2010 International Conference on Field Programmable Logic and Applications

An IEEE 754-2008 Decimal Parallel and Pipelined FPGA Floating-Point Multiplier

Malte Baesler, Sven-Ole Voigt, Thomas Teufel
Institute for Reliable Computing
Hamburg University of Technology
Schwarzenbergstr. 95, D-21073 Hamburg
malte.baesler@tuhh.de, s.voigt@tuhh.de, teufel@tuhh.de

***Abstract*—Decimal floating point operations are important for applications that cannot tolerate errors from conversions between binary and decimal formats, for instance, scientific, commercial, and financial applications. In this paper we present an IEEE 754-2008 compliant parallel decimal floating-point multiplier designed to exploit the features of Virtex-5 FPGAs. It is an extension to a previously published decimal fixed-point multiplier. The decimal floating-point multiplier implements early estimation of the shift-left amount and efficient decimal rounding. Additionally, it provides all required rounding modes, exception handling, overflow, and gradual underflow. Several pipeline stages can be added to increase throughput. Furthermore, different modifications are analyzed including shifting by means of hard-wired multipliers and delayed carry propagation adders.**

I. Introduction

Numerical problems are usually formulated in decimal notation. Therefore, the problems should be solved with a decimal floating-point arithmetic in order to avoid conversion errors during input and output. Because of the increasing importance, specifications for decimal floating-point arithmetic have been added to the recently approved IEEE 754-2008 Standard for Floating-Point Arithmetic that offers a more profound specification than the former Radix-Independent Floating Point Arithmetic IEEE 754-1987. For this reason, new efficient algorithms, in particular multiplication, have to be investigated and providing hardware support for decimal arithmetic is becoming more and more a topic of interest.

In this paper our previously published parallel fixed-point multiplier [1] is extended to support decimal floating-point multiplication. It is fully parallel and several configurable pipeline stages can be inserted. A fast leading zeros counter is presented and different new implementations are traded off, including delayed carry propagation adders and shifting by means of multiplication. It is IEEE 754-2008 compliant, i.e., it supports all rounding modes, exception handling, overflow, and gradual underflow. Particularly, the latter requires a significant overhead and is therefore often omitted.

The outline is given as follows: Section II shortly describes the fixed-point multiplier. Section III introduces the decimal floating-point multiplication followed by a description of the proposed architecture in detail. Post-place & route results are presented in section IV and finally in section V the main contributions of this paper are summarized.

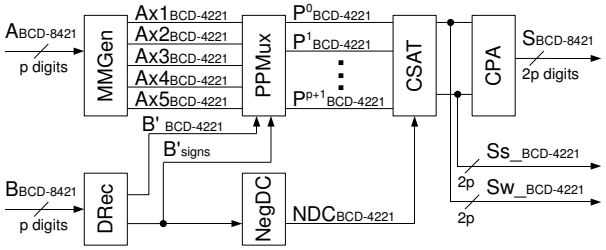

Fig. 1. Parallel fixed-point multiplier

II. Decimal Fixed-Point Multiplier

The Decimal Fixed-Point Multiplier (DFixMul) computes the product $A \cdot B$ of the unsigned decimal multiplicand A and multiplier B, both with the same precision p. It is fully combinational and can be pipelined. In particular, it is based on BCD recoding schemes, fast partial product generation, and a BCD-4221 Carry Save Adder (CSA) reduction tree. It is optimized for use on Xilinx Virtex-5 FPGAs.

A decimal number Z is called BCD-$X_1X_2X_3X_4$ coded when Z can be expressed by (1).

$$Z = \sum_{i=0}^{p-1} Z_i \cdot 10^i, \quad Z_i = \sum_{k=0}^{3} Z_{ik} \cdot X_i, \quad Z_{ik} \in \{0,1\} \quad (1)$$

Time-critical components are BCD-8421 Carry Propagation Adders (CPA) that are used in the generation of the multiplicand multiples as well as for final addition. The adders proposed in [3] are designed and placed on slice-level, considering a minimum carry chain length and least possible propagation delays.

The DFixMul presented in this paper is designed for Virtex5 devices and is based on a previously published work [1], which was optimized for Virtex-II Pro FPGAs. The architectures are very similar. However, the low-level components are different, i.e., CSA, CPA, and the Multiplicand Multiples Generator (MMGen). Generally, the fixed-point multiplier consists of six functional blocks as depicted in Fig. 1. The basic idea is to generate $p + 1$ partial products and to sum them up, which is performed by the parallel Carry Save Adder Tree (CSAT) and the final BCD-8421 CPA. The CSAT is based on (3:2) CSA blocks for BCD-4221 format. The partial products are

1946-1488/10 $26.00 © 2010 IEEE
DOI 10.1109/FPL.2010.98

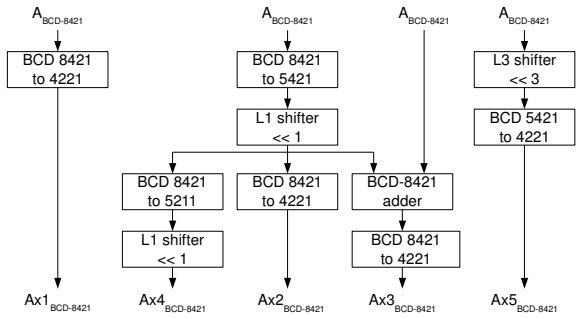

Fig. 2. Multiplicand multiples generator (MMGen)

the multiplicand's multiples and are selected via the Partial Product Multiplexer (PPMux). Due to the multiplier recoding that transforms the multiplier's digit set $\{0,\ldots,9\}$ into the signed digit set $\{-5,\ldots,5\}$, as well as a simple method to handle negative partial sums (10's complement), only five multiples (A×1, A×2, A×3, A×4, A×5) have to be generated by the (MMGen) a priori.

The MMGen exploits the correlation between shift operation and constant value multiplication [2]. Thus, the multiples $A \times 2$, $A \times 4$, and $A \times 5$ can be easily computed by digit recoding and constant shift operations. A recoding is very fast and consumes two (6:2) LUTs per digit, whereas a constant shift operation costs nothing because it is just a renaming of signals. Hence, all multiples can be easily generated by simple shift operations and digit recodings, only $A \times 3$ requires an additional CPA, as depicted in Fig. 2.

The Decimal Recoding Unit (DRec) shown in Fig. 1 transforms each multiplier's digit B_k from the digit set $\{0,\ldots,9\}$ into the signed digit set $\{-5,\ldots,5\}$. This recoding increases the number of partial products by one $(p+1)$ but gets along without any ripple carry. Hence, it is a very fast operation.

Since the multiplier's output is of length $2p$ but one single partial product is of length p, for 10's complement generation each partial product has to be extended and if required padded with nines. To keep the input length of CSAT short, the Negative Digits Correction Unit (NegDC) combines the paddings of all partial products in a single word and passes it to the CSAT. This is feasible because adding several words composed of leading nines and following zeros always yields to a decimal word composed of only 0, 8, and 9. Moreover, the position of the nines and eights can be calculated very fast by using the FPGA's fast carry chain, see [1].

The reduction of the partial products is performed by a (n:2) CSA tree. The tree is composed of parallel and consecutively wired BCD-4221 (3:2) CSAs, that add three BCD-4221 digits yielding a sum and a carry digit, both of BCD-4221 coding scheme. The $n = p + 2$ decimal words are composed of $p + 1$ partial products and one summand that regards the sign paddings, as described previously. The redundant carry-save format of the CSAT can be further reduced by a carry propagation adder of length $2p$ to obtain a unique result.

Several pipeline stages can be optionally implemented by means of VHDL generic switches. For a 16×16 digits multiplier this is one possible pipelining stage to buffer the input words, three for the MMGen, one for PPMux, six for the CSAT (one for each reduction stage), and two for the final BCD-8421 conversion and CPA. Altogether, these are 11 possible pipeline stages for the BCD-4221 carry-save format output and 13 stages for the final BCD-8421 carry-propagation format output.

III. Decimal Floating-Point Multiplier

Similar to Binary Floating-Point (BFP) numbers, a Decimal Floating-Point (DFP) number D is composed of the sign bit s, the non-negative integer significand C, and the biased non-negative integer exponent: $D = (-1)^s \cdot C \cdot 10^{E-bias}$. The biased exponent variable E in this paper relates to IEEE 754-2008's q in the following way: $q = E - bias$.

Unlike the BFP format, the DFP format's significand is non-normalized. This leads to a redundancy, i.e., a decimal number might have multiple representations. The set of representations is called the floating-point number's *cohort*. For example $1234000 \cdot 10^0$, $123400 \cdot 10^1$, and $1234 \cdot 10^3$ are members of the same cohort.

DFP numbers are encoded in three fields: a sign field s, a combination field G, and a trailing significand field T. Two fixed-width basic formats for DFP numbers (*decimal64* and *decimal128*) are specified in IEEE 754-2008 and are provided in Table I. The combination field G encodes the biased exponent, the most significant digit (MSD), and informations about infinity and Not a Number (NaN). The trailing significand is either encoded via Densely Packed Decimal (DPD) algorithm or as an unsigned binary integer. The floating-point multiplier presented in this paper satisfies the *decimal64* interchange format with DPD encoding.

TABLE I
DFP format parameters

		decimal64	**decimal128**
storage width [bits]	*k*	64	128
precision [digits]	p	16	34
sign [bits]	s	1	1
combination filed [bits]	G	13	17
trailing significand [bits]	T	50	110
maximum exponent	q_{max}	369	6111
minimum exponent	q_{min}	-398	-6176
exponent bias	*bias*	398	6176

In contrast to decimal fixed-point multipliers, there are only a few papers presenting designs for decimal floating-point multiplication that are fully or partly in compliance with IEEE 754-2008. Erle et al. [4] describe an iterative as well as a parallel DFP multiplier that include early estimation of the Shift-Left Amount (SLA) and efficient decimal rounding. Their proposed iterative DFP multiplier can be extended to support gradual underflow but their parallel DFP cannot. However, the underflow feature increases the iterative multiplier's maximum latency significantly. In [5] is presented a DFP multiplier that is implemented on an FPGA but does not

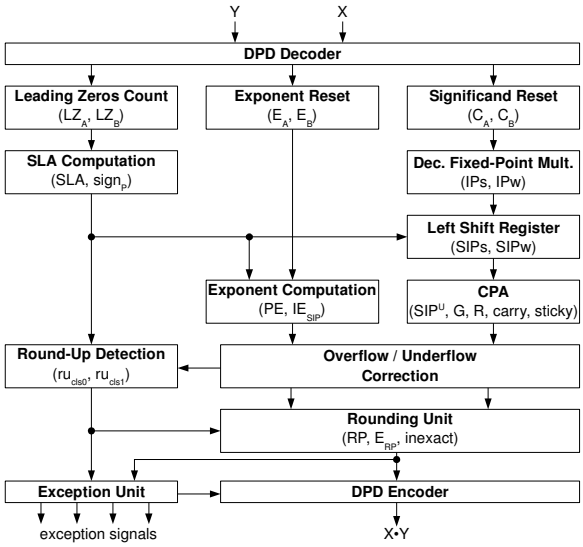

Fig. 3. Block diagram of DFP multiplier

support all rounding modes, gradual underflow, and exception handling as required by IEEE 754-2008. The DFP multiplier presented in this paper is fully compliant with IEEE 754-2008 for the *decimal64* format. It is combinational and thus can be pipelined to increase throughput. It applies the concept of early estimation of the SLA and efficient decimal rounding as proposed by Erle et al. [4].

A. Proposed Parallel Floating-Point Multiplier

The Decimal Floating-Point Multiplier (DFlMul) proposed in this paper is an extension of the DFixMul. A block diagram is depicted in Fig. 3. A DFP multiplication begins with the decoding of the input operands and the extraction of the signs, the significands, and the exponents. If one of the operands is a signaling NaN (sNAN) or quiet NaN (qNAN), then the result is also a quiet NaN with the payload of the original NaN. Hence, in order to preserve the NaN payload, the exponent and significand reset units revise the non-NaN operand's significand and exponent to one and zero, see Fig. 4. Additionally, the exponents of operands that are equal to zero are reset in order to prevent of possible exceptions because of exponent overflow.

1) significand reset:
   if $(\overline{\text{qNAN}_\text{A}} \wedge (\text{qNAN}_\text{B} \vee \text{sNAN}_\text{B})) \vee (\text{qNAN}_\text{A} \wedge \text{sNAN}_\text{B})$ then
      $\text{significand}_\text{A} = 1$ and $\text{significand}_\text{B}$ remains unchanged
   else if $(\text{qNAN}_\text{A} \wedge \overline{\text{sNAN}_\text{B}}) \vee \text{sNAN}_\text{A}$ then
      $\text{significand}_\text{B} = 1$ and $\text{significand}_\text{A}$ remains unchanged
   else
      $\text{significand}_\text{A}$ and $\text{significand}_\text{B}$ remain unchanged
2) exponent reset:
   if $(\overline{\text{qNAN}_\text{A}} \wedge (\text{qNAN}_\text{B} \vee \text{sNAN}_\text{B})) \vee (\text{qNAN}_\text{A} \wedge \text{sNAN}_\text{B}) \vee \text{significand}_\text{A} = 0$ then
      $\text{exponent}_\text{A} = 0$ and exponent_B remains unchanged
   else if $(\text{qNAN}_\text{A} \wedge \overline{\text{sNAN}_\text{B}}) \vee \text{sNAN}_\text{A} \vee \text{significand}_\text{B} = 0$ then
      $\text{exponent}_\text{B} = 0$ and exponent_A remains unchanged
   else
      exponent_A and exponent_B remain unchanged

Fig. 4. Algorithm for exponent and significand reset

In the next step the decimal fixed-point multiplication is calculated. The result are two $2p$-digit BCD-4221 coded words. We first do not use a $2p$-digit CPA to eliminate this redundancy because summing up the result after left shifting (so-called delayed CPA) leads to a CPA of length $p+2$ only. But later, we will also consider an implementation which uses the fixed-point multiplier's unique CPA output and we will compare the implementation results with those of the delayed CPA design. In parallel to multiplication, the significands are examined and their leading zeros (LZ_A, LZ_B) are counted to determine the SLA. Moreover, the Intermediate Exponent of the Shifted Intermediate Result (IE_SIP) is calculated by the exponent computation unit and the signs are XORed to determine the product's sign (sign_P). According to the SLA, the multiplier's outputs IPs and IPw are left-shifted and afterwards summed up. Furthermore, the p-digit upper word of the shifted intermediate product (SIP^U), the guard digit (G), the round digit (R), the sticky bit (sb), and a carry signal are determined. Depending on the current IE_SIP, the overflow and underflow correction unit can insert a corrective right shift when gradual underflow occurs. Likewise, a corrective left shift is necessary when IE_SIP exceeds the maximum exponent while SIP^U has still leading zeros. If the number of essential digits in the intermediate product is greater than the precision p or if gradual underflow occurs, then rounding is required. The Rounding Unit (RU) computes the rounded product (RP), the exponent of the rounded product (E_RP), and an inexact signal that is asserted when accuracy is lost due to rounding. Finally, the result is encoded again considering sign_P, RP, E_RP, infinity and NaN. The Exception Unit (ExU) might assert additional exception signals, i.e., invalid operation (result is NaN), inexact (a rounding has been performed), overflow, and underflow. In the following, the SLA computation and exponent computation, shifting, rounding, overflow and underflow correction, and exception generation are explained in detail and an example is given.

The proposed architecture uses the concept of early estimation of SLA [4] which is calculated simultaneously to fixed-point multiplication and reduces the maximum number of left shift positions during succeeding rounding to one. The SLA ranges from 0 to p and is computed based on LZ_A and LZ_B.

$$\text{SLA} = min(\text{LZ}_\text{A} + \text{LZ}_\text{B},\ p) \tag{2}$$

The SLA is used to shift the p most significant digits of the Intermediate Products (IP_s and IP_w) into the upper word and to perform rounding according to the least significant digits. Two solutions for parallel shift registers are analyzed. The first one uses multiplexers and the second one applies the hard-wired multipliers of DSP48E slices. The latter is possible because an Lk-shift complies with a multiplication by 2^k. The multiplier-based shift register has the advantage that it saves LUTs and the number of leading zeros in power representation ($\text{LZ}_\text{A}^\text{pow}$, $\text{LZ}_\text{B}^\text{pow}$) can be calculated very fast by means of the FPGA's carry logic, see Fig. 5. SLA in power representation ($\text{SLA}^\text{pow} = 2^\text{SLA}$) is then computed by a multiplication: $\text{SLA}^\text{pow} = min(\text{LZ}_\text{A}^\text{pow} \cdot \text{LZ}_\text{B}^\text{pow}, 2^p)$. Otherwise, when using

the multiplexer-based shift register the power representations have to be transformed into binary representation and added afterwards, see (2).

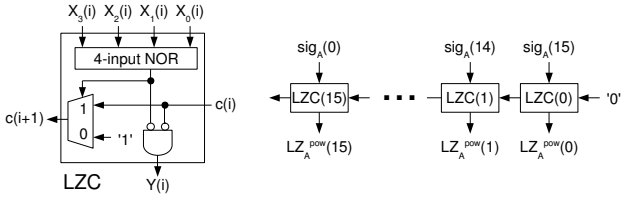

Fig. 5. Leading zeros counter

Due to the IEEE 754-2008 DFP format has multiple representations of the same value, the concept of preferred exponents (PE) were introduced. For multiplication the PE is

$$\mathrm{PE} = \mathrm{E_A} + \mathrm{E_B} - \mathrm{bias}. \tag{3}$$

Prior to rounding, the fixed-point multiplier's MSDs are shifted into the upper word of length p and the lower word is used for round-up detection. Thus, $\mathrm{IE}^{\mathrm{SIP}}$ is calculated by means of PE and SLA [4]:

$$\begin{aligned} \mathrm{IE_{SIP}} &= \mathrm{PE} + p - \mathrm{SLA} \\ &= \mathrm{E_A} + \mathrm{E_B} - \mathrm{bias} + p - \mathrm{SLA}. \end{aligned} \tag{4}$$

For the DFixMul two solutions are analyzed. The first uses the redundant carry-save output and applies a CPA *after* shifting. The second directly uses the reduced carry-propagation output of the fixed-point multiplier. The drawback of the carry-save output solution is the doubled resource usage for the left shift register. On the other hand, the utilization of the CPA after the operands are left-shifted reduces the maximum CPA's length to $p + 2$ instead of $2p$, i.e., the addition of length $2p$ can be subdivided into two parallel additions of the lower and upper parts. The lower part is only required to calculate the sticky and carry bits, where the sticky bit indicates if any of the digits beyond the round digit is nonzero. Fig. 6 depicts the CPA for the carry-save output solution.

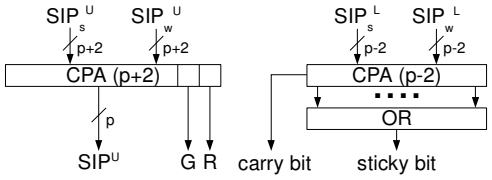

Fig. 6. CPA for carry-save output solution

Rounding is required when all significant digits of the fixed-point product cannot be placed in a single word of length p. Rounding either selects the shifted intermediate product truncated to p digits or its incremented value. IEEE 754-2008 standardizes five rounding modes. These are *roundTiesToEven*, *roundTiesToAway*, *roundTowardPositive*, *roundTowardNegative*, and *roundTowardZero*. The selection of the truncated intermediate product ($TP^{+0} = SIP^U$) or incremented truncated intermediate product ($TP^{+1} = SIP^U + 1$) is determined by the Round-Up Detection Unit (RUpD). It depends on the rounding mode, $\mathrm{sign_P}$, $\mathrm{SIP^U}$, G, R, sticky bit, and the carry signal. Due to early estimation of the SLA the possibility of a final left shift by one position arises. Thus, for a given rounding mode the RUpD calculates two different round-up values, ru^{cls0} and ru^{cls1}, in case of no corrective left shift and a corrective left shift by one, respectively. The $\mathrm{IE_{SIP}}$ is decremented by one when a corrective left shift has to be applied. The computations of ru^{cls0} and ru^{cls1} are listed in Table II.

TABLE II
ROUND-UP COMPUTATION

RoundTiesToEven	$ru^{cls0} = (G > 5) + (G = 5) \cdot [l + (R > 0) + c + sb] + (G = 4) \cdot (R = 9) \cdot c \cdot [l + sb]$ $ru^{cls1} = (R > 5) + (R = 5) \cdot [c + lg + sb] + (R = 4) \cdot c \cdot [lg + sb]$
RoundTiesToAway	$ru^{cls0} = (G \geq 5) + (G = 4) \cdot (R = 9) \cdot c$ $ru^{cls1} = (R \geq 5) + (R = 4) \cdot c$
RoundTowardPositive	$ru^{cls0} = \overline{sign} \cdot [(G > 0) + (R > 0) + c + sb]$ $ru^{cls1} = \overline{sign} \cdot [(R > 0) + c + sb]$
RoundTowardNegative	$ru^{cls0} = sign \cdot [(G > 0) + (R > 0) + c + sb]$ $ru^{cls1} = sign \cdot [(R > 0) + c + sb]$
RoundTowardZero	$ru^{cls0} = (G = 9) \cdot (R = 9) \cdot c$ $ru^{cls1} = (R = 9) \cdot c$

Legend: G=guard digit, R=round digit, c=carry bit, sb=sticky bit, l=LSB of $\mathrm{TP^{+0}}$, lg = LSB of G, '+'= logical OR, '·'=logical AND

In this design no rounding overflow can occur, which has been proven by Erle et al. [4]. A rounding overflow might arise when the SIP^U is incremented due to rounding and a carry out of the MSD position occurs. The rounding algorithm is explicitly described in [4]. However, the algorithm presented in this paper slightly differs due to an additional carry signal. Fig. 7 summarizes the calculations of the Rounded Product (RP), the Exponent of the Rounded Product ($\mathrm{E_{RP}}$), and the inexact flag that is asserted whenever the RP is inexact due to rounding. The algorithm simply chooses between $\mathrm{TP^{+0}}$, $\mathrm{TP^{+1}}$, or these values left shifted by one digit with either G or G+1 concatenated.

The Overflow/Underflow Correction Unit (OUC) handles overflow and gradual underflow. When $(\mathrm{q_{max}} + p) > \mathrm{IE_{SIP}} > \mathrm{q_{max}}$ and SIP has sufficient leading zeros, then $\mathrm{IE_{SIP}}$ is set to the maximum exponent and the significand is corrected by a left shift. On the other hand, when $(\mathrm{q_{min}} - p) < \mathrm{IE_{SIP}} < \mathrm{q_{min}}$, then the significand is right-shifted and the exponent is increased up to $\mathrm{q_{min}}$. This is also called gradual underflow. The OUC has to be performed before rounding, otherwise in case of gradual underflow the result might be computed incorrectly, which is illustrated in the following simple example with rounding mode *RoundTiesToEven* $\bigcirc$, $A = 3461713317126677 \cdot 10^{-363}$ and $B = 4686509784478936 \cdot 10^{-54}$. The exact result is $A \cdot B = 1622335333177520.5657159150175672 \cdot 10^{-417}$. If the rounding is performed before gradual underflow correction (GUC), it leads to the wrong result (5). On the other hand, if the GUC is accomplished before rounding, the correct result

$GRsbc = (G > 0) \vee (R > 0) \vee sb \vee c$
$Rsbc = (R > 0) \vee sb \vee c$

Case 1: (MSD of TP^{+0}) > 0 and (MSD of TP^{+1}) > 0
- a) if $(ru^{cls0} = 0)$ then
  $RP = TP^{+0}$
  $E_{RP} = IE_{SIP}$, inexact= $GRsbc$
- b) if $(ru^{cls0} = 1)$ then
  $RP = TP^{+1}$
  $E_{RP} = IE_{SIP}$, inexact= $GRsbc$

Case 2: (MSD of TP^{+0}) $= 0$ and (MSD of TP^{+1}) $= 0$
- a) if $(GRsbc = 0)$ then
  if $(IE_{SIP} = PE)$ or $(IE_{SIP} \leq q_{min})$ then
  $RP = TP^{+0}$
  $E_{RP} = IE_{SIP}$, inexact= $GRsbc$
  if $(IE_{SIP} > PE)$ and $(IE_{SIP} > q_{min})$ then
  $RP = \{(TP^{+0} \ll 1), G\}$
  $E_{RP} = IE_{SIP} - 1$, inexact= $Rsbc$
- b) if $(GRsbc = 1)$and$(IE_{SIP} \leq q_{min})$ then
  if $(ru^{cls0} = 0)$ then
  $RP = TP^{+0}$
  $E_{RP} = IE_{SIP}$, inexact= $GRsbc$
  if $(ru^{cls0} = 1)$ then
  $RP = TP^{+1}$
  $E_{RP} = IE_{SIP}$, inexact= $GRsbc$
- c) if $(GRsbc = 1)$and$(IE_{SIP} > q_{min})$ then
- c.1) if $(ru^{cls1} = 0)$ then
  $RP = \{(TP^{+0} \ll 1), G\}$
  $E_{RP} = IE_{SIP} - 1$, inexact= $Rsbc$
- c.2) if $(ru^{cls1} = 1)$ then
  if $(G < 9)$ then
  $RP = \{(TP^{+0} \ll 1), (G + 1)\}$
  $E_{RP} = IE_{SIP} - 1$, inexact= $Rsbc$
  if $(G = 9)$ then
  $RP = \{(TP^{+1} \ll 1), 0\}$
  $E_{RP} = IE_{SIP} - 1$, inexact= $Rsbc$

Case 3: (MSD of TP^{+0}) $= 0$ and (MSD of TP^{+1}) > 0
- a) same as in Case 2
- b) same as in Case 2
- c.1) same as in Case 2
- c.2) if $(ru^{cls1} = 1)$ then
  $RP = TP^{+1}$
  $E_{RP} = IE_{SIP}$, inexact= $GRsbc$

Fig. 7. DFP rounding algorithm

(6) is achieved:

$$\bigcirc (A \cdot B) = 1622335333177521 \cdot 10^{-417}$$
$$GUC(\bigcirc (A \cdot B)) = 0001622335333177 \cdot 10^{-398} \quad (5)$$
$$GUC(A \cdot B) = 0001622335333177.520565... \cdot 10^{-398}$$
$$\bigcirc GUC(A \cdot B) = 0001622335333178 \cdot 10^{-398} \quad (6)$$

The OUC algorithm is described in Fig. 8.

if $(q_{max} + p > IE_{SIP} > q_{max})$ then
  shift $= IE_{SIP} - q_{max}$
  $IE_{SIP}^{O/U} = IE_{SIP} -$ shift
  $SIP^{O/U} = (SIP \ll$ shift$)(2p - 1 : 0)$
  if $(SIP^{O/U}(3p - 1) \vee \ldots \vee SIP^{O/U}(2p)) = 1$ then
    overflow $= 1$
else if $(q_{min} - p < IE_{SIP} < q_{min})$ then
  shift $= q_{min} - IE_{SIP}$
  $IE_{SIP}^{O/U} = IE_{SIP} +$ shift
  $SIP^{O/U} = (SIP \gg$ shift$)(3p - 1 : p)$
  if $(SIP^{O/U}(p - 1) \vee \ldots \vee SIP^{O/U}(0)) = 1$ then
    inexact $= 1$
  if $(SIP^{O/U}(3p - 1) \vee \ldots \vee SIP^{O/U}(2p)) = 0$ then
    underflow $= 1$

Fig. 8. Overflow/underflow correction algorithm

$A = 0000000123456789 \cdot 10^{-2}$, $B = 0000001000000001$
1) SLA computation:
   $LZ_A = 7$, $LZ_B = 6 \Rightarrow$ SLA $= min(13, 16) = 13$
2) IE_{SIP} computation:
   $E_A = -2 + 398 = 396$, $E_B = -7 + 398 = 391$
   $\Rightarrow$ PE $= 396 + 391 - 398 = 389 \Rightarrow$ $IE_{SIP} = 389 + 16 - 13 = 392$
3) Calculation of TP^{+0}, TP^{+1}, G, R, sb, and c:
   $\text{significand}_A \cdot \text{significand}_B = 00000000000000123456789123456789$
   $\Rightarrow$ SIP $= 0123456789123456\ 7890000000000000$
   $\Rightarrow$ $TP^{+0} = 0123456789123456$, $TP^{+1} = 0123456789123457$
   G=7, R=8, sb='1', c='0'
4) No exponent underflow or overflow
5) Rounding (RoundUpTiesToEven) with case 2.c.2, see Fig. 7:
   ru^{cls0}='1', ru^{cls1}='1'
   $\Rightarrow$ RP $= 1234567891234568$, $E_{RP} = 392 - 1 = 391 = -7 + 398$
   $\Rightarrow A \cdot B = 1234567891234568 \cdot 10^{-7}$

Fig. 9. Example of a decimal64 multiplication with roundTiesToEven mode

An example that illustrates the algorithm of the DFP multiplier is depicted in Fig. 9.

The ExU generates the four exception signals *invalid operation*, *inexact*, *overflow*, and *underflow* according to IEEE 754-2008. An *invalid operation* emerges when one of the operands is a signaling NaN or one operand is zero and the other one is infinity. The *inexact* signal is asserted whenever the result must be rounded. The overflow exception is signaled when a result's magnitude exceeds the largest finite number and the underflow exception is signaled when a result is both tiny and inexact. Tininess is when the result's magnitude is between zero and the smallest normal number.

The decimal floating-point multiplier presented in this paper has been implemented in three different variants. Both, the first one (type 1) and second one (type 2) use the fixed-point multiplier's redundant carry-save output. However, type 1 applies DSP48E slices as shift registers, whereas the shift registers of type 2 are multiplexer-based. Type 1 and 2 require an additional CPA of length $p + 2$ after left shifting the intermediate result. The third modification (type 3) uses the fixed-point multiplier's unique carry-propagation output of length $2p$ and multiplexer-based shift registers. The implementation results for this three variants are presented in section IV.

The floating-point multipliers are designed to support pipelining which can be controlled via VHDL generic switches. The type 1 multiplier can be subdivided into 24 stages, the type 2 and type 3 multiplier into 22 stages.

IV. Implementation Results

All circuits are modeled using VHDL. For synthesis and implementation Xilinx ISE 10.1 has been used. The 16 digits DFixMul and the decimal64 DFlMul have been implemented for Xilinx Virtex-5 devices with speed grade -2. The fixed-point multiplier with CPA output has been implemented for several pipeline configurations, see Table IV. The results show that the minimum overall latency of about 17.5 ns can be achieved without any pipeline registers and the best operating frequency of 229 MHz can be obtained with 10 pipeline registers. However, using 6 or more pipeline registers does not reduce the longest path delay significantly but increases the overall latency instead.

TABLE III
POST-PLACE & ROUTE RESULTS FOR DECIMAL FLOATING-POINT MULTIPLIERS TYPE 1, TYPE 2, AND TYPE 3

#pipe-line regs.	type 1 (mul-based shifting, delayed CPA)					type 2 (mux-based shifting, delayed CPA)					type 3 (mux-based shifting, no delayed CPA)				
	delay [ns]	latency [ns]	#LUT	#LUT +FF	#FF	delay [ns]	latency [ns]	#LUT	#LUT +FF	#FF	delay [ns]	latency [ns]	#LUT	#LUT +FF	#FF
0	40.0	40.0	8371	8371	0	35.0	35.0	9067	9067	0	35.9	35.9	8201	8201	0
1	20.0	40.0	8145	8277	31	18.0	35.9	9368	9446	284	19.0	38.0	7490	7622	299
2	16.9	51.0	6294	6452	463	14.0	41.9	8427	8636	711	12.6	37.8	9421	9570	830
3	14.0	55.9	6703	7089	1463	11.0	43.9	8484	8768	1984	11.0	43.9	9042	9364	1983
4	10.0	49.9	7264	7489	1604	9.0	44.9	8791	9018	1253	9.0	44.9	8984	7966	1198
5	9.0	53.8	7216	7526	1201	8.5	50.9	8007	8240	1254	8.7	52.1	7836	8066	1805
6	8.0	55.8	6690	7004	1424	7.7	53.7	7886	8285	2437	7.5	52.3	7816	7985	2422
7	7.5	59.8	6734	7159	1780	6.6	52.6	8022	8385	2706	6.7	53.4	7938	8165	2895
8	7.2	64.5	7339	7697	2575	6.0	53.8	8046	8409	2529	6.2	56.0	7963	8433	3492
9	6.1	60.8	7490	7925	2275	5.5	54.7	8057	8417	2679	6.1	60.8	7987	8494	2416
10	6.0	65.7	7521	8085	2991	5.3	58.6	8087	8667	3373	5.7	62.5	7981	8459	2549
11	5.8	69.3	7550	8119	3729	5.4	64.5	8087	8690	3441	5.4	64.5	7999	8521	3599
12	5.5	71.2	7500	8257	3497	5.3	68.6	8160	8868	3890	5.3	68.6	7989	8824	4106
13	5.3	73.9	7527	8324	4046	5.2	72.4	8191	8844	4237	5.3	73.8	7565	8302	4002
14	5.4	80.6	7530	8275	4121	5.2	77.6	8207	9255	4549	5.3	79.9	7671	8476	4390
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

Note: For more than 14 pipeline registers the delay values do not differ significantly.

TABLE IV
POST-PLACE & ROUTE RESULTS FOR DFIXMUL WITH CPA OUTPUT

#pipeline registers	delay [ns]	max. freq. [MHz]	#LUT	#LUT +FF	#FF	latency [ns]
0	17.47	57	6514	6514	0	17.96
1	10.97	91	5873	5974	1185	21.12
2	7.98	125	5347	5479	778	23.94
3	5.97	168	6028	6149	1192	23.88
4	5.48	183	6198	6438	1821	27.40
5	4.83	207	6369	6664	2027	28.74
6	4.46	225	5575	5992	3216	30.66
7	4.52	221	5574	6216	3282	36.16
8	4.48	223	5576	6115	3478	39.33
9	4.42	226	5610	6561	3877	44.80
10	4.36	229	6138	6989	4140	47.08
11	4.44	225	6283	7149	4350	53.64
12	4.44	225	6318	7271	4649	58.11
13	4.44	225	6519	7574	4917	62.72

TABLE V
AREA BREAKDOWN OF THE DECIMAL FLOATING-POINT MULTIPLIER

component	type 1 #LUT / DSP		type 2 #LUT / DSP		type 3 #LUT / DSP	
LZ count+SLA comp.	140	1	150	0	150	0
DPD decoder	266	0	266	0	266	0
DPD encoder	210	0	210	0	210	0
left shift register	0	16	948	0	948	0
CPA	499	0	499	0	0	0
overfl./underfl. correct.	386	8	1146	0	1146	0
round-up det.+rounding	346	0	346	0	346	0
DFiMul	see table IV					

The three floating-point multiplier's variants (type 1, type 2, type 3) have been implemented with various pipeline stages. The post-place & route results are summarized in Table III. The results show that the minimum overall latency of 35.0 ns can be achieved by the type 2 variant without any pipeline stages. The best delay of 5.2 ns and thus the highest operating frequency of over 192 MHz can be obtained with the type 2 multiplier using 13 pipeline stages. The implementation results for type 2 and type 3 are very similar, hence, the benefits of a delayed CPA with reduced length are not as advantageous as expected and even uses more LUTs. Type 1 consumes least LUTs, but 25 DSP48E slices are additionally required. For each DSP48E slice two configurable pipeline registers can be applied using the attributes *MREG* and *PREG*. Anyway, type 1 is always slower than type 2 or type 3. The area breakdown per main component is listed in Table V. Most of the LUTs are used by the the DFixMul and by shift registers, which are employed in the left shift register block and the overflow/underflow correction unit. However, the type 1 multiplier uses less LUTs because it implements these shift registers by use of DSP48E slices. The best placements for insertion of pipeline registers have been determined empirically. Thus, the position and number of pipeline registers can vary for two successive pipeline configuration and the number of flip flops does not increase monotonically with increasing number of pipeline stages. Furthermore, the number of used LUT differs, although it should be independent of the number of pipeline stages. That is because for each pipeline configuration the whole design is synthesized and implemented including speed optimization. However, speed optimization encompasses register balancing, resource sharing and logic replication, which have an impact on the final LUT utilization.

The comparison of our proposed designs with other designs is complicated because to the best of our knowledge no comparable designs have been published yet. There are architectures such as [4] but they are designed for ASICS. On the other hand, besides the design proposed in this paper, the only two FPGA-based decimal floating-point multipliers are presented in [5]. Both are implemented for Virtex4 devices. The first one is optimized for low latency and uses 3494 slices, 1713 flip flops, 5961 LUTs, and 32 Block SelectRAMs (BRAM). It has an overall latency of 66 ns. The second one is optimized for low area and uses 1222 slices, 1123 flip flops, 1966 LUTs, and 8 BRAMs. Its overall latency is 161.2 ns.

TABLE VI
POST-PLACE & ROUTE RESULTS FOR 64 BIT BINARY FLOATING-POINT MULTIPLIER GENERATED WITH COREGEN

64 bit binary floating-point multiplier without DSP48E						
#pipeline registers	delay [ns]	max. freq. [MHz]	#LUT	#LUT +FF	#FF	latency [ns]
0	16.0	62.5	2655	2655	0	16.0
3	7.5	133.9	2637	2654	762	22.5
6	4.2	239.8	2284	2297	1989	25.2
9 (max)	3.5	286.5	2288	2476	2434	31.5

64 bit binary floating-point multiplier with 9 DSP48E						
#pipeline registers	delay [ns]	max. freq. [MHz]	#LUT	#LUT +FF	#FF	latency [ns]
0	22.8	43.9	221	349	0	22.8
5	5.0	198.9	205	259	207	25.0
10	3.6	275.2	347	473	444	36.0
15 (max)	2.2	456.8	398	574	542	33.0

64 bit binary floating-point multiplier with 11 DSP48E						
#pipeline registers	delay [ns]	max. freq. [MHz]	#LUT	#LUT +FF	#FF	latency [ns]
0	22.9	43.7	159	277	0	22.9
5	5.4	186.1	133	251	165	27.0
10	3.5	285.1	236	414	388	35.0
16 (max)	2.0	506.1	309	518	491	32.0

However, a comparison is limited because it is based on a Virtex4 device and does not implement all rounding modes, exception handling, and gradual underflow.

To compare our design with multiplier designs implemented for the same FPGA chip, we have analyzed a binary 64 bit floating-point multiplier on a Virtex-5 provided by the Xilinx Core Generator. The binary floating-point multiplier can be implemented using 0, 9, 10, or 11 DSP48E slices. Furthermore, the number of implemented pipeline registers can be adjusted. The implementation results are summarized in Table VI. Compared to the binary floating-point multiplier without DSP48E usage, the DFlMul proposed in this paper uses 3.0-3.5 times more LUTs and has a 1.8-2.2 times higher latency. However, it must taken into account that decimal floating-point multiplication has a much greater overhead than binary floating-point multiplication.

Considering a medium Xilinx Virtex5 FPGA device such as XC5VLX110T and an average LUT usage of 8000-9000 LUTs, 11.5% - 13% of the area is then occupied and leaves enough area for implementations of decimal floating-point addition, subtraction and division. Hence, a complete IEEE 754-2008 compliant decimal floating-point co-processor for decimal64 data types might fit into a single FPGA chip.

V. CONCLUSION

In this paper we extended a previously published decimal fixed-point multiplier to a decimal floating-point multiplier that maps onto FPGA architectures (in particular Virtex 5 devices) and allows the implementation of an IEEE 754-2008 compliant co-processor. The design is fully parallel and can be pipelined by means of configurable pipeline stages. We compared different implementations, including multiplier-based shift registers and delayed CPAs. Finally, we analyzed the performance with respect to the number of pipeline stages and we presented implementation results that are useful to trade off overall latency against longest path delay. Summarizing these values, we achieved a decimal floating-point multiplication within 35 ns and obtained a maximum operating frequency of 192 MHz using 13 pipeline stages.

REFERENCES

[1] Malte Baesler and Thomas Teufel. FPGA Implementation of a Decimal Floating-Point Accurate Scalar Product Unit with a Parallel Fixed-Point Multiplier. *Reconfigurable Computing and FPGAs, International Conference on*, 0:6–11, 2009.

[2] A. Vazquez, E. Antelo, and P. Montuschi. A new family of high-performance parallel decimal multipliers. In *18th IEEE Symposium on Computer Arithmetic*, pages 195–204, June 2007.

[3] M. Vazquez and G. Sutter and G. Bioul and J. P. Deschamps. Decimal Adders/Subtractors in FPGA: Efficient 6-input LUT Implementations. *Reconfigurable Computing and FPGAs, International Conference on*, 0:42–47, 2009.

[4] Mark A. Erle and Brian J. Hickmann and Michael J. Schulte. Decimal Floating-Point Multiplication. *IEEE Transactions on Computers*, 58(7):902–916, 2009.

[5] Carlos Minchola and Gustavo Sutter. A FPGA IEEE-754-2008 Decimal64 Floating-Point Multiplier. *Reconfigurable Computing and FPGAs, International Conference on*, 0:59–64, 2009.