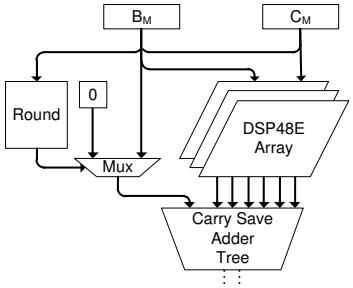


Figure 6: Internal structure of mantissa multiplier with integrated rounding unit

The increase of the mantissa width between operators depends on the rounding mode. For the case "Round half away from zero", only a single additional bit is required (as shown in Figure 5). Thus, A, C, and R are basically in IEEE 754 format, but with an extra bit of mantissa to transfer the original *unrounded numbers* between operators, leading to 65b operands and results. However, for other rounding modes, the transfer of the complete, unrounded internal mantissa would be required, which is a potentially expensive operation (162b in the example).

D. Eliminating the variable-distance shift

The final step of normalization is the variable-distance shifter: The number of leading zeros after the addition of two signed numbers can be anything from zero to mantissa bit width plus one. The shifter thus must support distances from zero to the full width, which makes the MSB of the result depend on *every single bit* of its input, that being 162b wide in the FMA unit. Obviously, a major improvement in latency

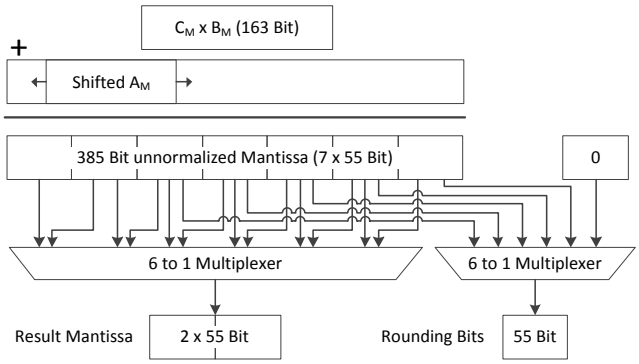


Figure 7: Replacing shifting by a 6-to-1 multiplexer

could achieved if this potentially very slow step could be eliminated.

To simplify the final shift, we propose replacing it with a multiplexer, which is actually doing a shift in larger blocks of bits. To determine the block size, we consider the requirements on the result and then work backwards toward the width of the adder: In our result, we want to achieve at least the accuracy of IEEE 754 double precision format with its 52b mantissa. Since we now explicitly represent the leading **1**, we need one more bit. Similarly, since we no longer use an explicit sign bit but two's complement notation, we need an extra bit in the mantissa. Finally, we have to add a guard bit to catch a possible overflow in the mantissa[1]. This yields a total width of 55b, we thus convert the addition result (whose width we derive later in this subsection) into blocks of 55b.

Since the number of leading zeros in the non-normalized result is unknown and generally not a multiple of 55b, the first non-zero digit could be positioned anywhere in the result. When shifting by multiples of 55b, the result mantissa must thus be composed of at least two 55b blocks, making it 110b wide in total (see Fig. 7).

After determining the result mantissa width to be 110b, we have to consider the impact of this decision on the input and internal widths of the succeeding FMA units. For the first, we now have to increase the width of our critical A and C inputs to accommodate a 110b mantissa, while B can remain in IEEE 754 format (52b mantissa plus implied leading **1**). The latter is highly beneficial, since the *number of inputs* to the multiplier CSA tree depends on the width of the smaller operand (that being B_M). On the other hand, the *widths* of the multiplication and addition stages grows significantly: The multiplier now has a (52+1)b wide multiplicand B_M and a 110b wide multiplicator C_M, yielding a total of 163b. The adder stage grows from 162b to 385b, since, for large exponent differences, the 110b wide addend A_M must be alignable even completely left or completely right of the

[1]The reason for the possible overflow in the CS format is discussed in Sec. III-E.

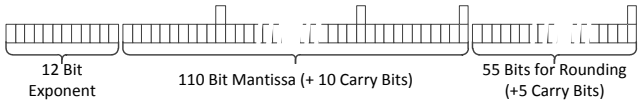


Figure 8: Complete floating-point format with PCS mantissa

product $C_M * B_M$. This yields 110b+163b+110b = 383b, rounded up to 385b, the next multiple of 55b as described in Sec. III-E. The entire multiply/shift/add/mux structure is shown in Fig. 7.

Looking at these choices from the circuit performance view, we see that the multiplier latency should be unchanged, since the height of its CSA tree depends on the *number* of inputs, which has remained constant. However, the increased *width* of the operands has a detrimental effect on adder performance: On a Xilinx Virtex 6 FPGA (speed grade -1), the register-to-register latency of even of single 385b adder is about 8.95ns, which is far too slow for our performance requirements. Hence, the increase in bit width due to the elimination of the variable-distance shifter can no longer be handled using plain binary format addition. Instead, we break the excessive carry chains by explicitly representing carries of smaller addition widths. This leads to a major shift away from the variations of the IEEE 754 format we have been using so far (mostly with different mantissa widths) towards a CS representation of mantissas in floating-point numbers.

E. Floating-point representation using a PCS mantissa

At first glance, an FCS representation using 110 carry bits in addition to the 110 binary mantissa bits is not feasible, since it would again double the size of the multiplier. However, the latency of the addition can already be improved by employing just a limited number of explicit carry bits in the mantissa representation. Such a PCS approach has already been demonstrated to be efficient for FPGA implementation [12].

Two constraints need to be considered for optimal carry bit distribution: To simplify the multiplexing step, the carry bits should be equally distributed in every 55b mantissa block. To allow a regular design of the operator, the distance between all carry bits should be equal. Combined, these two constraints allow the insertion of a carry bit only for every 5th, 11th or 55th bit of mantissa. When evaluating these alternatives, we discovered that the delay difference between a 5b and an 11b adder is so small (1.650ns vs. 1.742ns) that we can choose the more area efficient 11b distribution without a significant performance penalty. In this fashion, we reduce the internal FCS widths of a 385b wide sum and 384b of carries to the PCS format of 385b sum and 35b of carries (shown as Carry Reduction in Fig. 9). Using the same distribution for our CS inputs and the result, the prior 110b two's complement binary format for the mantissa (derived in Sec. III-D to match the accuracy of IEEE 754

double precision) is extended with 10b of carries into a PCS format.

However, rounding becomes more complicated, as CS does not guarantee unique representations for numbers: The plain binary representation for the value of 0.5d (decimal) is always 0.1000b (binary). However, when a CS format is used, each digit can take the values $\{\mathbf{0}, \mathbf{1}, \mathbf{2}\}$. The decimal value 0.5d could thus be represented in CS format as 0.0200cs *or* 0.0120cs. Even if the most-significant fractional digit is zero, values larger than 0.5d (which would need to be rounded up) can be represented in CS (e.g., 0.75d as 0.0220cs). Thus, it no longer suffices to examine a single bit to make an exact rounding decision. Instead, all mantissa bits must be considered, even if in rounding mode "round half away from zero" and "round to +infinity".

This would become very expensive for our current 385b addition result, which could (in the worst case) consist of five non-zero 55b blocks. Thus, we make the conscious decision to accept some misrounded numbers by considering only a *narrower part of the mantissa* for rounding: We examine only the *single* 55b block (with 5b of carries) immediately to the right of the 110b result chosen by the 6-1 multiplexer in Fig. 7, which results in a truncation before rounding. With this choice, an erroneous rounding-down would only occur if the saved carries would ripple through all 55b from the LSB to the MSB of the fractional part. In the proposed format, the largest number that would be erroneously rounded down is 0.50000000000000083d. This inaccuracy is acceptable for our use case. If more rounding accuracy is required, a wider part of the mantissa would need to be considered.

F. PCS-FMA Unit

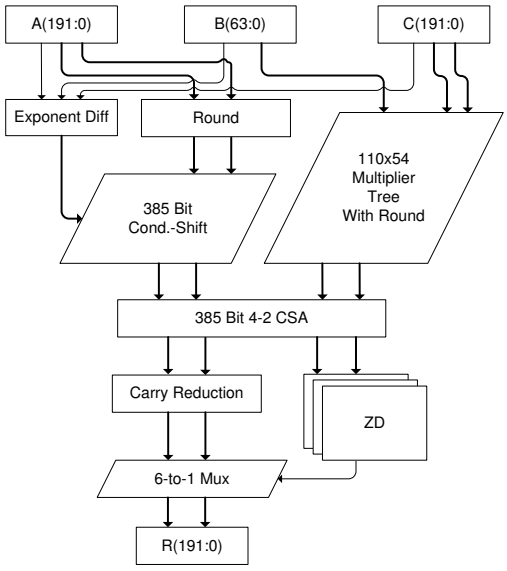


Figure 9: Proposed PCS-FMA architecture

Fig. 9 shows the final PCS-FMA unit. It accepts the non-critical input B in IEEE 754 double precision format, the time-critical inputs A and C are represented as a mantissa in 110b+10b PCS format, combined with 55b+5b of rounding data in PCS format, combined with a 12b exponent in excess-2047 notation. The latter was explicitly chosen to surpass the range of the 11b exponent specified by IEEE 754. In total, the A and C operands, as well as the FMA result, are expressed as 192b words.

We have not yet discussed how we actually compute the select signal of the 6-1 multiplexer in Fig. 7 to choose the most-significant non-zero 55b block(s) as result, as well as the 55b block immediately right of the result for subsequent rounding (Fig. 8). Since we have eliminated the variable-distance shifter commonly used in prior art, we no longer need to identify leading zero bits at *single-bit granularity* using techniques such as Leading Zero Anticipation (LZA [23]). Instead, it suffices to detect and disregard entire 55b *blocks* of leading zeros using a simple Zero Detector (ZD) to identify the block holding the most significant **1**.

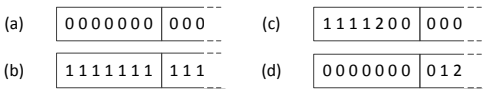


Figure 10: Different forms of leading zeros in two's complement CS representation

The ZD does need to handle some idiosyncrasies of the two's complement CS format we use for the mantissa. Obviously, leading blocks with all **0**s can be skipped (see Fig. 10.a). However, similarly, leading blocks with all **1**s can *also* be skipped: While they indicate a negative number, that same number can be represented with fewer bits as long as the MSB remains **1**. Thus, leading all-**1** blocks can also be skipped (see Fig. 10.b, the leftmost all-**1** block is skipped). Furthermore, a block of **1**s followed by a single **2** followed by **0**s to the end of the block is considered a block with value zero (due to the ripple carry from the **2** upwards) and will also be skipped (see Fig. 10.c). Finally, before actually skipping a leading all-**0** block, we have to be sure that its removal will not alter the value of the succeeding blocks. Fig. 10.d shows an example for this: At first glance, it appears that the leftmost all-**0** block could be skipped. However, when converting the value of the succeeding block from CS into binary, 012...cs = 100...b. Since that block is now the most significant block (the first one got skipped), the **1** in the MSB now indicates a *negative* number, which is incorrect (with the leading all-**0** block, the original value was *positive*). Thus, to avoid these overflows, we skip an all-**0** block *only* if the first two CS digits of the succeeding block are also **0**, avoiding all potential overflows.

While the Carry Reduction step of Sec. III-E is carried out in parallel with ZD, the latter is now critical and determines the total FMA latency.

G. Early leading zero anticipation

We can shorten the critical path further by replacing the ZD units with early leading zero anticipation. We combine our idea of zero-value consideration at block granularity with the prior art of LZA units. For each of the FMA inputs, we use an LZA unit to compute the lower bound for the number of leading zeros in the FMA output. Since B_M is in standard format (having an implied leading **1** in the mantissa), it does not need a dedicated LZA if subnormal numbers are disregarded (as we do here). LZA units are required only for A and C.

Most LZA units are inexact and have an error of up to one bit position. A further bit of uncertainty is introduced by the product $B_M * C_M$, with $1 \leq B_M < 2$. Finally, the sum of the shifted (aligned for different exponents) A_M with the product can potentially require an additional bit, increasing the total error in leading zero anticipation to three bits. To compensate for this maximum error and still exceed double-precision, the result mantissa block size introduced in Section III-D must be increased from 55b to 58b to make sure that in worst case, at least 53 significant mantissa bits are included in the two result blocks selected.

Special consideration must be focused on the issue of adding a product $B*C$ with an addend A that have different signs but a similar magnitude. This will lead to many leading zero blocks in the sum. Potentially, even all of the blocks may be zero if the two addends cancel each other out completely. In these cases of mantissas with very small magnitude, the anticipation error of the LZA-based approach leads to a larger relative inaccuracy compared to the precise (but slower) ZD-based approach described in Sec. III-F. However, since we have already taken the maximum LZA error into account by widening the mantissa, we ensure that even in these extreme cases, we will never be more inaccurate than IEEE 754 double precision.

Also, the early leading zero anticipation logic must *reliably* detect all-**0** input mantissas. Otherwise, the result block multiplexer could erroneously select leading all-**0** blocks for the result, even though a **1** (that should actually be in the leading block) is present in the less significant bits of the sum.

H. FCS-FMA for FPGAs with DSP pre-adders

The improvements described in Sec. III-G remove the ZD operation from the critical path. However, now the Carry Reduce step (Fig. 9) becomes critical. For FPGAs featuring fast pre-adder stages in their DSP blocks, even this step can be completely eliminated, but its removal still incurs a significant complexity cost.

In contrast to the Xilinx Virtex-5 family, the more recent Virtex-6 and -7 devices provide DSP48E**1** blocks that implement a 25b pre-adder on one of their inputs. The pre-adder can be used for C_M to add two 23b blocks of CS partial sum and carry bits, converting them to plain binary format, without the risk of a sign-changing overflow. The most significant block of C_M can actually be processed at the full pre-adder width of 25b, as it is a signed number itself.

The pre-adders allow the representation of A and C in *full carry save* representation, thus eliminating the Carry Reduce step. However, such a space-intensive format begins to tax the resources even of recent FPGAs. Due to routing difficulties using ISE 14.1 on Virtex-6, we were forced to reduce the mantissa from 116b (two 58b blocks) down to 87b (three 29b blocks). This reduces the size of most internal modules (multiplier, adder, etc.) by almost 25% at cost of a more complex multiplexer at the end (11-to-1 instead of 6-to-1). However it enables 200+ MHz operation.

When the result mantissa consists of three blocks, blocks of 29 FCS digits (each digit having 1b partial sum and 1b CS carry, together expressed in the unit *c* from here on) are required to surpass double accuracy: In the worst case, the first result block and the first digit of the second block can all be zero, but the following non-zero digit prevents the removal of the leading zero block (see Fig. 10.c). In addition, when using early leading zero estimation, there is a three bit uncertainty to consider, possibly causing three further digits (4c in total) of block two to be zero. On the other hand, this means that even in the worst case, at least 25c in block two and all 29c in block three are significant FCS digits (54c in total), exceeding IEEE 754 double-precision with its mantissa of 52b+1b binary digits.

The inputs to the FCS-FMA unit (shown in Fig. 11) consist of the three exponents (12b for A and C, 11b for B) and B_M in standard format (52b+1b leading **1**). A_M and C_M are represented in FCS as 87c each, accompanied by 29c of rounding data. The output is a 87c result mantissa, 29c of rounding data and 12b exponent.

The width of the result multiplexer must be sized accordingly: The multiplication yields a five block wide result. The shifter aligning the addend A_M to match exponents has an additional three blocks on the right hand (less significant side) and five blocks on the left hand (more significant side), yielding a total of 13 blocks, each 29c wide, for a total width of 377c.

The final multiplexer for the result selects from these 13 blocks the three most significant non-zero blocks. It thus accepts 13 blocks as inputs and selects from 11 possible positions for the three block result R_M holds, which holds at least 53 significant mantissa digits, possibly shifted across three blocks (87c). A parallel multiplexer outputs the 29c of the mantissa immediately to the right of the actual result R_M for rounding in a subsequent FCS-FMA operator.

I. Automatic P/FCS-FMA unit insertion in high-level synthesis

Manually replacing critical discrete multiply-add operations by FMA operations and performing the appropriate

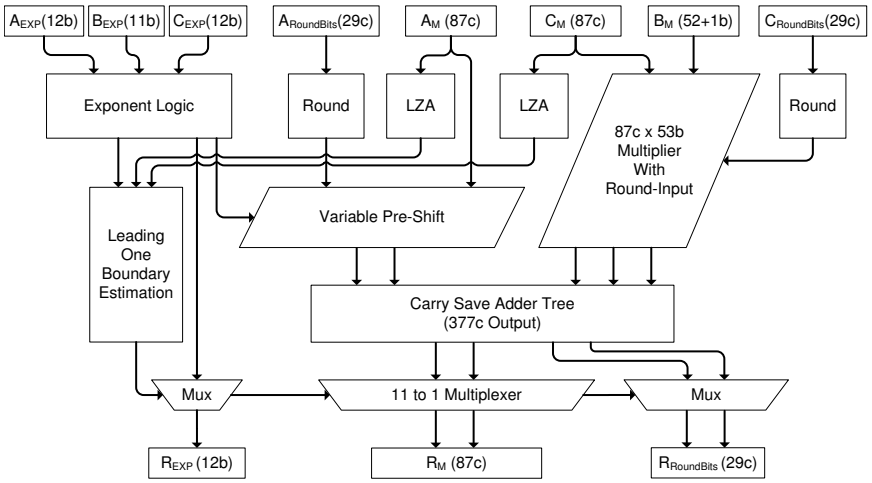
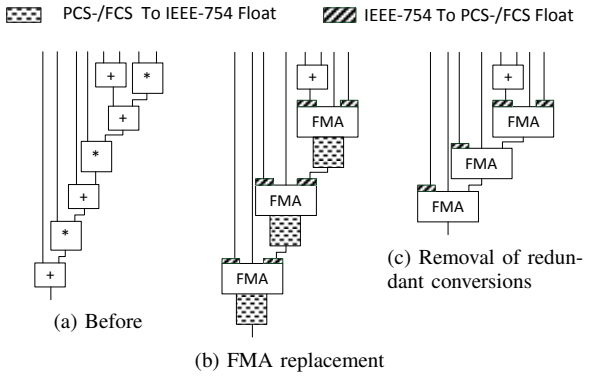


Figure 11: FCS-FMA unit exploiting DSP block pre-adders

type conversions is both tedious and error prone. We have integrated a pass into our C-to-hardware compiler Nymble that performs the required analysis and transformations automatically.

The datapath is initially assembled from IEEE 754 operators and scheduled (Fig. 12a). Then, the datapath is searched for pairs of successive multiply and add operators. If they are on the critical path, the pair gets replaced by a P/FCS-FMA unit, surrounded by the required conversion logic between the CS and IEEE 754 formats. After all critical multiply/adds have been greedily replaced by FMA units (Fig. 12b), redundant type conversions between FMA units are removed (Fig. 12c), the entire datapath is rescheduled, and the procedure repeats until no further FMA insertions can be performed.



(a) Before

(b) FMA replacement

(c) Removal of redundant conversions

Figure 12: Insertion steps

IV. Experimental Evaluation

For evaluation, both proposed P/FCS-FMA operations have been implemented on a Xilinx Virtex-6 FPGA. and their multiplier modules have been specially optimized to exploit the Xilinx DSP48E1 blocks. For comparison with the industrial and academic state-of-the-art, Xilinx CoreGen and the FloPoCo library were used to generate IEEE 754 double-precision units for the Virtex-6 family. Note that none of these units supports subnormals [14, 15] and all were constrained to achieve a minimum clock frequency of 200 MHz.

A. Synthesis results

FloPoCo allows the definition of target frequency, technology and bit width by command line parameter. We used the FPPipeline command to allow optimizations across the multiplier and adder units [24]. The resulting hardware model was synthesized with and without register balancing, using the better result as baseline for the comparison.

In contrast to FloPoCo, Xilinx CoreGen only allows the generation of separate multiply/add units and the specification of operator latency. Thus, we manually selected the configuration with the lowest latency that still managed to achieve the target clock. The specific designs chosen were the ”low latency” 5 cycle multiplier and ”low latency” 4 cycle adder. Our P/FCS-FMA units have been manually pipelined to 200 MHz operation.

Table I shows the synthesis results achieved using Xilinx ISE 14.1. All results are taken from post-layout timing reports. While FloPoCo achieves the smallest implementation (in terms of DSP usage), its latency of 11 cycles is also the slowest in the test. The FCS-FMA unit is the fastest unit, followed by the PCS-FMA unit. Note that the FCS-FMA unit achieves better area efficiency than the PCS variant due to its exploitation of the DSP48E1 pre-adder blocks, which would not be available on earlier FPGAs. However, both of our units require more area (LUTs) than their competitors.

Figure 13 shows the minimum computation time for a single Multiply-Add-Operation. It is calculated by multiplying the minimum cycle time with the number of clock cycles

Table I: Synthesis results

Architecture	f_{Max}	Cycles	LUTs	DSPs
Xilinx CoreGen	244	9	1253	13
FloPoCo FPPipeline	190	11	1508	7
PCS-FMA	231	5	5832	21
FCS-FMA	211	3	4685	12

required to complete one computation. The PCS- and FCS-FMA units are about 1.7x and 2.5x faster than their closest competitor.

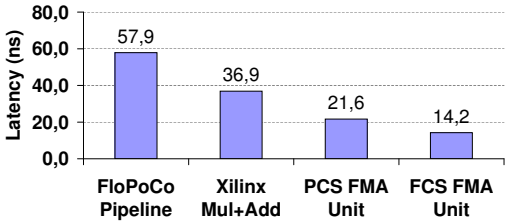


Figure 13: Latency (as minimum clock period times pipeline length) for FloPoCo, Xilinx and P/FCS-FMA pperations

B. Numerical Accuracy

As discussed earlier, with the exception of limitations in rounding fidelity, our P/FCS-FMA units are guaranteed to reach or exceed IEEE 754 double-precision accuracy. To study the impact of the potential misrounding (see Sec. III-C), we fed valid but random data into a pair of FMA units recursively computing the value $x[50]$ as described in Equation 1, where B_1 and B_2 are random numbers with $1 < |B_1| < 32$ and $1 > |B_2| > 0$.

$$x[n] = B_1 * x[n-1] + B_2 * x[n-2] + x[n-3] \quad (1)$$

The same computation is also performed on data widths of 64b (IEEE 754 double), 68b, and 75b using the Xilinx CoreGen floating-point operations as reference. The 68b and 75b variants employ a larger mantissa for improved accuracy.

Figure 14 illustrates the average mantissa error of 64b, 68b and FCS-FMA implementation. The result of the 75b CoreGen computation was used as golden reference to gauge the errors of the less accurate implementations. Both PCS and FCS-FMA units clearly outperform standard IEEE double precision in terms of average accuracy.

C. Energy consumption

The energy consumption was analyzed by the Xilinx XPower tool considering the actual switching activity of the units. Post-layout delays were extracted and the activity recorded in VCD/SAIF format using the Xilinx ISim simulator on the benchmark computations described in Sec. IV-B. The pipeline is examined in steady-state (producing one $x[i]$ per clock cycle) after sufficient priming.

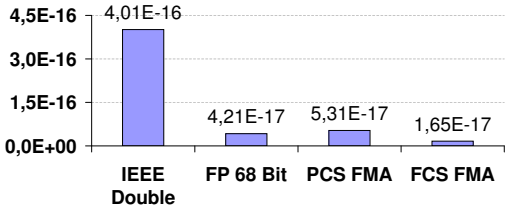


Figure 14: Average mantissa error in x[50] (arithmetic mean over 20 computations)

Table II: Average energy consumption per multiply-add computation (nJ)

Xilinx (Mul+Add)	FloPoCo	PCS-FMA	FCS-FMA
0.54	0.74	2.67	2.36

The increased performance of our P/FCS-FMA units comes at a 4x to 5x increase in energy consumption. The XPower analysis details showed that most of the energy was drawn in the large CSA trees of multiplication and addition. Obviously, our P/FCS-FMA units are not suitable for ultra low power operation. However, due to the much lower general energy consumption of FPGAs compared to GPGPUs and GPPs [25, 26], FPGA designs using P/FCS-FMAs may still be competitive energy-wise with other implementation technologies. Furthermore, both architectures are applicable to the high-performance computing domain.

D. Application in High-Level-Synthesis

The P/FCS-FMA units have been made available for high-level synthesis using the approach outlined in Sec. III-I. The Nymble hardware compiler was then used to compile parts of three convex solvers generated by CVXGEN as discussed in Sec. I. The `ldlsolve()` function, which holds the core solver algorithm, is selected for hardware compilation. It requires more than half of the execution time on a general-purpose processor and can thus be considered a compute kernel. As above, floating-point operators have been chosen for a target frequency of 200+ MHz.

The resulting schedule length is shown in Figure 15. It could be reduced by 26.0% to 50.1% when selectively replacing discrete multiply/add operations with up to 39 time-multiplexed P/FCS-FMA units. Note the higher performance gains achievable using the FCS approach, which is however limited to recent FPGA architectures due to its reliance on the DSP48E1 pre-adder functionality.

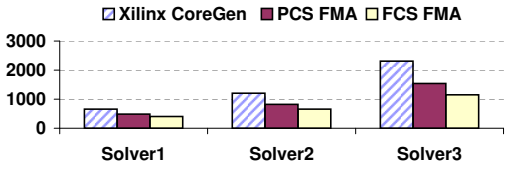


Figure 15: `ldlsolve()` schedule cycles for increasing solver complexity

V. Conclusion and future work

We performed an architecture exploration for the realization of fast fused multiply-add units, also taking into account specific features of recent FPGA families. The resulting operators rely on carry-save floating-point representations and could be shown to deliver up to 2.5x the performance of the industrial Xilinx CoreGen IEEE 754 double-precision operations.

Our P/FCS-FMA units can be employed selectively in high-level synthesis to accelerate the critical paths of compute kernels, converting between standard and custom floating-point representations as required. Application-level benchmarks on the synthesis results for the hardware acceleration of convex solvers have demonstrated speed-ups of up to 50%.

Since these benefits come at the cost of increased area and energy requirements, a selective use, as suggested by our own high-level synthesis integration, is recommended.

For future work, the use of different carry bit densities in the PCS-FMA could be explored when increasing the block size to 56b (instead of the 55b used here). Furthermore, the concept of mantissas represented in partial/full carry save formats could applied to other floating-point operations.

VI. Acknowledgment

This work is part of the LOEWE program funded by the Hessian Ministry of Higher Education, Research and Arts.

References

[1] H. Gadke-Lutjens, B. Thielmann, and A. Koch, "A Flexible Compute and Memory Infrastructure for High-Level Language to Hardware Compilation," *Int. Conf. on Field Programmable Logic and Applications*, pp. 475–482, 2010.

[2] J. Mattingley and S. Boyd, "CVXGEN: a code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, no. 1, pp. 1–27, Nov. 2011.

[3] E. Engineers, "IEEE Standard for Binary Floating-Point Arithmetic," pp. 1–23, 1985.

[4] E. Hokenek, R. Montoye, and P. Cook, "Second-generation RISC floating point with multiply-add fused," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 5, pp. 1207–1213, 1990.

[5] E. Quinnell, E. S. Jr, and C. Lemonds, "Three-path fused multiply-adder circuit," 2011.

[6] S. Jain, V. Erraguntla, S. R. Vangal, Y. Hoskote, N. Borkar, T. Mandepudi, and K. Vp, "A 90mW/GFlop 3.4GHz Reconfigurable Fused/Continuous Multiply-Accumulator for Floating-Point and Integer Operands in 65nm," *2010 23rd Int. Conf. on VLSI Design*, pp. 252–257, Jan. 2010.

[7] J. Brooks and C. Olson, "Processor Pipeline which Implements Fused and Unfused Multiply-Add Instructions," 2012.

[8] E. Quinnell, *Floating-point fused multiply-add architectures*, Dissertation, University of Texas at Austin, 2007.

[9] H. H. Saleh and E. E. Swartzlander, "A floating-point fused dot-product unit," in *2008 IEEE Int. Conf. on Computer Design*. Oct. 2008, pp. 427–431, IEEE.

[10] E. Swartzlander and H. Saleh, "FFT implementation with fused floating-point operations," *Computers, IEEE Transactions ...*, vol. 61, no. 2, pp. 284–288, 2012.

[11] 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, pp. 73–79, Jan. 2010.

[12] F. D. Dinechin and B. Pasca, "An FPGA-specific approach to floating-point accumulation and sum-of-products," *... Technology, 2008. FPT ...*, pp. 33–40, 2008.

[13] B. Catanzaro and B. Nelson, "Higher Radix Floating-Point Representations for FPGA-Based Arithmetic," *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pp. 161–170, 2005.

[14] 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.

[15] Xilinx, "Xilinx LogiCORE IP Floating-Point Operator v5.0 Product Specification," Tech. Rep., 2011.

[16] K. S. Hemmert and K. D. Underwood, "Fast, Efficient Floating-Point Adders and Multipliers for FPGAs," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 3, no. 3, pp. 1–30, Sept. 2010.

[17] 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, July 2011.

[18] M. Langhammer, "Floating point datapath synthesis for FPGAs," in *2008 Int. Conf. on Field Programmable Logic and Applications*. 2008, pp. 355–360, IEEE.

[19] M. Langhammer and T. VanCourt, "FPGA Floating Point Datapath Compiler," in *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*. 2009, pp. 259–262, IEEE.

[20] B. Parhami, *Computer Arithmetic*, Oxford University Press, second edition, 2010.

[21] A. K. Verma and P. Ienne, "Improved use of the carry-save representation for the synthesis of complex arithmetic circuits," in *Proceedings of the 2004 IEEE/ACM Int. conf. on Computer-aided design*. 2004, pp. 791–798, IEEE Computer Society.

[22] A. Verma, A. K. Verma, H. Parandeh-Afshar, P. Brisk, and P. Ienne, "Synthesis of Floating-Point Addition Clusters on FPGAs Using Carry-Save Arithmetic," *2010 Int. Conf. on Field Programmable Logic and Applications*, pp. 19–24, Aug. 2010.

[23] M. Schmookler and K. Nowka, "Leading zero anticipation and detection-a comparison of methods," *Computer Arithmetic, 2001. ...*, pp. 7–12, 2001.

[24] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *2009 Int. Conf. on Field Programmable Logic and Applications*. Aug. 2009, pp. 59–64, IEEE.

[25] J. Huthmann, P. Müller, F. Stock, D. Hildenbrand, and A. Koch, "Compiling Geometric Algebra Computations into Reconfigurable Hardware Accelerators," in *Dynamically Reconfigurable Architectures*, P. M. Athanas et al., Eds., Dagstuhl, Germany, 2010, number 10281 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[26] A. Engel, B. Liebig, and A. Koch, "Feasibility analysis of reconfigurable computing in low-power wireless sensor applications," in *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 261–268. 2011.