REDUCED COMPLEXITY SINGLE AND MULTIPLE CONSTANT MULTIPLICATION IN FLOATING POINT PRECISION

Martin Kumm, Katharina Liebisch, Peter Zipf

Digital Technology Group
University of Kassel, Germany
Email: kumm@uni-kassel.de, liebisch@student.uni-kassel.de, zipf@uni-kassel.de

ABSTRACT

This paper addresses the automatic generation and optimization of single and multiple constant multipliers in IEEE 754 floating point precision for FPGAs. It is shown that sharing of partial results in multiplication and exponent addition as well as the handling of special input values can greatly reduce the overall hardware complexity. Two methods are used to reduce the complexity of the integer multiplier block: An existing method using adder arithmetic only and a novel optimization method using a reduced amount of embedded multipliers to compute products with large coefficient values. Superior results are shown compared to previous methods. Using adder arithmetic, a slice reduction of 18% for single and 38% for multiple constant floating point multiplication could be achieved. Using embedded multipliers, nearly half of the multipliers could be saved for multiple constants compared to the conventional approach.

1. INTRODUCTION

The increased complexity and speed of FPGAs combined with the demand for higher accuracy has been a driving force for many floating point applications in the last years. The floating point format is in particular attractive in real time digital signal processing systems but also for FPGAs acting as co-processor in custom computing machines. Besides the higher dynamic range, the direct use of floating point on FPGAs has the big benefit that no additional effort for fixed point conversion is needed, so algorithms can be directly transferred from software (like Matlab or Simulink) to hardware. Of course, the penalty is that much more resources are needed, justifying the research for reduced complexity floating point operations. Examples are basic operators like generic multiplication [1], constant multiplication [2] but also more complex operations like sum of squares [3]. These operators can be further used as library elements in high-level synthesis tools to enable the design of custom floating point data paths, even for software programmers [3].

In this paper, we address the efficient generation and optimization of floating point cores which multiply with single or multiple constants. While the single constant multiplication (SCM), or scaling operation is used in nearly every kind of numeric algorithm, the multiple constant multiplication (MCM) frequently appears in digital signal processing algorithms. Especially, it is used in products of vectors with constant matrices including digital filters [4] (finite/infinite impulse response, FIR/IIR) and discrete transforms (e. g., the fast Fourier transform [5]).

One important component of floating point MCM (FPMCM) is an integer MCM. It can be reduced to additions, subtractions and shifts which are organized in a so-called adder graph. Finding an adder graph with minimal number of adders and subtractors (shifts are assumed to be free) is known as MCM problem which is an NP-complete problem [6]. Many efficient heuristics exist [6, 7, 8] that are able to realize an adder graph with very few operations. However, the direct implementation on FPGAs leads to very slow designs [9] so that efficient pipelining is necessary [4]. But even with optimal pipelining using binary integer linear programming (BILP) techniques [4], it was shown recently that the direct optimization of pipelined adder graphs (PAGs) is advantageous [10, 11]. The heuristic behind the optimization in [10] is called reduced pipelined adder graph (RPAG) algorithm. This algorithm is used in this work for realizing slice-only multiplier blocks and is further extended to an FPGA specific reduction of embedded multipliers.

The contributions of this paper are: 1) the extension of MCM to floating point precision, which is, to the best of our knowledge, the first time this topic is investigated, 2) an optimized SCM method that outperforms state-of-the-art methods, 3) a novel method to reduce the amount of embedded multipliers for hybrid integer MCM blocks.

2. FLOATING POINT MULTIPLE CONSTANT MULTIPLICATION

Consider a floating point number x which is coded using sign s_x, mantissa f_x and exponent e_x, representing the value

$$x = (-1)^{s_x} \cdot (1.f_x) \cdot 2^{e_x - bias} \quad . \tag{1}$$

The components of the floating point product $p_n = x \cdot c_n$ can be obtained by multiplying the (integral) mantissas, adding the exponents subtracted by the bias and the XOR of the signs:

$$1.f'_{p_n} = 1.f_x \cdot 1.f_{c_n} \quad (2)$$

$$e'_{p_n} = e_x + e_{c_n} - bias \quad (3)$$

$$s_{p_n} = s_x \oplus s_{c_n} \quad (4)$$

The resulting mantissa f'_{p_n} must be rounded and normalized to the mantissa word size, denoted as w_f, to get the final f_{p_n}. As the value of the input mantissas are in the interval $1.f_x, 1.f_{c_n} \in [1, 2)$, the product mantissa will result in $1.f'_{p_n} \in [1, 4)$. Hence, if normalization is necessary, it can be done using a single bit shift to the right while incrementing the exponent, leading to the final e_{p_n}.

In floating point data paths, each operation is usually realized by specific floating point operators which include normalization and rounding of the final result and handling of special values (0, $\pm\infty$ and not-a-number, NaN). However, exceptions from that show that many resources can be saved by computing more complex operations using integer arithmetic and performing the post processing as late as possible. One example is the hand optimized sum of squares operation [3]. Of course, this is only beneficial if the dynamic between several processed signals is limited during integer operation as the bit width may extensively grow. This is exactly the case for FPMCM as there is only one input. For SCM, which can be seen as special case of MCM, redundancies in the coefficient representation can be used to reduce the effort of the integer multiplier which was already proposed by Brisebarre et al. [2]. Much more resources can be reduced in the MCM case: 1) Redundancies between several constant integer mantissas can be utilized, 2) equal exponent computations of constants can be shared, 3) special values of the input have to be detected only once.

2.1. Proposed Architecture

The proposed architecture is shown in Fig. 1. It is explained using the 32 bit single precision format [12] but can be easily extended to other floating point formats. Input x is split into its components, the hidden bit is appended to the mantissa and special values are detected. The computation of (2) with all constants is done in a pipelined integer multiplier block, producing N results. Each product is rounded, normalized and combined with the corresponding exponent and sign in a post-processing stage. The exponent computation, post-processing and integer multiplication are described in the following.

2.1.1. Exponent Computation

There are two possibilities in realizing the exponent computation. The obvious way is to implement an adder for each

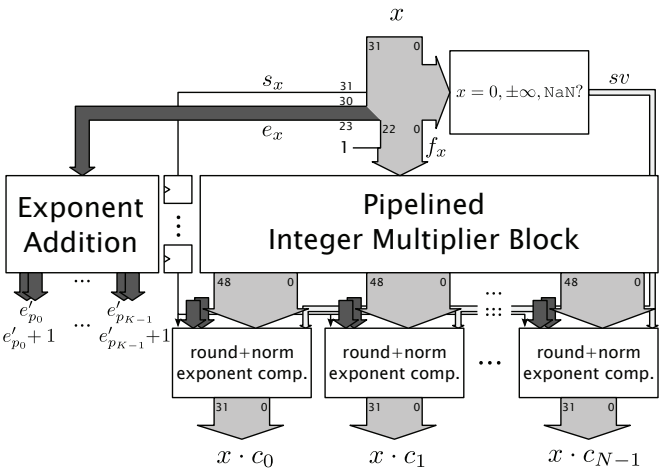

Fig. 1. Proposed architecture for the single precision floating point MCM operation

output that adds $(e_{c_n} - bias)$ to the input exponent e_x with an optional incrementation in case of normalization. This operation can be mapped to a single FPGA ripple carry adder using the carry-in for incrementation.

The second way is to add both possible offsets $(e_{c_n} - bias)$ and $(e_{c_n} - bias + 1)$ to e_x and to select the correct offset after normalization using a multiplexer (MUX). At first sight, this seems to be inefficient as twice the adders and an additional MUX are needed, but now the adder results can be shared between the post-processing stages as they are independent from normalization. Adders can be saved when many equal exponents or exponents that differ by one exist. More exactly, adders can be saved if the condition

$$\left| \bigcup_{n=0}^{N-1} \{e_{c_n}, e_{c_n} + 1\} \right| < N \quad (5)$$

is met. It is shown later that this is the case for all instances of the MCM benchmark set. Of course, the condition is not met for single constant multiplication. As shown in the following, the required MUX is already available using a free port of the 4:1 MUX used for special value handling.

2.1.2. Post Processing

The post processing, which is needed for each output, includes rounding, normalization, selection of the exponent and handling of special values. The detailed architecture is shown in Fig. 2.

For the multiplication with a constant, the rules $0 \cdot c_n = 0$, $\pm\infty \cdot c_n = \pm\infty \cdot (-1)^{s_{c_n}}$ and NaN$\cdot c_n$ =NaN must be realized. Overflows and underflows of the exponent must result in $(-1)^{s_x} \cdot \infty$ and zero, respectively. The detection is done in the block 'exception handling', which computes

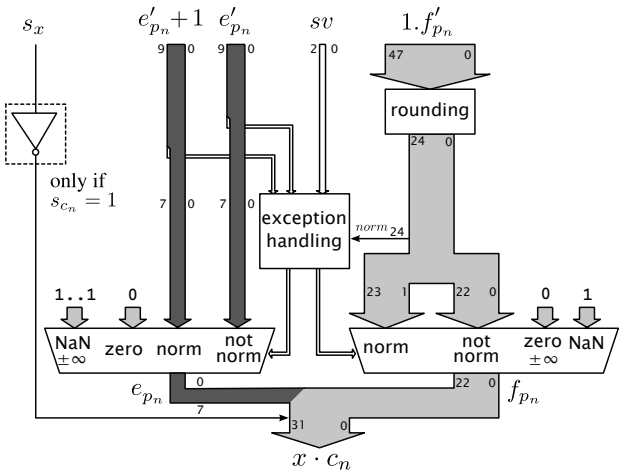

Fig. 2. Detailed architecture of one round, normalize and exponent selection block of Fig. 1 (pipeline registers are not shown for clarity)

the control signals of the exponent and mantissa multiplexers depending on special values of the input sv and overflows/underflows of the exponent. Using shared exponents, the exponent multiplexer selects the right exponent result or a $11\ldots11$ vector in case of `NaN` or $\pm\infty$, or a 0 in case of a zero output value.

The resulting product $1.f'_{p_n}$ is first rounded using the standard "round to nearest, even if tie" scheme [12]. The normalization which depends on the MSB (bit $norm$) is done using the mantissa multiplexer which is also used to select special values of the mantissa (0 for zero output or $\pm\infty$, non-zero for `NaN`). As modern FPGAs directly offer 4:1 multiplexers, the exponent and mantissa multiplexers map well to FPGA logic.

The full design process was automated using VHDL code generation. The generated cores are IEEE 754 compliant [12] except the support for denormalized values. The use of shared exponents is enabled when criteria (5) is met, otherwise distinct adders are implemented. The optimization of the integer multiplier block is described in the next section.

3. RPAG EXTENSION FOR HYBRID ADDER/MULTIPLIER OPTIMIZATION

3.1. Outline of the RPAG Algorithm

The RPAG algorithm [10] constructs a pipelined adder graph (PAG) to compute a multiplication with several integer constants. An example PAG for the coefficient set $\{11280171,\ 13342037, 11508221,\ 9104278,\ 13820478\}$ is shown in Fig. 3. Each node in a PAG corresponds to a registered adder, subtractor or a pure register. The shifted add/subtract operations are formally represented as $\mathcal{A}$-operation [6],

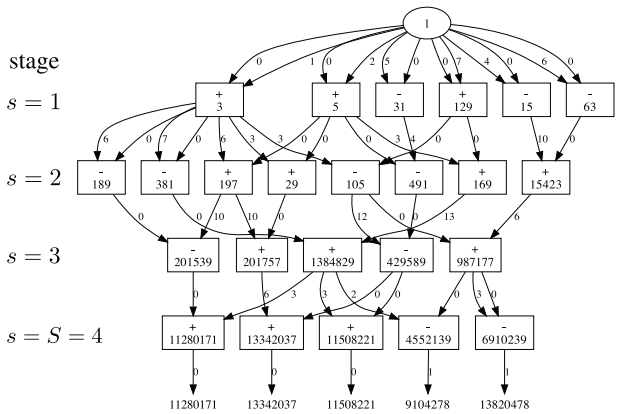

Fig. 3. Pipelined adder graph for the $N = 5$ benchmark filter

which is defined as

$$\mathcal{A}_q(u,v) = |2^{l_1}u + (-1)^{sg}2^{l_2}v|2^{-r} \tag{6}$$

with $q = (l_1, l_2, r, sg)$, where u and v are the input arguments, l_1, l_2 and r are shift factors and the sign bit $sg \in \{0, 1\}$ denotes whether an addition or subtraction is performed. All edge weights correspond to shift factors l_1 and l_2, e. g., node 3 in the first stage of Fig. 3 is realized by adding the unshifted input x with the single left shifted input, leading to: $2^0x + 2^1x = 3x$.

A short outline of RPAG is given in the following, a detailed description can be found in [10]. The minimal number of pipeline stages S is identical with the maximum of all minimal adder depths $\text{AD}_{\text{min}}(t)$, which is defined as the number of adder stages needed to compute a coefficient t:

$$S := \max_{t \in T} \text{AD}_{\text{min}}(t) \tag{7}$$

The minimal adder depth can be computed using

$$\text{AD}_{\text{min}}(t) = \lceil \log_2(\text{nz}(t)) \rceil \tag{8}$$

where $\text{nz}(t)$ is the number of nonzero digits of t in minimum signed digit (MSD) representation [10].

In contrast to most existing MCM algorithms, the RPAG algorithm searches from the output nodes of the adder graph to the input node 1. That means, starting from stage $s = S$, valid elements in stage $s - 1$, called predecessors, are evaluated if they are able to compute elements in stage s using one $\mathcal{A}$-operation. The locally best elements are then selected in a greedy manner. The evaluation is started with single predecessors which are realized by one of the three possible topologies of Fig. 4 (a)-(c):

a) Element w has a lower adder depth than s and can be copied from p using a register

b) w can be computed by $p \cdot (2^k \pm 1)$ with $k \in \mathbb{N}$

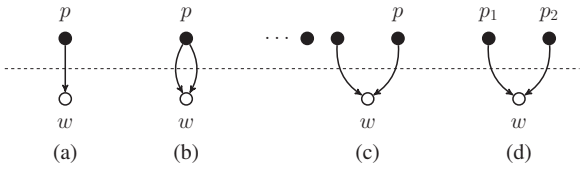

Fig. 4. Predecessor graph topologies for (a)-(c) a single predecessor p, (d) a predecessor pair (p_1, p_2)

c) w can be obtained by an $\mathcal{A}$-Operation using p and another element which was already realized

If a single predecessor was found, the best one is chosen using a gain criteria [10]. Otherwise, pairs (p_1, p_2) of predecessors are evaluated witch are obtained from the MSD representation of w according to the topology in Fig. 4 (d).

Each valid predecessor p is constrained to have a lower adder depth than the actual stage s:

$$\mathrm{AD}_{\min}(p) < s \tag{9}$$

In each iteration, a single predecessor or a predecessor pair is selected until all elements in stage s can be computed using the selected predecessors. Now, s is decreased by one and the process is repeated until $s = 1$ is reached.

As the best local choice in a greedy algorithm does not necessarily lead to the best global solution, RPAG supports a configuration option to randomly select one of the n-th best predecessors. Using this option, the result can be further enhanced by evaluating several runs of the algorithm.

3.2. Using RPAG for Embedded Multiplier Reduction

PAGs can be efficiently realized on FPGAs using the fast carry chains. However, modern FPGAs provide more and more embedded multipliers which can, of course, be involved in constant multiplication. The Xilinx Virtex-5/6/7 FPGAs provide asymmetric 18x25 bit signed multipliers in their DSP48E(1) slices [13]. Even for single precision, these multipliers are too small, as a 24x24 bit unsigned multiplication is necessary to compute (2). Different methods exist for constructing large multipliers from smaller ones [14, 1]. In the simplest case, coefficient b of the (integer) product $a \cdot b$ can be decomposed into its bits b_i which are rearranged into two numbers b^L and b^H of half the word size w:

$$a \cdot b = a \cdot \sum_{i=0}^{w-1} 2^i b_i \tag{10}$$

$$= a \cdot \sum_{i=0}^{w/2-1} 2^i b_i + 2^{w/2} a \cdot \sum_{i=0}^{w/2-1} 2^i b_{i+w/2} \tag{11}$$

$$= a \cdot b^L + 2^{w/2} a \cdot b^H \tag{12}$$

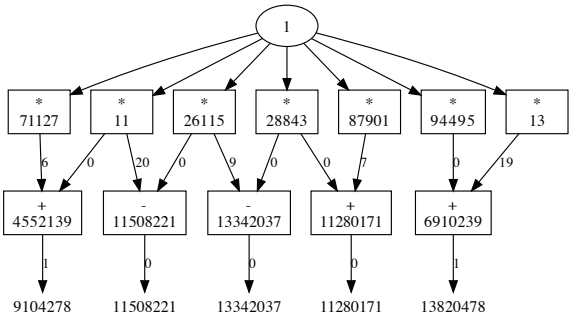

Fig. 5. Hybrid adder/multiplier graph adder graph for the $N = 5$ benchmark filter

Using this scheme, a single precision multiplier can be realized using two DSP48E slices where the adder of one DSP slice can be used for final summation.

Note, that the composition of b as described in (12) is identical to the $\mathcal{A}$-operation in (6), where $u = b^L$, $v = b^H$ and $q = (0, w/2, 0, 0)$. Hence, the constants b^L and b^H can be seen as predecessors p_1 and p_2 of b. As in the PAG optimization, there is a large degree of freedom in choosing p_1 and p_2 to compute a given b. This fact can be used to find partial results that can be shared to reduce the total number of multipliers. The main difference to RPAG is, that the predecessors are not constrained to have a smaller adder depth, instead they must have a reduced word size. This can be integrated in RPAG by replacing (9) with

$$\lceil \log_2(p) \rceil < w_{\max} \ . \tag{13}$$

This leads to the hybrid adder/multiplier optimization. Due to the unsigned representation of p, $w_{\max}$ is equal to 17 bit for DSP48E slices. For single precision with $w_f = 24$ it is guaranteed that two constants p_1 and p_2 can be found to produce any mantissa. Thus, RPAG was modified to reduce the word size using a single stage, which is suitable for all single precision mantissas. Fig. 5 shows an example of a hybrid realization of the PAG from Fig. 3 where the first stage uses embedded multipliers. Only seven embedded multipliers are needed compared to ten according to (12). Note, that it is not possible to just use the last two stages of Fig. 3, as the second last stage has larger word size than 17 bit.

4. RESULTS

Synthesis experiments were made for several single and multiple constant multiplication instances using the single precision format (for variables and constants) as this is used by most floating point digital signal processors. In total, 50 runs per instance were performed while RPAG and hybrid RPAG were configured to randomly select the 1st or 2nd best solution.

Table 1. Benchmark of floating point single constant multiplication

C	FloPoCo [2]				Proposed Method			
	FF	LUT	Slices	$f_{\max}$ [MHz]	FF	LUT	Slices	$f_{\max}$ [MHz]
$\pi/2$	405	363	130	236.2	**357**	**268**	**102**	**431.4**
$\sqrt{2}$	461	435	154	215.8	**447**	**341**	**134**	**414.4**
e	404	360	133	214.8	**385**	**305**	**110**	**435.0**
$\sin(\pi/3)$	440	422	145	217.3	**374**	**304**	**103**	**429.0**
$\sin(\pi/4)$	461	433	150	237.9	**449**	**337**	**127**	**440.5**
$\sin(\pi/8)$	408	365	137	208.3	**370**	**288**	**101**	**416.5**
$\cos(\pi/8)$	407	360	133	230.4	**357**	**262**	**107**	**429.0**
$\sin(\pi/16)$	**381**	339	117	202.7	384	**267**	**114**	**368.7**
$\cos(\pi/16)$	413	375	140	227.1	**385**	**290**	**117**	**392.6**
$\tan(\pi/16)$	460	370	139	240.5	**416**	**327**	**114**	**411.7**
Average:	424	382	138	223.1	392	299	113	416.9
Improv.:	7.5%	21.8%	18.1%	86.9%	-	-	-	-

The synthesis was done using Xilinx ISE 13.4 for the Virtex-6 FPGA (XC6VSX475T-1). The pipeline depth is seven for all the MCM blocks of the proposed method. Registers are placed at the input and output of each core to avoid that timing analysis is dominated by routing delays. All synthesis results were obtained after place & route.

4.1. Single Constant Multiplication

The SCM results are compared with the state-of-the-art method of Brisebarre [2] which is part of the FloPoCo project[1]. FloPoCo internally uses a non-standard floating point representation which uses two extra bits for the handling of special values. This representation is very advantageous, as only one decoding and coding block for special values is necessary for each IEEE-754 compliant input and output, respectively. As our implementation includes these blocks, the corresponding FloPoCo blocks `InputIEEE` and `OutputIEEE` were used and pipeline registers were placed between all FloPoCo blocks to have a fair comparison. As the SCM block of FloPoCo (`FPConstMult`) uses mathematical rounding instead the standard "round to nearest even if tie" [12] this was also used in the SCM experiment.

The synthesis results for SCM are listed in Table 1 (best values are marked bold). Some of the constants are chosen arbitrary, the trigonometric constants appear, e.g., in FFTs [5]. The pipeline depth is between 6-7, where 3-4 stages (depending on the adder depth of the constant) are used for the integer multiplier block. A slice reduction of 18% could be achieved compared to FloPoCo [2] while the speed is improved by 87% on average. This is mainly due to more compact adder graphs of RPAG which uses less registers (up to five less) and adders (up to two less). Using retiming of ISE, the speed of FloPoCo could be increased to 359.8 MHz but at a price of 22 additional slices, which corresponds to a slice reduction of 28% and a speedup of 16% (all average values). Hence, always less resources and higher speed could be achieved compared to FloPoCo.

[1] Available at http://flopoco.gforge.inria.fr

Table 2. Required adders for exponents

N	Conventional	Shared
5	5	4
10	10	8
20	20	11
30	30	12
40	40	15
50	50	17
60	60	19
70	70	22
Avg.:	35.63	13.5

4.2. Multiple Constant Multiplication

As there is no competing method for FPMCM, additional code generators were written in Matlab for cores that use several SCM cores of FloPoCo and, alternatively, floating point multiplier blocks of Xilinx LogiCORE [15] using Coregen. Coefficient sets of eight FIR filters of different length served as a realistic benchmark set. They were designed using the Parks-McClellan algorithm (Matlab `firpm` function) and quantized to single precision. All filters are low pass filters with a normalized passband and stopband frequency of $f_p = 0.3$ and $f_s = 0.4$, respectively. The number of taps was varied from 10 to 140 resulting in passband/stopband approximation errors from -15 dB to -110 dB. Due to the symmetry of the coefficients, the number of unique coefficients N is half the number of taps.

4.2.1. Adder Savings due to Exponents Sharing

The number of adders required to compute the exponents are listed in Table 2 for both methods described in Section 2.1.1. Clearly, many adders can be saved for the benchmark coefficients using shared exponents. To give an example, the exponents of the coefficient set for $N = 5$ are $e_c = (123, 123, 123, 124, 125)$. According to (3) and (5), $e_{c_n} - bias$ and $e_{c_n} - bias + 1$ are required leading to exponent offsets of {-4,-3,-2,-1} for single precision ($bias = 127$). This is performed using 4 instead of 5 additions. Of course, filter coefficients are not random so exponent sharing may become worse for other coefficient values.

Table 3. Benchmark of floating point multiple constant multiplication using slices only

N	Coregen [15] (logic only)				FloPoCo [2]				Proposed Method (logic only)			
	FF	LUT	Slices	$f_{\max}$	FF	LUT	Slices	$f_{\max}$	FF	LUT	Slices	$f_{\max}$
5	3204	2966	1017	289.2	1194	1563	486	210.3	**893**	**1050**	**310**	**333.0**
10	6427	5950	1944	279.9	2015	2755	813	163.4	**1463**	**1784**	**554**	**306.5**
20	12818	11893	3843	237.8	3321	4980	1518	167.1	**2717**	**3434**	**995**	**300.5**
30	19228	17841	5665	226.5	4886	7473	2214	154.4	**3511**	**4453**	**1367**	**233.7**
40	25635	23749	7503	204.3	6094	9225	2681	139.0	**4396**	**5723**	**1627**	**255.4**
50	32054	29732	8810	201.2	6982	11516	3271	157.2	**5349**	**7026**	**2018**	**228.8**
60	38431	35677	11022	185.0	8346	14138	4191	153.8	**6143**	**8159**	**2384**	**218.2**
70	44872	41635	12592	180.0	9478	15364	4531	146.4	**7183**	**9663**	**2832**	**211.2**
Average:	22833.6	21180.4	6549.5	225.5	5289.5	8376.8	2463.1	161.4	**3956.9**	**5161.5**	**1510.9**	**260.9**
Improv.:	82.67%	75.63%	76.93%	15.71%	25.19%	38.38%	38.66%	61.61%	-	-	-	-

Table 4. Resource usage for the integer multiplier block using RPAG, hybrid RPAG and the conventional approach

N	RPAG	Hybrid RPAG		Conventional	
	Add/Sub	Add/Sub	Mul	Add	Mul
5	24	5	7	5	10
10	38	10	17	10	20
20	71	20	23	20	40
30	93	30	30	30	60
40	118	40	40	40	80
50	139	50	50	50	100
60	160	60	59	60	120
70	185	70	69	70	140
Average:	103.6	35.625	36.9	35.625	71.25

Table 5. Benchmark of floating point multiple constant multiplication using logic and embedded DSP48 blocks

N	Coregen (full DSP)			Proposed hybrid Method		
	Slices	DSP48	$f_{\max}$	Slices	DSP48	$f_{\max}$
5	236	10	342.7	**161**	**7**	**349.8**
10	488	20	**289.9**	**301**	**17**	263.9
20	921	40	258.8	**534**	**23**	**320.2**
30	1329	60	270.9	**827**	**30**	**298.2**
40	1726	80	248.2	**1035**	**40**	**306.8**
50	2166	100	222.6	**1352**	**50**	**272.4**
60	2593	120	214.4	**1593**	**59**	**303.8**
70	3043	140	203.3	**1803**	**69**	**304.3**
Average:	1562.8	71.25	256.4	**950.8**	**36.9**	**302.4**
Improv.:	39.16%	48.25%	17.97%	-	-	-

4.2.2. Multiple Constant Multiplication using Slices

The results for MCM blocks using Coregen and FloPoCo floating point multipliers are compared with the proposed method in Table 3. Here, Coregen was configured to fully pipelined, logic only cores. On average, the number of slices is reduced by 76.9% and 38.7% compared to Coregen and FloPoCo, respectively. The corresponding speedups are 15.7% and 61.6%. The highest resource reductions are due to the fact that common intermediate values can be efficiently re-used in the integer multiplier block.

4.2.3. Multiple Constant Multiplication using DSP48

The hybrid DSP/adder graph approach of Section 3.2 was used to find a set of 17 bit multiplier coefficients, from which the products can be computed using one additional adder stage. An example for $N = 5$ was already shown in Fig. 5. Table 4 lists the required number of adders/subtractors and multipliers for RPAG, hybrid RPAG and the conventional approach based on splitting the constant according to (12). The hybrid RPAG approach clearly outperforms the conventional approach. In general, the same number of adders/subtractors are used but for large N, only half of the embedded multipliers are necessary. However, it must be noted that each adder of the conventional method can be completely mapped into one DSP48E(1) slice of Virtex-5/6/7, which contains a 48 bit output adder including a 17 bit input shift. Using the proposed method, the adder can be used only for some of the nodes (e. g. 6910239 in Fig. 5), as each DSP slice has only one output which prevents the reuse of the multiplier result when the adder is in use. A separate output of the multiplier would be a nice extension for future FPGA generations. Compared to RPAG, the hybrid method uses approximately 68 adders less by using 37 additional multipliers on average. Thus, the reduction of two adders is paid for about one multiplier.

Results from synthesis are shown in Table 5. Here, Core-

gen was configured for pipelined cores with full DSP usage [15]. As expected, the number of DSP48E1 slices is drastically reduced. Due to the resource sharing an additional slice reduction of 39% and a speedup of 18% could be achieved on average.

The runtime of the RPAG and the hybrid RPAG algorithms was between 2 seconds and 7 minutes per instance and run on an Intel Nehalem 2.93 GHz CPU.

5. CONCLUSION

The automatic generation and optimization of architectures for the single and multiple constant multiplication in floating point precision targeting FPGAs was presented. It was shown, that a significant resource reduction can be achieved by sharing as many components as possible. In particular, these are partial results in integer multiplication, adders for exponent computation and the handling of special input values. A novel method for reducing FPGA-embedded multipliers was proposed which is suitable for multiple constant mantissa multiplication in single precision format.

Many applications (like digital filters) include the addition of the MCM output signals in some kind of adder tree. As floating point addition is very costly (in area and delay), further redundancy reduction could be exploited in future work, e. g., in moving the normalization of each product and summation result to the end of the adder tree while enlarging the integer data path. In addition, further work may be directed towards extending the hybrid approach which is currently limited to constant multiplications of twice the word size of the embedded multipliers.

6. REFERENCES

[1] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran, "Multipliers for floating-point double precision and beyond on FPGAs," *ACM SIGARCH Comp. Arch. News*, vol. 38, 2011.

[2] N. Brisebarre, F. de Dinechin, and J. Muller, "Integer and floating-point constant multipliers for FPGAs," in *Application-Specific Systems, Architectures and Processors (ASAP), Int. Conf. on*, 2008.

[3] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, 2011.

[4] M. Kumm and P. Zipf, "High Speed Low Complexity FPGA-based FIR Filters Using Pipelined Adder Graphs," in *Field Programmable Technology, Int. Conf. on (ICFPT)*, 2011.

[5] F. Qureshi and O. Gustafsson, "Low-complexity reconfigurable complex constant multiplication for FFTs," in *Circuits and Systems, IEEE Int. Sym. on (ISCAS)*, 2009.

[6] Y. Voronenko and M. Pueschel, "Multiplierless multiple constant multiplication," *ACM Trans. Alg.*, vol. 3, no. 2, 2007.

[7] O. Gustafsson, "A Difference Based Adder Graph Heuristic for Multiple Constant Multiplication Problems," in *Circuits and Systems, IEEE Int. Sym. on (ISCAS)*, 2007.

[8] L. Aksoy, E. O. Günes, and P. Flores, "Search Algorithms for the Multiple Constant Multiplications Problem: Exact and approximate," *Microprocessors and Microsystems*, vol. 34, 2010.

[9] S. Mirzaei, R. Kastner, and A. Hosangadi, "Layout Aware Optimization of High Speed Fixed Coefficient FIR Filters for FPGAs," *Int. Journal of Reconf. Computing*, vol. 3, 2010.

[10] M. Kumm, M. Faust, P. Zipf, and C.-H. Chang, "Pipelined Adder Graph Optimization for High Speed Multiple Constant Multiplication," in *Circuits and Systems, IEEE Int. Sym. on (ISCAS)*, 2012.

[11] U. Meyer-Baese, G. Botella, D. Romeroa, and M. Kumm, "Optimization of High Speed Pipelining in FPGA-based FIR Filter Design using Genetic Algorithm," in *SPIE Defense Security+Sensing*, 2012.

[12] "IEEE Standard for Binary Floating-Point Arithmetic," *ANSI/IEEE Std 754-1985*, 1985.

[13] *Virtex-6 FPGA DSP48E1 Slice User Guide UG369 (v1.3)*, Xilinx Inc., 2011.

[14] F. de Dinechin and B. Pasca, "Large multipliers with fewer DSP blocks," in *Field Programmable Logic and Applications, Int. Conf. on (FPL)*, 2009.

[15] *LogiCORE IP Floating-Point Operator v6.0*, Xilinx Inc., 2012.