Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2010, Article ID 357839, 13 pages
doi:10.1155/2010/357839

Research Article

A Decimal Floating-Point Accurate Scalar Product Unit with a Parallel Fixed-Point Multiplier on a Virtex-5 FPGA

Malte Baesler, Sven-Ole Voigt, and Thomas Teufel

Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, 21073 Hamburg, Germany

Correspondence should be addressed to Malte Baesler, malte.baesler@tu-harburg.de

Received 26 February 2010; Revised 1 October 2010; Accepted 20 November 2010

Academic Editor: Viktor K. Prasanna

Decimal Floating Point operations are important for applications that cannot tolerate errors from conversions between binary and decimal formats, for instance, commercial, financial, and insurance applications. In this paper, we present a parallel decimal fixed-point multiplier designed to exploit the features of Virtex-5 FPGAs. Our multiplier is based on BCD recoding schemes, fast partial product generation, and a BCD-4221 carry save adder reduction tree. Pipeline stages can be added to target low latency. Furthermore, we extend the multiplier with an accurate scalar product unit for IEEE 754-2008 *decimal64* data format in order to provide an important operation with least possible rounding error. Compared to a previously published work, in this paper, we improve the architecture of the accurate scalar product unit and migrate to Virtex-5 FPGAs. This decreases the fixed-point multiplier's latency by a factor of two and the accurate scalar product unit's latency even by a factor of five.

1. Introduction

Financial calculations are usually carried out using decimal arithmetic, because the conversion between decimal and binary numbers introduces unacceptable errors that may even violate legal accuracy requirements [1]. Therefore, commercial application often use nonstandardized software to perform decimal floating-point arithmetic. These software implementations are usually 100 to 1000 times slower than equivalent binary floating-point operations in hardware [1]. 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 [2] that offers a more profound specification than the former Radix-Independent Floating Point Arithmetic IEEE 754-1987 [3]. Therefore, new efficient algorithms have to be investigated, and providing hardware support for decimal arithmetic is becoming more and more a topic of interest. However, most modern microprocessors still lack of support for decimal floating-point arithmetic, because additional hardware is costly. The POWER6 is the first microprocessor with implementing the IEEE 754-2008 decimal floating-point format fully in hardware [4, 5], while the earlier released Z9 architecture already supports decimal floating-point operations but implements them mainly in millicode [6]. Nevertheless, the POWER6 decimal floating-point unit is as small as possible and optimized to low cost. Thus, its performance is low. It reuses registers from the binary floating-point unit, and the computing unit mainly consists of a wide decimal adder. Other floating-point operations such as multiplication and division are based on this adder, that is, they are performed sequentially.

Due to the increasing integration density of CMOS devices, Field-programmable Gate Arrays (FPGAs) have recently become attractive for complex computing tasks, rapid prototyping, and testing algorithms. Furthermore, today's FPGA vendors integrate additional dedicated hard-wired logic, such as embedded multipliers, DSP slices, large amount of on-chip RAM, and fast serial transceiver modules. Thus, using FPGA platforms as coprocessors is an interesting alternative to traditional and expensive VLSI designs.

Besides the four basic arithmetic floating-point operations, that is, addition +, subtraction −, multiplication ×, and division /, a fifth arithmetical operation was introduced in the IEEE 754-2008 standard, that is called fused multiply-accumulate (MAC). This operation can assist to improve the accuracy of scalar products. Unfortunately, this approach does not go far enough as consecutively applied

MAC operations, for example, a scalar product, can still lead to totally wrong results because of cancellation. The reason is rounding of intermediate results. For example, the summation of $a_1 = 10^{30}, a_2 = -10^{30}, a_3 = 10$, and $a_4 = -20$, each with 16 digits precision, can lead to four different results, depending on the order of execution

$$\begin{aligned} ((a_1 + a_2) + a_3) + a_4 &\longrightarrow -10, \\ ((a_1 + a_3) + a_2) + a_4 &\longrightarrow -20, \\ ((a_1 + a_4) + a_2) + a_3 &\longrightarrow 10, \\ ((a_1 + a_3) + a_4) + a_2 &\longrightarrow 0. \end{aligned} \tag{1}$$

Scalar products are calculated in many applications, in which cancellation may cause serious problems or numerical overhead slows down algorithms. This includes linear system solving, least squares problems, and eigenvalue problems [7]. In order to overcome these problems, we consider another operation, the so-called accurate scalar product or accurate MAC [8] which is calculated in two steps. First, the products are computed exactly and are added to a long fixed-point register without loss of accuracy. Then, to obtain a floating-point number, the result is rounded only once. This approach guarantees an optimal scalar product with least significant bit accuracy. It can be shown that by providing the accurate scalar product all operations of computer arithmetics can be performed with maximum accuracy, too [9].

Specifications for decimal arithmetic have been added to IEEE 754-2008 mainly for financial applications. Generally, these applications only use a limited range of floating-point numbers such that cancellation errors are not an issue, and an accurate scalar product unit seems to be no gain for decimal arithmetic. Nevertheless, the accurate scalar product unit proposed in this work might be useful because scalar product calculations and accumulations are common operations in financial mathematics, for instance, in portfolio valuation and optimization. Thus, even if cancellation is not an issue, the accurate scalar product unit speeds up these operations because one multiplication and accumulation are computed in the pipeline in one cycle without interlocks, and the high accuracy is gained at no extra cost.

As specified by IEEE 754-2008 [2], the computation of the elementary floating-point operations $+, -, \times$, and / is performed by the computation of the exact (infinitely precise) result followed by a rounding to the destination format. We extend this accuracy requirement to the accurate scalar product operation. Let us denote $R = R(b, p, q_{\min}, q_{\max})$ a floating-point system, where b is the radix, p is the significand's precision, and $q_{\min}$ and $q_{\max}$ are the exponent's range. Moreover, $\mathrm{fl}(x) : \mathbb{R} \rightarrow R$ is a rounding operation that induces floating-point addition $\oplus$ and multiplication $\otimes$ such that

$$a \oplus b := \mathrm{fl}(a + b), \quad a \otimes b := \mathrm{fl}(a \times b), \quad \forall a, b \in R. \tag{2}$$

Then the exact scalar product s can be expressed by

$$\begin{aligned} s &:= \sum_{i=1}^{n} a_i \times b_i = a_1 \times b_1 + \cdots + a_n \times b_n, \\ &\forall a_i, b_i \in R(b, p, q_{\min}, q_{\max}),\ i = 1 \cdots n, \end{aligned} \tag{3a}$$

and the accurate floating-point scalar product $\bar{s}$ by

$$\bar{s} := \mathrm{fl}\left(\sum_{i=1}^{n} a_i \times b_i\right) = \mathrm{fl}(s). \tag{3b}$$

For comparison, the traditional floating-point scalar product $\tilde{s}$ is computed by software, rounding each intermediate result. It can be expressed by

$$\tilde{s} := (a_1 \otimes b_1) \oplus \cdots \oplus (a_n \otimes b_n). \tag{3c}$$

The novelty of the decimal fixed-point multiplier presented here is its parallel and pipelined FPGA nature that is faster than other comparable FPGA implementations and is even time competitive with binary multipliers implemented in FPGAs. The concept of accurate scalar product is not new, but hardware support for binary MAC is seldom and even more rare for the decimal accurate MAC. However, [9] presents a decimal accurate scalar product, but most of the components are serial and have long latencies. Contrary to this, in the new FPGA-based design presented here, we use a fast parallel decimal multiplier and a parallel accurate scalar product unit that can be pipelined to improve latency. This paper summarizes and extends the research published in [10]; in particular, it gives a more detailed introduction and description of the proposed architecture. Furthermore, we improved the speed of the decimal fixed-point multiplier by a factor of two and the accurate scalar product unit by a factor of five, respectively. The outline is given as follows. Section 2 begins with an overview of decimal fixed-point multiplication followed by the description of our proposed parallel decimal multiplier. Section 3 shortly introduces accurate scalar product and presents our proposed architecture. In Section 4, the accurate MAC unit is extended by the concept of working spaces which allow a quasiparallel use of the accurate scalar product unit. Post-place & route results are presented in detail in Section 5, and finally in Section 6, the main contributions of this paper are summarized. Additionally two proofs about complement calculation and simplification of the summation of sign extensions are given in the appendix.

2. Decimal Fixed-Point Multiplier

The Decimal Fixed-Point Multiplier is the basic component of the accurate MAC unit. It computes the product $A \cdot B$ of the unsigned decimal multiplicand A and multiplier B, both natural numbers with the same precision p.

Decimal multiplication is more complex than binary multiplication due to the inefficiency of the digit representation on binary logic. It requires to handle carries

across decimal and binary boundaries and introduces digit correction stages. Furthermore, the number of multiplicand multiples that have to be computed is higher because each digit ranges from 0 to 9. To reduce this complexity, several different approaches have been proposed that are described in the following. All of them have in common that the multiplication is performed in two steps: the generation of partial products and their accumulation. However, they differ in the optimization of these steps.

For the calculation of the partial products, there are two approaches proposed. The first method generates and stores the required multiplicand multiples $\{A\times1,\ldots,A\times9\}$ a priori which are then distributed to the reduction stage through multiplexers controlled by the multipliers digits. Since this approach requires the generation of eight multiples and some of them, for example, $A \times 3$, require a time-consuming carry propagation, Erle et al. [11, 12] proposed a reduced set of multiples $\{A \times 1, A \times 2, A \times 4, A \times 5\}$. All remaining multiplicand multiples can be generated by adding only two from the set. Lang and Nannarelli [13] describe a parallel design that recodes the multiplier's digit set $\{0,\ldots,9\}$ into the digit sets $\{0, 5, 10\}$ and $\{-2,\ldots,+2\}$ exploiting that the multiples $A\times2$ and $A\times5$ can be calculated very fast due to the absence of carry propagation. Vazquez et al. [14, 15] present three different multiplier recoding schemes. The Signed-Digit Radix-10 Recoding transforms the digit set $\{0,\ldots,9\}$ into the signed digit set $\{-5,\ldots,5\}$. The drawback is the need of a carry propagate adder for the calculation of the multiple $A \times 3$. The two others recoding schemes are Signed-Digit Radix-4 Recoding and Signed-Digit Radix-5 Recoding using the transformation sets $\{0, 4, 8\}$, $\{-2,\ldots,+2\}$ and $\{0, 5, 10\}$, $\{-2,\ldots,+2\}$. Both do not need a slow carry propagate adder for partial product generation but require a more complex partial product reduction.

The second method generates the partial products only as needed using digit-by-digit multipliers with overlapped partial products. To reduce the many combinations, in [16] is proposed a digit recoding of both operands from $\{0,\ldots,9\}$ to $\{-5,\ldots,+5\}$. In [17] is described a direct implementation for BCD digit multipliers. It implements a binary digit multiplication followed by a binary product to BCD conversion. Compared to this, in [18] the digit-by-digit multiplier is implemented by means of the FPGA's memory; however, no digit recoding is applied.

The accumulation of the partial products consists of two stages: the fast reduction (addition) to a two-operand and a final carry propagate addition. Similar to binary multiplication, the accumulation of the partial products can be performed sequentially, in parallel, or by a semiparallel approach. A sequential multiplier iteratively adds up each partial product to an accumulated sum. In [19], the accumulation is performed sequentially by decimal (3 : 2) carry save adders and a final carry propagate adder which leads to a short critical path delay and low area usage but longer latency. It performs a multiplication in $p+4$ cycles. In parallel multipliers, the area consumption is much higher, but the latency can be reduced and the architecture can be pipelined to achieve a higher throughput. In [13], a fully parallel multiplier with digit recoding (see above) is presented. The accumulation is performed by a tree of carry save adders and a final carry propagate adder. Vazquez et al. [14] present a new family of parallel decimal multipliers. The carry-save addition in the reduction stage uses new redundant decimal BCD-4221 and BCD-5211 digit encodings. In [20] is introduced a new method of partial product generation and together with the reduction scheme of [13] and the carry propagate addition method of [14] this design is believed to be the fastest design in literature but sacrifices area for high speed. Despite the partial product reduction scheme presented in [20] is the fastest for ASIC designs, the reduction scheme presented in [14] is more appropriate to FPGA designs. The reason is that [20] is based on BCD full adders which introduce a delay of two lookup tables per reduction stage, whereas the reduction scheme presented in [14] can be implemented with a delay of only one lookup table per reduction stage.

Contrary to the several works on implementations in ASICs, decimal multipliers are not often implemented in FPGAs. These few are [10, 18, 21]. The method in [21] exploits the FPGA's internal binary adders and uses decimal to binary conversion and vice versa. This approach is only feasible for small multipliers. The decimal multiplication in [18] is sequentially and is based on digit-by-digit multipliers that are implemented by memory (BRAM or distributed RAM). It also describes a combinational multiplier design which is only applicable for small precisions p. In a recent work [10], we proposed a fully combinational decimal fixed-point multiplier optimized for Xilinx Virtex-II Proarchitectures [22]. It is based on fast partial product generation and a combinational fast carry save adder tree. It can be pipelined to achieve a high throughput which is a crucial feature for the usage in an accurate scalar product unit. In this work, we adapted the design for Xilinx Virtex-5 devices [23], and in doing so we could double speed and throughput.

2.1. Proposed Parallel Decimal Multiplier. The proposed Decimal Fixed-Point Multiplier computes the product $A \cdot B$ of the unsigned decimal multiplicand A and multiplier B. 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, which is based on [15]. It is optimized for use on Xilinx Virtex-5 FPGAs. A decimal natural number Z is called BCD-$\beta_3\beta_2\beta_1\beta_0$ coded when Z can be expressed by

$$Z = \sum_{m=0}^{p-1} Z_m \cdot 10^m,$$
$$Z_m = \sum_{n=0}^{3} Z_{mn} \cdot \beta_n, \quad Z_{mn} \in \{0, 1\}. \tag{3}$$

Time-critical components are BCD-8421 carry propagate adders (CPAs) that are used in partial product generation to calculate the multiplicand's triple fold $A \times 3$ and for final addition. The adders are proposed in [24] and are designed and placed on slice level, considering a minimum carry chain length and least possible propagation delays. Figure 1 shows

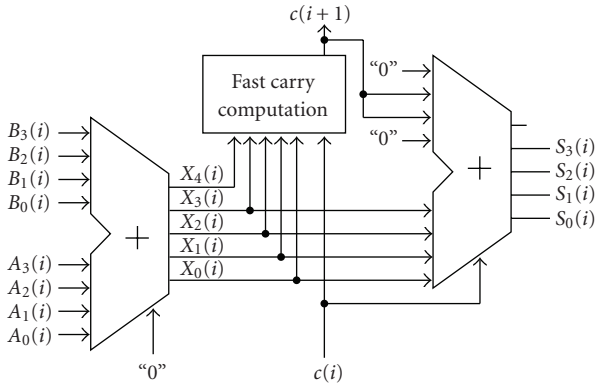


Figure 1: BCD-8421 full adder.

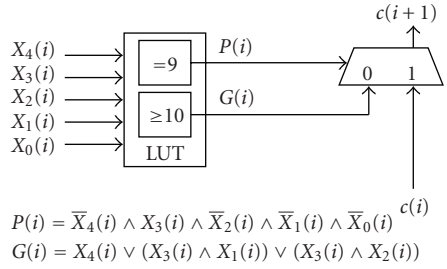


Figure 2: Fast carry computation.

an elementary BCD-8421 full adder. It consists of an adding and a correction stage using two binary 4-bit adder and a fast carry computation unit that is depicted in Figure 2. It exploits the FPGA's internal fast carry chains to minimize latency. The fast carry computation unit implements two functions on the intermediate result of the first stage, *propagate* $P(i) = P(y(i))$ and *generate* $G(i) = G(y(i))$ with $y(i) := 16x_4(i) + 8x_3(i) + 4x_2(i) + 2x_1(i) + x_0(i)$,

$$P(i) = P\big(y(i)\big) = \begin{cases} 1 & \text{if } y(i) = 9, \\ 0 & \text{otherwise,} \end{cases}$$
$$G(i) = G\big(y(i)\big) = \begin{cases} 1 & \text{if } y(i) \geq 10, \\ 0 & \text{otherwise,} \end{cases} \tag{4}$$

and the carry signal $c(i+1)$ yields to

$$c(i+1) = \begin{cases} c(i) & \text{if } P(i) = 1, \\ G_i & \text{otherwise.} \end{cases} \tag{5}$$

Altogether the adder consumes 9 lookup tables (LUTs) per digit. In particular, the fast carry-bypass logic (carry computation unit) spans only over one LUT.

Generally, the fixed-point multiplier consists of six functional blocks as depicted in Figure 3. 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 carry propagate adder (CPA). The CSAT is based on (3 : 2) CSA blocks for BCD-4221 format.

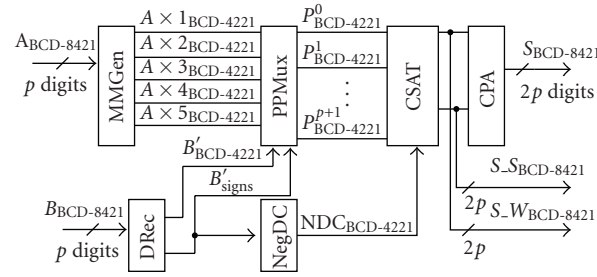


Figure 3: Parallel fixed-point multiplier.

The partial products are 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 $\{-5, \ldots, 5\}$ [15], and a simple method to handle negative partial sums (10's complement), only five multiples ($A \times 1$, $A \times 2$, $A \times 3$, $A \times 4$, $A \times 5$) have to be generated by the multiplicand multiples generator (MMGen) a priori. It can be easily proven that the 10's complement can be calculated by inverting each bit of all digits and adding one (see the appendix) . The functionality of the negative digits correction (NegDC) block is explained in the following.

The MMGen is similar to the generator of multiplicand multiples for SD radix10 encoding in [15], but the decimal quaternary tree is replaced by the BCD-8421 CPA. It exploits the correlation between shift operation and constant value multiplication. For example, a BCD-5421 coded decimal number left shifted by one bit is equivalent to a multiplication by 2, and the result is being BCD-8421 coded. Similarly, a BCD-5211 coded number left shifted by one results in a multiplication by two with a BCD-4221 coded result. And finally, a BCD-8421 coded decimal number shifted by three results in a multiplication by 5, and the result is of type BCD-5421.

$$\begin{aligned} (X)_{\text{BCD}-5421} \ll 1 &\equiv (X \cdot 2)_{\text{BCD}-8421}, \\ (X)_{\text{BCD}-5211} \ll 1 &\equiv (X \cdot 2)_{\text{BCD}-4221}, \\ (X)_{\text{BCD}-8421} \ll 3 &\equiv (X \cdot 5)_{\text{BCD}-5421}. \end{aligned} \tag{6}$$

A recoding operation 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, with exception of $A \times 3$, all multiples can be easily generated by simple shift operations and digit recodings. For the $A \times 3$ multiple, an additional CPA is inevitable which unfortunately limits the maximum working frequency and thus emphasizes the need of pipelining. Alternatively, the multiples could be composed of two operands and be added in the following CSAT, as proposed in [12]. This would speedup the MMGen but would also double the inputs to the CSA and increase significantly its complexity and resource consumption. Figure 4 depicts the functionality of the MMGen. It is similar to the generator of multiplicand multiples presented in [14], but we replaced the decimal quaternary tree by our BCD-8421 adder.

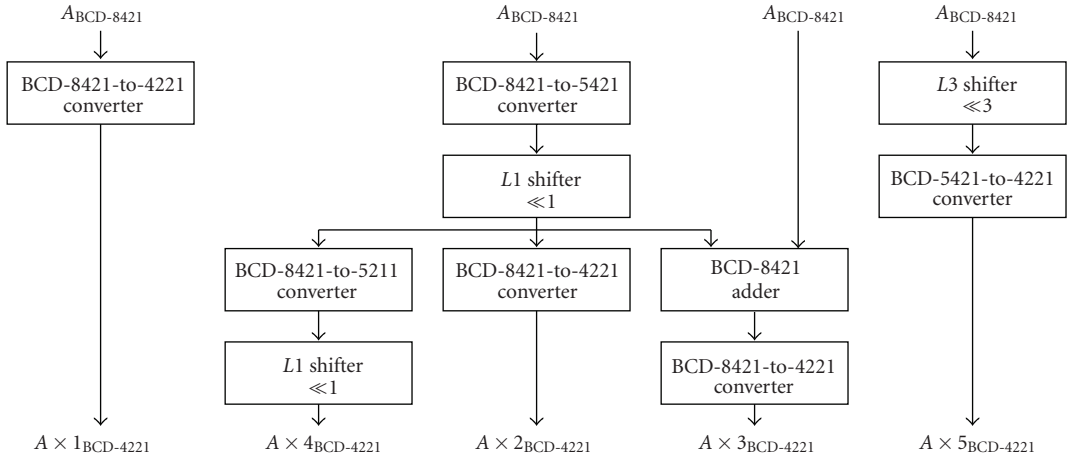


Figure 4: Multiplicand multiples generator.

The decimal recoding unit (DRec) depicted in Figure 3 reduces the number of multiplicand multiples that have to be computed by the MMGen, as proposed by Vazquez et al. [15]. In the first step, it transforms each multiplier's digit B_k from the digit set $\{0,\ldots,9\}$ into the signed digit set $\{-5,\ldots,4\}$ and an output carry bit c_k which coincides with the sign signal sign_k

$$\left(B'_k, c_k\right) = \begin{cases} (B_k, 0) & \text{for } B_k \in 0,\ldots,4, \\ (B_k - 10, 1) & \text{for } B_k \in 5,\ldots,9. \end{cases} \tag{7}$$

In the second step, the carry signal from the previous digit is added to the intermediate result

$$B''_k = B'_k + c_{k-1}. \tag{8}$$

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 necessary padded with 9. 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 (see the appendix) . For example,

$$\begin{array}{ll} & 999999990000 \\ + & 999900000000 \\ + & 990000000000 \\ \hline = & x989899990000. \end{array} \tag{9}$$

Moreover, as shown in Figure 5 the position of the nines and eights can be calculated very fast by means of the FPGA's fast carry chain considering the following equations:

$$\text{NDC}(i) = \begin{cases} 9 & \text{for } c(i) = 0 \text{ and } \text{sign}\, B(i) = 1, \\ 8 & \text{for } c(i) = 1 \text{ and } \text{sign}\, B(i) = 1, \\ 0 & \text{else,} \end{cases} \tag{10}$$

$$c(i+1) = \begin{cases} 1 & \text{for } \text{sign}\, B(i) = 1, \\ c(i) & \text{else.} \end{cases}$$

The reduction of the partial products is based on BCD-4221 $(3:2)$ CSAs [15] that reduce three BCD-4221 digits to a sum and a carry digit, both of BCD-4221 coding scheme. In a first version, CSA1, the carry save adder is implemented as proposed by Vazquez et al. [15]. It consists of a 4-bit binary $(3:2)$ CSA and a BCD-4221 to BCD-5211 digit recoder. By means of an implicit shift operation of the BCD-5211 coded carry digit, we obtain a multiplication by two. The block diagram of CSA1 is shown in Figure 6. It consumes overall six LUTs per digit. The drawback of this architecture is that the computation of the sum digit s_i has a latency of one LUT, whereas the computation of the carry digit w_i has a latency of two LUTs. To reduce the computation latency of w_i, we propose a new type of carry save adder, CSA2. It consists of a 2-bit binary $(3:2)$ CSA and a carry digit computation unit. The block diagram of CSA2 is shown in Figure 7. The 2-bit binary $(3:2)$ CSA sums up the two least significant bits of the three input digits and generates the sum digit. The carry digit is computed from the remaining six most significant bits of the three input digits which requires four $(6:2)$ LUTs. The CSA2 method also consumes six LUT per digit but has a lower latency than CSA1.

The $(n:2)$ CSA tree is composed of parallel and consecutively wired $(3:2)$ CSAs. It reduces n decimal words to two BCD-4221 coded decimal words. 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 CSAT is organized in stages, each reduces p_i

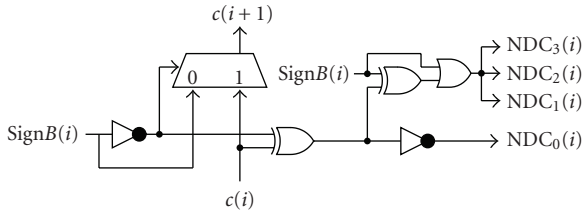


Figure 5: NegDC unit.

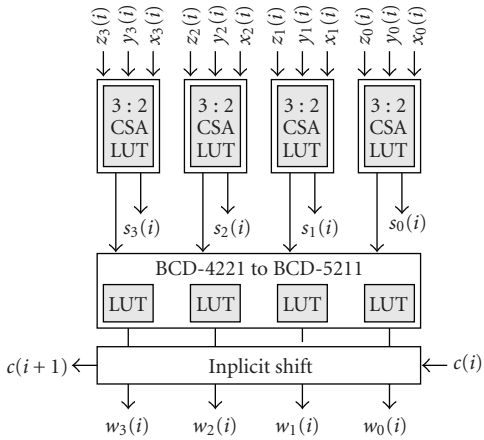


Figure 6: (3 : 2) CSA type1 implementation.

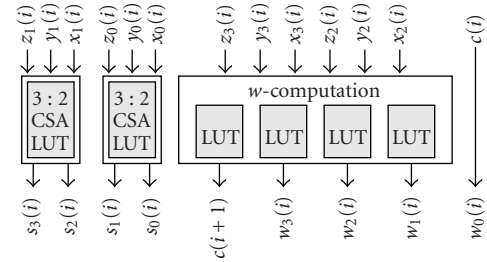


Figure 7: (3 : 2) CSA type2 implementation.

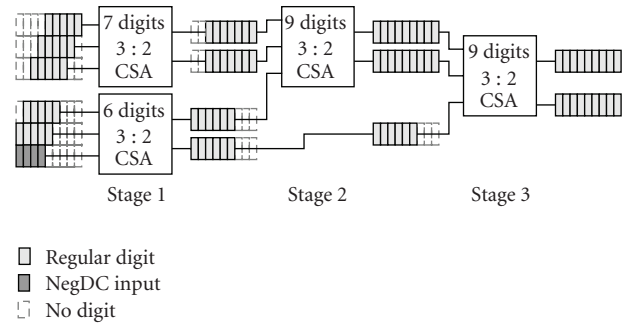


Figure 8: Example: CSA Tree for 6 input words.

words to $p_{i+1} = p_i \cdot 2/3$ words. As in general the ranges of the input words differ, the word length increases with each stage as depicted exemplary in Figure 8.

The redundant carry-save format of the CSAT can be further reduced by a carry propagate adder of length $2p$ to obtain a unique result. However, this CPA can be omitted because the accurate scalar product unit processes on the carry-save format directly.

The maximum frequency of the fixed-point multiplier is limited on the one hand by time-critical components like the CPA and on the other hand by the FPGA's routing overhead. While the maximum propagation delay of the time-critical components can be determined in advance, the routing delay depends highly on the overall project's size. Hence, several pipeline registers 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 registers for the BCD-4221 carry-save format output and 13 stages for the final BCD-8421 carry-propagation format output. It should be noted that the last CSA stage can be combined with the final BCD-8421 converter, as it is proposed by [15]. However, since the following accurate scalar product unit accumulates redundant BCD-4221 numbers, this improvement could not be applied.

3. Accurate Scalar Product

The accurate scalar product is important for applications in which cancellation may cause problems or numerical overhead slows down algorithms. It is calculated in two steps. First, the products are computed exactly and are summed up to a long fixed-point register without loss of accuracy. Then the result is rounded only once to obtain a floating-point number. Hardware support for the accurate binary scalar product is rare; the accurate decimal scalar product is even less supported by hardware. In [25] is presented a coprocessor with an accurate binary scalar product using the concept of the long accumulator. Reference [9] presents a decimal floating-point arithmetic with hardware as well as software support. It implements the concept of accurate scalar product, but due to the given hardware restrictions most of the components are serial and have long latencies. Contrary to this, in the new FPGA-based design presented here, we use a fast parallel multiplier and parallel shift registers. We accelerate the scalar product's accumulation by use of carry save adders and get rid of overflow and carry signals by the concept of carry caches. Our design is pipelined and requires generally five cycles to multiply and accumulate with an operating frequency of more than 100 MHz.

3.1. Proposed Accurate Scalar Product Unit. The fundamental concept of the long accumulator (LA) is to provide a fixed-point register that covers the entire floating-point range of products, as well as adder, that accumulates these products without loss of accuracy, see Figure 9. When computing the scalar product (3a) n, individual results coming from the decimal fixed-point multiplier are shifted and added

to a section of the LA. The respective section depends on the operands' exponents and is calculated by the address generator (AGen). In order to avoid time-consuming carry propagation, the central adder (CAdd) is implemented as carry-save adders which implies a doubling of the LA's memory to store both operands. Contrary to [9], positive as well as negative operands are accumulated in the same LA by using 10's complement data format. To prevent time-consuming ripple-carry propagations due to sign swapping and overflow, we use a so-called carry cache (CC) that buffers any overflow signals. Contrary to a previously published paper [10], in this work, we have simplified the carry handling by removing the principle of fast carry resolution in case of a carry cache overflow. Instead, we have increased the block size of the long accumulator for carry cache (LACC) to 16 digits, assuming that the CC will never overflow. Actually, in the worst case scenario, it would take the CC over three years to overflow at a reasonable working frequency of 100 MHz. Applying this simplification, we could increase the operating frequency significantly. Before the final accurate scalar product can be output and stored on a temporary result stack (ResSt), the two carry-save operands of the long accumulator for operands (LAOPs) and the entries of the LACC must be summed up and reduced by a final carry propagate adder (FCPA). Therefore, the entire long accumulator would have to be traversed which is a highly inefficient step, since due to locality most applications normally use a minor percentage of the LA and the remaining entries equal zero. To solve this problem, we introduced a so-called touched blocks register (TBR). During MAC operation, the TBR marks the corresponding blocks of the LA as *touched*, which means they are most likely unequal to zero. During final result calculation, only these blocks, that have previously been marked as touched, are actually addressed and read out.

The required length l in digits of the long accumulator can be calculated from the significand's length p and the minimum exponent's value $q_{\min}$ and maximum exponent's value $q_{\max}$, respectively. In order to consider possible overflows, k more guarding digits are provided on the left. For our design, the number of guarding digits $k = 18$ is chosen. Considering a maximum working frequency of 100 MHz, it would take the LA over 300 years to overflow. Hence, 18 guarding digits are a reasonable choice. Since a multiplication doubles the significand's length and the exponent's range, the LA must hold a total number of digits as follows:

$$l = k + 2 \cdot (q_{\max} - q_{\min}) + 2 \cdot p. \tag{11}$$

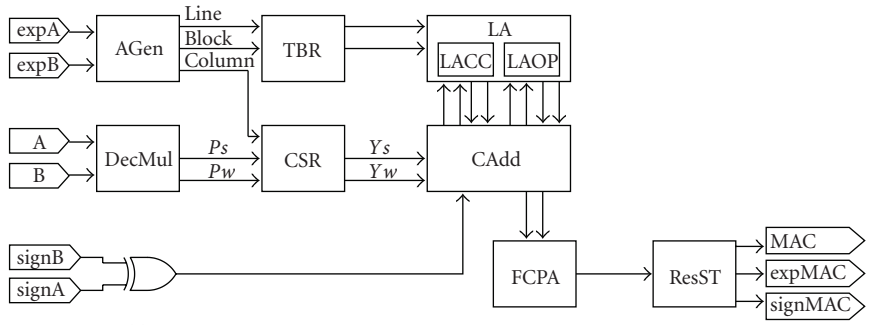
We implemented the MAC unit for IEEE 754-2008 *decimal64* interchange format with $p = 16$ digits precision. With $k = 18$, $q_{\max} = 369$, and $q_{\min} = -398$, the accumulator length results in $l = 1584$. The interchange format *decimal32* with 7 digits precision is downward compatible and thus can be applied to the decimal MAC unit, too.

The LA is implemented by use of local Block SelectRAM (BRAM). It is organized in $\lceil l/p \rceil$ segments, each covers $p = 16$ digits. Since the shifted multiplier's result always fits into $3p$ digits, three arbitrary consecutive segments can be addressed, yielding a word of $3p$ digits. Therefore, the LA is organized in three blocks with $l/(3 \cdot 16) = 33$ lines. It provides memory for both the long accumulator for operands (LAOP) as well as for the long accumulator for carry cache (LACC). To each LAOP block, an LACC block is assigned that handles any overflow signals during accumulation. This prevents pipeline interrupts and allows the storage of negative numbers in 10's complement data format. One LA line comprises of LACC and LAOP each with three blocks composed of 16 digits with 4 bits. As the central adder is of $(4:2)$ carry-save type with length $3p$, two carry-save operands and two carry cache entries must be stored separately. The advantage of this approach is its high speed because of the absence of a ripple carry signal. The drawback is twice as much memory consumption. Since BRAM is a dual-ported memory, the two carry-save operands can be accessed simultaneously through different ports. Thus, $n = ((4 \cdot 16 \cdot 3) \cdot 2) \cdot 2 = 768$ bits must be addressed in parallel which requires 12 parallel dual-ported BRAMs with 32 bit data ports. Each BRAM has a memory depth of 1024, but both operands only need a depth of $2 \cdot (l/(3 \cdot p)) = 66$. The remaining memory can be used for the implementation of the so-called working spaces (WSs) which are introduced below. The LA runs at double data rate, because within one cycle the operands and carry cache entries from the LA have to be fetched and added to the multiplier's output and then in the same cycle the result has to be written back to the LA.

When a block address is not a multiple of three, then the operand spans over two memory lines, that is, the least significant digits (LSDs) are not located in the first block but in the second or third. The block alignment is performed by the shift register which is therefore implemented as a cyclic shift register, see Figure 10. Alternatively, the block alignment could have been implemented between LA and CAdd, but this approach would have increased the longest path and would have reduced the overall operating frequency.

The drawback of the memory organization in lines comprising three segments is a complicated address generation, that is, the need of a division by three. An alternative solution with four blocks per line leads to an easier address calculation but also requires larger multiplexers for operand shift operations. Fortunately, the complicated division by three can be accomplished by applying an embedded binary multiplier, as described in the following.

The address generator (AGen) shown in Figure 9 transforms the input exponents into three addresses (column, block, and line address) to access the LA and to control the shift register. The line and block addresses define a segment $s = \text{line} \cdot 3 + \text{block}$, and the column address locates the position inside this segment. Thus, each digit in the LA can be characterized by its exponent E that relates to the three addresses as follows: $E = \text{line} \cdot 48 + \text{block} \cdot 16 + \text{column}$. The central adder can only sum up block-aligned operands. For that reason, the multiplier's result has to be shifted cyclically. The shift left amount (SLA) arises from the column and



AGen: Address generator
TBR: Touched blocks register
DecMul: Decimal multiplier
CSR: Cyclic shift register
LA: Long accumulator
LACC: LA for carry cache
LAO: LA for operands
CAdd: Central adder
FCPA: Final carry propagation adder
ResSt: Reset stack

Figure 9: Accurate scalar product unit.

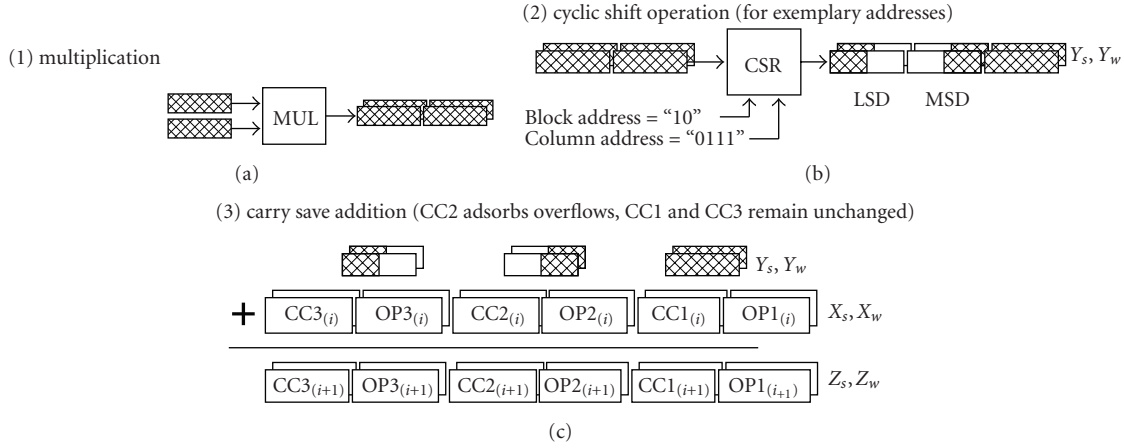


Figure 10: Block alignment of LA and CSR.

block addresses, whereas the block and line addresses are used to address the LA,

$$\text{SLA} = \text{block\_addr} \cdot 16 + \text{column\_addr}. \tag{12}$$

Unfortunately, the memory partitioning applies a division by $3 \cdot 16 = 48$ to determine the line address. That division is accomplished by inverse multiplication considering the maximum digit's exponent of $E_{\max} = 1550$. This approach requires besides logical, shift, and add operations one additional binary fixed-value multiplication which can be performed by the dedicated multiplier of the FPGA's DSP48E slices, see Algorithm 1 .

Once the result has been computed from the decimal multiplier it enters the shift register before it is accumulated by the central adder and is stored in the LA, as already described above. The shift register extends the decimal multiplier's outputs from $2p$ to $3p$ length and shifts the operands according to the column address. Because the decimal multiplier internally uses digit recoding combined with 10's complement representation, there might arise a carry signal (whenever at least one multiplier's digit is greater than or equal to five) which is discarded by the subsequent CPA but is still present as a *hidden carry* in the output of carry-save format. In such cases, the most significant digits

$E = \exp(\text{A}) + \exp(\text{B})$
$\text{line} = \lfloor E/(3*16) \rfloor = (E*1366) \gg_2 16$
$\text{res} = E - \text{line}*48 = E - (\text{line} \ll_2 5 + \text{line} \ll_2 4)$
$\text{block} = \text{res} \gg_2 4$
$\text{column} = \text{res} \,\&\, 0\text{b}1111$

Algorithm 1: Address generation.

Table 1: Shift register post-place & poute results.

	T [ns]	freq_{max} [MHz]	# LUTs
mul-based, 48 bit	5.072	197	193
mul-based, 2×48 digits	5.177	193	1201
mux-based, 48 bit	2.565	389	960
mux-based, 2×48 digits	2.993	334	7512

(MSDs) of the extended $3p$ word must be padded with 9's, and the overflow has to be cleared by a subtraction of 1 in the carry cache adder, see Algorithm 2 . However, the main challenge is the vast shift depth up to 47 digits along with a large number of operands to be shifted, that is two operands each with four bits per digit. These are $2 \cdot 4 = 8$ 48-bit cyclic shift register. Since serial shift register with low resource consumption cannot be pipelined, only parallel solutions are applicable. Two solutions for parallel cyclic shift registers are analyzed, the first one is a shift register using multiplexers and the second one applies the hard-wired multiplier of the DSP48E slices. The latter one is possible because an Lk-shift operation complies with a multiplication by 2^k. Virtex-5 devices support the design of large multiplexers by using the dedicated F7AMUX, F7BMUX, and F8MUX multiplexers. Hence, four LUTs can be combined into a $16:1$ multiplexer.

A 48-bit shift register can be implemented by three 16-bit shift register stages wired consecutively. These shift registers are composed of 16-bit multiplexers or multipliers. Each stage can be pipelined to obtain a low latency as shown in Figure 11. Table 1 summarizes the maximum delay and the number of LUTs used for both cyclic shift register solutions. The multiplexer-based solution is faster but requires much more LUTs, up to 6.25 times more. Since the longest path in the accurate scalar product unit is bounded by the central adder (approximately 10 ns), the multiplier-based cyclic shift register is preferred because of its far less resource usage.

The central adder is a $(4:2)$ CSA to keep latency low. The four inputs are two cyclically shifted words from the decimal multiplier and two operands from the long accumulator. The central adder is composed of two sequentially arranged $(3:2)$ CSA stages. Furthermore, negative numbers are applied by their 10's complement that requires an additional correction of +1. Since the multiplier's output is of redundant carry-save type, two correction factors of +1 are needed. For this purpose, the carry inputs of the $(3:2)$ CSA stages are used. Each CSA stage also produces a carry signal that has to be absorbed by the Carry Cache described below. One $(3:2)$ CSA stage comprises of three 16-digit $(3:2)$ CSAs that are interconnected depending on the block address, see Figure 12.

To handle overflow during accumulation without interfering the pipeline and to allow the storage of negative numbers in 10's complement format without carry propagation, we introduced the CC. It temporarily adds and stores carry and sign signals. The CC uses the carry-save format, too. To each LAs operand block is assigned a CC block which consists of 16 digits and adsorbs the two carry signals of the LA (cout1, cout2) and the two negative sign signals due to 10s complement (sign). Because of its size, the CC blocks are not supposed to overflow. Finally, the CCAs neutralize the hidden carry signal, too, that is weighted negative in case of positive numbers but positive in case of negative numbers. Summarizing all factors yields to the pseudocode depicted in Algorithm 2.

The final result is computed by successively reading out the LA, starting with the least significant digit (LSD) and reducing the CC's entries as well as the LA's operands by means of the CAdd. Finally, this redundant result is summed up by the final carry propagate adder and stored on the result stack (ResSt). Hence, the FCPA produces a series of positive and negative floating-point numbers with the precision of $3 \cdot 16 = 48$ digits and ascending block aligned exponents. The carry out signal of the FCPA is fed back to the FCPA's carry input. The ResSt is composed of a dual ported memory. On the one port, the result of the FCPA is written into the memory, whereby zero entries are omitted. On the other port, the result is accessible for external components with either greatest or smallest number first, depending on requirements of the further data processing. For example, when a final rounding is required to fit the result into IEEE 754-2008 data format, then it is advantageous to read out the greatest number first.

As application is usually subject to locality only a small percentage of the LA is filled with nonzero entries. Thus, it would be very inefficient to traverse the complete LA during final readout. Due to performance issues, we introduced the so-called touched blocks register (TBR). Each time the MAC unit accesses a block in the LA, an according flag in the TBR is set to indicate highly probable nonzero data. Only these previously *touched* blocks in the LA are regarded to compute the final result. In order to reduce the complexity for final result computation, four consecutive blocks are marked as *touched* instead of three as might be expected. This method simplifies the final result computation because possible overflows are already considered and no further exceptions must be regarded.

The parallel fixed-point multiplier as well as the accurate scalar product unit are designed to support pipelining. As already described, the fixed-point multiplier with redundant carry-save output has 11 configurable pipeline registers that can be switched on and off by VHDL generic switches. The accurate scalar product unit further adds three stages for the cyclic shift register and also three stages for the final carry propagate adder. Especially the latter ones are important to reduce the longest asynchronous path and to achieve high operating frequencies.

4. Working Spaces

The introduction of so-called working spaces (WS) allows the quasiparallel use of the MAC unit, that is, there can be several users concurrently accessing the MAC unit without interfering each other. The users can be different processors or different processes on one processor. There can even be a single process that handles more than one accurate scalar product unit, for example, to compute complex scalar products, interval scalar products, and so forth. Working

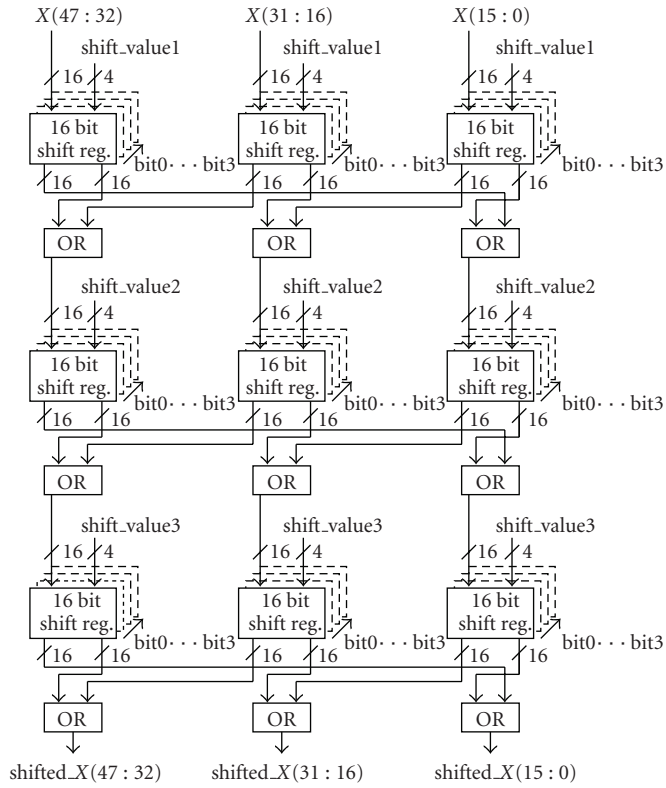


Figure 11: 48-digit cyclic shift register.

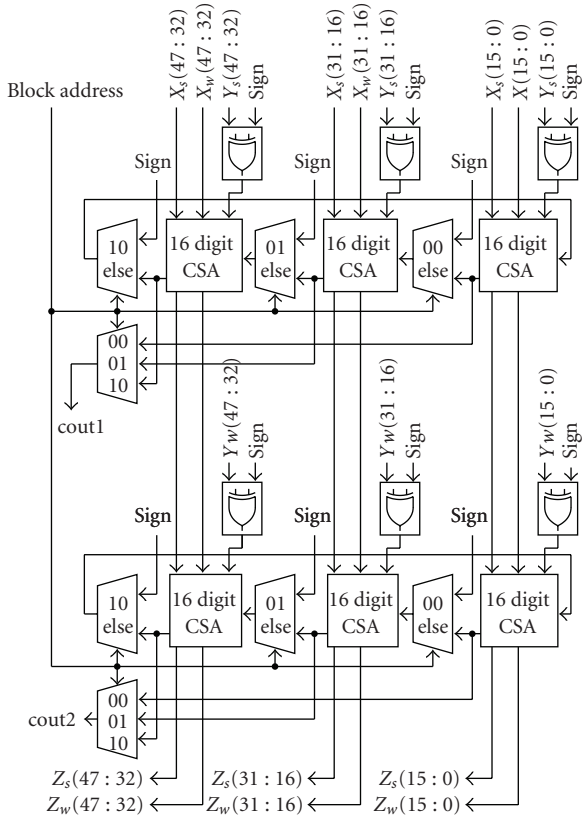


Figure 12: Central (4 : 2) CSA for operands.

```
CC[line + block] += cout1 + cout2
                    −2 · sign
                    −hidden_carry · (−1)^sign
```

Algorithm 2: Pseudocode for CCA calculation.

spaces are realized by duplications of all memory elements together with some additional multiplexers. These are the long accumulator with operand storage and carry cache, the touched blocks register, and the reset stack. The assignment and access to the working spaces has to be managed by a central control unit, for example, an operating system. The number of working spaces can be set by VHDL generics, too. Actually, it is only limited by available resources.

5. Synthesis Results

All circuits are modeled using VHDL. For synthesis and implementation Xilinx ISE 10.1 [26] has been used. The fixed-point multiplier and the accurate scalar product unit have been implemented for Xilinx Virtex-5 speed grade -2 devices. Firstly, only the fixed-point multiplier with unique carry propagate output has been implemented for several pipeline configurations, see Table 2 and Figure 13.

The results show that the minimum overall latency of about 18 ns can be achieved with no pipeline registers, and

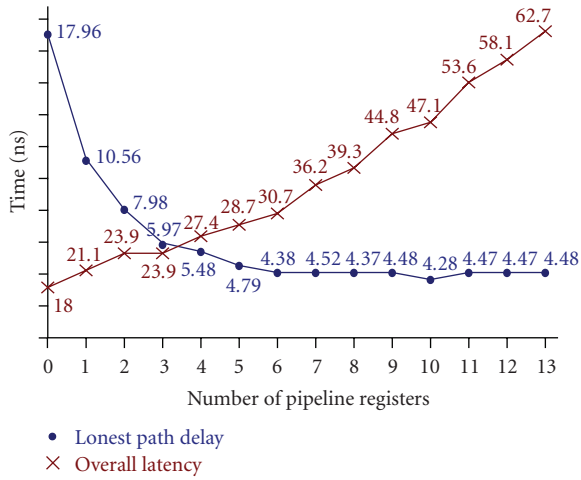


Figure 13: Latency and delay (Post-Place & Route Results) for decimal multiplier, $p = 16$ digits,

Table 2: Post-place & route results for decimal fixed-point multiplier with CPA output.

No. of pipeline stages	longest path delay [ns]	max. freq. [MHz]	No. of LUT [ns]	latency
0	17.96	57	6557	17.96
1	10.56	95	5916	21.12
2	7.98	125	5385	23.94
3	5.97	167	6074	23.88
4	5.48	183	6237	27.40
5	4.79	209	6402	28.74
6	4.38	229	5576	30.66
7	4.52	221	5574	36.16
8	4.37	229	5576	39.33
9	4.48	224	5610	44.80
10	4.28	234	6139	47.08
11	4.47	224	6283	53.64
12	4.47	224	6318	58.11
13	4.48	224	6520	62.72

the best operating frequency of 234 MHz can be obtained with 10 pipeline registers. However, using 6 or more pipeline registers does not reduce the longest path delay significantly and increases the overall latency instead. The LUT usage varies only slightly for different pipeline configurations. In [18], combinational and sequential memory-based digit-by-digit multipliers are analyzed for Xilinx Virtex-4 platforms. A combinational 16×16 multiplier uses 22,033 LUTs and has an overall latency of 26.9 ns. A sequential 16×16 multiplier uses 1,054 LUTs, 8 BRAMs, and has an overall latency of 110.5 ns. A fair speed comparison with the design proposed in this work is difficult because of different FPGA devices. Nevertheless, the unpipelined design proposed in this work is 50% faster than the combinational multiplier proposed in [18]. The sequential multiplier uses rather few

Table 3: Comparison of decimal fixed-point and binary multiplier results.

Multiplier	$T_{latency}$ [ns]	No. of LUTs	No. of DSP48E
Decimal multiplier 0 pipeline registers	17.964	6557	0
CoreGen LUT-based 0 pipeline registers	11.965	2945	0
CoreGen DSP48E-based 0 pipeline registers	25.712	0	10
Decimal multiplier 6 pipeline registers	4.377	5576	0
CoreGen LUT-based 6 pipeline registers	3.475	2982	0
CoreGen DSP48E-based 6 pipeline registers	4.958	123	10

Table 4: Post-place & route results.

	T[ns]	No. of LUTs	No. of RAMB36	No. of DSP48E
MAC[1], 1–8 WS	9.964	10,032	18	73
MAC[1], 8–16 WS	9.971	11,891	36	73
MAC[1], 16–24 WS	9.976	13,302	54	73

[1] Decimal fixed-point multipliers in accurate MAC use two pipeline registers.

LUTs. But contrary to the combinational multiplier, it has a poor latency and cannot be pipelined. Thus, only the combinational multiplier might be suitable for an accurate scalar product unit. However, it uses a considerable amount of LUTs more than the multiplier proposed in this work.

To compare our design with multiplier designs implemented for the same FPGA chip, we have analyzed a binary 53×53 multiplier on a Virtex-5 provided by the Xilinx Core Generator, see Table 3. Our architecture is faster than the DSP48E-based binary multiplier. On the other hand, our fixed-point multiplier consumes approximately twice as much LUTs as the binary LUT-based multiplier and is slower, but it has to be considered that decimal multiplication is much more complex than binary multiplication.

The accurate MAC unit has been implemented with two pipeline registers for the decimal fixed-point multiplier. Together with the three pipeline registers of the cyclic shift register, this amounts to a 5-cycle latency to calculate and store the product of two operands on the long accumulator. The accurate MAC unit can be clocked with up to 100 MHz. Compared to a previously published paper [10], this is an improvement by a factor of five. In comparison, a software implementation of a single 16 digits floating-point multiplication without any long accumulator on a high-performance processor already uses 233 cycles, on lower performance architectures even more [27].

The resource consumption of the accurate MAC unit depends on the number of implemented working spaces. Table 4 summarizes the resource consumption for different configurations.

6. Conclusion

In this paper, we presented a decimal fixed-point multiplier that maps onto FPGA architectures and can help to implement a fully IEEE 754-2008 compliant coprocessor. We analyzed the performance with respect to the number of pipeline registers. Moreover, we integrated the decimal multiplier into an MAC unit which can compute scalar products without loss of accuracy and thus can prevent numerical cancellation. Using the MAC unit on multitasking machines is supported by the concept of working spaces. Compared to a previously published paper [10], we ported our former architecture that was designed to map on (4 : 1) LUT-based Xilinx Virtex-II Pro devices to up to date (6 : 2) LUT-based Xilinx Virtex-5 devices. Furthermore, we improved the algorithm of the accurate scalar product unit. For the fixed-point multiplier, we could achieve a speedup of two, and for the entire accurate scalar product unit we could even achieve a speedup of five. Even though the migration from Virtex-II to Virtex-5 devices has improved the speed of the accurate scalar product unit, the greater part of the speedup is attributable to the improved algorithm.

Appendix

Proofs

Theorem 1 (B's complement). *Let $B \in \mathbb{N}$ denote a radix, p the precision, $X = \sum_{i=0}^{p-1} x_i \cdot B^i$ a positive integer with digits $x_i = \sum_{k=0}^{n-1} x_{ik} \cdot r_k$, $x_{ik} \in \{0,1\}$, and $r_k \in \mathbb{Z}$.*

If $\sum_{k=0}^{n-1} r_k = B - 1$, then the B's complement of X can be easily computed by

$$-X = -B^p + 1 + \sum_{i=0}^{p-1} \overline{x_i} \cdot B^i, \quad \text{with } \overline{x_i} = \sum_{k=0}^{n-1} \overline{x_{ik}} \cdot r_k, \quad \overline{x_{ik}} = \begin{cases} 0 & \text{when } x_{ik} = 1, \\ 1 & \text{when } x_{ik} = 0. \end{cases} \tag{A.1}$$

Proof. Firstly, we prove the B's complement for one digit

$$\begin{aligned} x_i &= \sum_{k=0}^{n-1} x_{ik} \cdot r_k \in \{0, 1, \ldots, B-1\}, \\ &\Longrightarrow \overline{x_i} = \sum_{k=0}^{n-1} \overline{x_{ik}} \cdot r_k = \sum_{k=0}^{n-1} -(x_{ik} - 1) \cdot r_k, \\ &= \sum_{k=0}^{n-1} r_k - \sum_{k=0}^{n-1} x_{ik} \cdot r_k = B - 1 - x_i \\ &\Longrightarrow -x_i = -B + \overline{x_i} + 1. \end{aligned} \tag{A.2}$$

Then, we calculate the complement $-X$ of a number X

$$\begin{aligned} -X &= \sum_{i=0}^{p-1} (-x_i) \cdot B^i \\ &= \sum_{i=0}^{p-1} (-B + \overline{x_i} + 1) \cdot B^i \\ &= \sum_{i=0}^{p-1} \left(-B^{i+1} + \overline{x_i} \cdot B^i + B^i\right) \\ &= -B^p + 1 + \sum_{i=0}^{p-1} \overline{x_i} \cdot B^i. \end{aligned} \tag{A.3}$$

□

Theorem 2 (Sum of leading nines). *The sum of words composed of leading 9's and following 0's*

$$s = \sum_{i=1}^{p} X_i \quad \text{with } X_i = \sigma_i \cdot \sum_{k=i}^{p} 9 \cdot 10^k, \quad \sigma_i \in \{0,1\} \tag{A.4}$$

is a decimal word for which the $p+1$ least significant digits consist of zero or more leading 9's followed only by 0's, 8's, and 9's, that is,

$$s = c_p \cdot 10^{p+1} + \sum_{i=j}^{p} 9 \cdot 10^i + \sum_{i=1}^{j-1} y_i \cdot 10^i, \tag{A.5}$$

with $p \geq j \geq 1$, $y_i \in \{0, 8, 9\}$ and $c_p \in \mathbb{N}$.

Proof. We prove the assumption by complete induction.

(1) If all $\sigma_i = 0$, for all $i = 1, \ldots, p$, then the assumption is true because $s = 0$.

(2) If $\sigma_k = 1 \wedge \sigma_i = 0, i \neq k$, then the assumption is also true because the sum s consists of leading 9's followed by 0's.

(3) Let us assume that the hypothesis is true for all $j-1 \geq k \geq 1$ with $\sigma_k \in \{0,1\}$, that is,

$$s_{j-1} = \sum_{i=1}^{j-1} X_i = c_{j-1} \cdot 10^{p+1} + \sum_{i=j}^{p} 9 \cdot 10^i + \sum_{i=1}^{j-1} y_i \cdot 10^i, \tag{A.6}$$

with $y_i \in \{0, 8, 9\}$ and c_{j-1} considers the most significant digits of the sum. Then, for the next inductive step j, we have to consider $\sigma_j = 0$ and $\sigma_j = 1$. The condition $\sigma_j = 0$ leads to $s_j = X_j + s_{j-1} = s_{j-1}$, that is, the assumption is true. For $\sigma_j = 1$ follows

$$\begin{aligned} s_j &= X_j + s_{j-1} \\ &= \sum_{i=j}^{p} 9 \cdot 10^i + c_{j-1} 10^{p+1} + \sum_{i=j}^{p} 9 \cdot 10^i + \sum_{i=1}^{j-1} y_i 10^i \\ &= \left(c_{j-1} + 1\right) \cdot 10^{p+1} + \sum_{i=j+1}^{p} 9 \cdot 10^i + 8 \cdot 10^j + \sum_{i=1}^{j-1} y_i 10^i \\ &= c_j 10^{p+1} + \sum_{i=j+1}^{p} 9 \cdot 10^i + \sum_{i=1}^{j} y_i 10^i, \end{aligned} \tag{A.7}$$

which finally proves the assertion. □

References

[1] M. F. Cowlishaw, "Decimal floating-point: algorism for computers," in *Proceedings of the 16th IEEE Symposiumon Computer Arithmetic (ARITH-16 '03)*, pp. 104–111, IEEE Computer Society, Washington, DC, USA, 2003.

[2] IEEE, "IEEE 754-2008 Standard for Floating-Point Arithmetic," September 2008.

[3] ANSI/IEEE, "ANSI/IEEE 754-1987 Standard for Radix-Independent Floating-Point Arithmetic," October 1987.

[4] L. Eisen, J. W. Ward, H. W. Tast et al., "IBM POWER6 accelerators: VMX and DFU," *IBM Journal of Research and Development*, vol. 51, no. 6, pp. 663–683, 2007.

[5] E. Schwarz and S. Carlough, "Power6 decimal divide," in *Proceedings of the Application-Specific Systems, Architectures, and Processors (ASAP '07)*, IEEE Computer Society, 2007.

[6] A. Y. Duale, M. H. Decker, H. G. Zipperer, M. Aharoni, and T. J. Bohizic, "Decimal floating-point in z9: an implementation and testing perspective," *IBM Journal of Research and Development*, vol. 51, no. 1-2, pp. 217–227, 2007.

[7] X. Li, J. W. Demmel, D. H. Bailey et al., "Design, implementation and testing of extended and mixed precision BLAS," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 152–205, 2002.

[8] U. Kulisch, *Advanced Arithmetic for the Digital Computer, Design of Arithmetic Units*, Springer, Secaucus, NJ, USA, 2002.

[9] G. Bohlender and T. Teufel, "BAPSC: a decimal floating point processor for optimal arithmetic," in *Computer Arithmetic: Scientific Computation and Programming Languages*, pp. 31–58, B. G. Teubner, Stuttgart, Germany, 1987.

[10] M. Baesler and T. Teufel, "FPGA implementation of a decimal floating-point accurate scalar product unit with a parallel fixed-point multiplier," in *Procedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 6–11, IEEE, December 2009.

[11] M. Erle and M. Schulte, "Decimal multiplication via carry-save addition," in *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 0–348, 2003.

[12] M. A. Erle, B. J. Hickmann, and M. J. Schulte, "Decimal floating-point multiplication," *IEEE Transactions on Computers*, vol. 58, no. 7, pp. 902–916, 2009.

[13] T. Lang and A. Nannarelli, "A radix-10 combinational multiplier," in *Proceedings of the 40th Asilomar Conference on Signals, Systems, and Computers (ACSSC '06)*, pp. 313–317, October 2006.

[14] A. Vázquez, E. Antelo, and P. Montuschi, "A new family of high - Performance parallel decimal multipliers," in *Proceedings of the18th IEEE Symposium on Computer Arithmetic (ARITH '07)*, pp. 195–204, June 2007.

[15] A. Vazquez, E. Antelo, and P. Montuschi, "Improved design of high-performance parallel decimal multipliers," *IEEE Transactions on Computers*, vol. 59, no. 5, Article ID 5313798, pp. 679–693, 2010.

[16] E. M. Schwarz, M. A. Erle, and M. J. Schulte, "Decimal multiplication with efficient partial product generation," in *Proceedings of the17th IEEE Symposium on Computer Arithmetic (ARITH '05)*, pp. 21–28, IEEE Computer Society, Washington, DC, USA, 2005.

[17] G. Jaberipur and A. Kaivani, "Binary-coded decimal digit multipliers," *IET Computers and Digital Techniques*, vol. 1, no. 4, pp. 377–381, 2007.

[18] G. Sutter, E. Todorovich, G. Bioul, M. Vazquez, and J.-P. Deschamps, "FPGA implementations of BCD multipliers," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 36–41, IEEE, December 2009.

[19] M. A. Erle, M. J. Schulte, and B. J. Hickmann, "Decimal floating-point multiplication via carry-save addition," in *Proceedings of the 18th IEEE Symposiumon Computer Arithmetic (ARITH '07)*, pp. 46–55, June 2007.

[20] G. Jaberipur and A. Kaivani, "Improving the speed of parallel decimal multiplication," *IEEE Transactions on Computers*, vol. 58, no. 11, Article ID 5184812, pp. 1539–1552, 2009.

[21] H. C. Neto and M. P. Vestias, "Decimal multiplier on FPGAusing embedded binary multipliers," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '08)*, pp. 197–202, September 2008.

[22] Xilinx. Virtex-II Pro and Virtex-II Pro X Platform FPGAs-Complete Data Sheet, November 2007.

[23] Xilinx, "Virtex-5 Family Overview," February 2009.

[24] M. Vazquez, G. Sutter, G. Bioul, and J. P. Deschamps, "Decimal adders/subtractors in FPGA: efficient 6-input LUT implementations," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '09)*, pp. 42–47, December 2009.

[25] C. Baumhof, *Ein Vektorarithmetik-Koprozessor in VLSATechnikzur Unterstuetzung des Wissenschaftlichen Rechnens*, 1996.

[26] Xilinx Inc, "Xilinx ISE 10.1 Design Suite Software Manuals and Help," 2008, http://www.xilinx.com.

[27] M. Schulte, N. Lindberg, and A. Laxminarain, "Performance evaluation of decimal floating-point arithmetic," in *Proceedings of the 6th IBM Austin Center for Advanced Studies Conference*, 2005.