Performance Comparison of Radix-10 Multiplier Using BCD and 2's Compliment Codes

I. Introduction

Decimal fixed-point and floating-point formats are important in financial, commercial, and user-oriented computing, where conversion and rounding errors that are inherent to floating-point binary representations cannot be tolerated [3]. The new IEEE 754-2008 Standard for Floating-Point Arithmetic [15], which contains a format and specification for decimal floating-point (DFP) arithmetic [1], [2], has encouraged a significant amount of research in decimal hardware [6], [9], [10], [28], [30]. Furthermore, current IBM Power and z/System families of microprocessors [5], [8], [23], and the Fujitsu Sparc X microprocessor [26], oriented to servers and mainframes, already include fully IEEE 754-2008 compliant decimal floating-point units (DFPUs) for Decimal64 (16 precision digits) and Decimal128 (34 precision digits) formats. Since area and power dissipation are critical design factors in state-of-the-art DFPUs, multiplication and division are performed iteratively by means of digit-by-digit algorithms [4], [5], and therefore they present low performance.

Moreover, the aggressive cycle time of these processors puts an additional constraint on the use of parallel techniques [6], [19], [30] for reducing the latency of DFP multiplication in high-performance DFPUs. Thus, efﬁcient algorithms for accelerating DFP multiplication should result in regular VLSI layouts that allow an aggressive pipelining. Hardware implementations normally use BCD instead of binary to manipulate decimal fixed-point operands and integer signiﬁcands of DFP numbers for easy conversion between machine and user representations [21], [25]. BCD encodes a number X in decimal (non-redundant radix-10) format, with each decimal digit \( X_i \in \{0, 9\} \) represented in a 4-bit binary number system. However, BCD is less efficient for encoding integers than binary, since codes 10 to 15 are unused. Moreover, the implementation of BCD arithmetic is more complicated than binary, which leads to area and delay penalties in the resulting arithmetic units. A variety of redundant decimal formats and arithmetic have been proposed to improve the performance of BCD multiplication. The BCD carry-save format [9] represents a radix-10 operand using a BCD digit and a carry bit at each decimal position. It is intended for carry-free accumulation of BCD partial products using rows of BCD digit adders arranged in linear [9], [20] or tree-like configurations [19]. Decimal signed-digit (SD) representations [10], [14], [24], [27] rely on a redundant digit set \{±a; ±b; ±c; ±d\}, \{±5; ±a; ±b\}, to allow decimal carry-free addition. BCD carry-save and signed-digit radix-10 arithmetic offer improvements in performance with respect to non-redundant BCD. However, the resultant VLSI implementation’s current technologies of multi-operand adder trees may result in more irregular layouts than binary carry-save adders (CSA) and compressor trees. Some approaches rely on binary arithmetic to perform decimal multi-operand addition and multiplication. In [6], decimal multi-operand addition is implemented using columns of binary compressors and subsequent binary-to-BCD conversions. Also, decimal multi-operand addition can be improved using binary carry-save adders and decimal doublers if digits are not represented in BCD but in certain decimal codes, namely, 4221 and 5211. These 4-bit decimal codes satisfy that the sum of the weights of the bits is equal to 9, so that all the 16 4-bit combinations represent a decimal digit in \( 0:9 \). These codes have been used to speed-up decimal multi-operand addition and multiplication [29], [30], [31]. The additional redundancy available in the 4-bit encoding is used to speed-up BCD operations while retaining the same data path width. Furthermore, these codes are self-complementing so that the 9’s complement of a digit, required for negation, is easily obtained by bit-inversion of its 4-bit representation. A disadvantage of 4221 and 5211 codes, is the use of a non-redundant radix-10 digit set \{0, 9\} as BCD. Thus, the redundancies constrained to the digit bounds, so that complex decimal multiples, such as 3X, cannot be obtained in acarry-free-way.

In this work, we focus on the improvement of parallel decimal multiplication by exploiting the redundancy of twodecimal representations: the ODDS and the redundant BCD excess-3 (XS-3) representation, a self-complementing codewith the digit set \([-3, 12]\). We use a minimally redundant digit set for the recoding of the
BCD multiplier digits, the signed-digit radix-10 recoding [30], that is, the recoded signed digits are in the set \{-5; -4; -3; -2; -1; 0; 1; 2; 3; 4; 5\}. For this digit set, the main issue is to perform the _3 multiplemultiplication without long carry-propagation (note that -2 and -5 are easy multiples for decimal [30] and that _4 is generated astwo consecutive _2 operations). We propose the use of general redundant BCD arithmetic (that includes the ODDS, XS-3 and BCD representations) to accelerate parallel BCD multiplication in two ways:

- Partial product generation (PPG). By generating positivemultiplicand multiples coded in XS-3 in a carry-freeform. An advantage of the XS-3 representation over non-redundant decimal codes (BCD and 4221/5211 [30]) is that all the interesting multiples for decimal partial product generation, including the 3X multiple, can be implemented in constant time with an equivalent delay of about three XOR gate levels. Moreover, since XS-3 is a self-complementing code, the 9’s complement of a positive multiple can be obtained by just inverting its bits as in binary.

- Partial product reduction (PPR). By performing the reduction of partial products coded in ODDS viabinary carry-save arithmetic. Partial products can be berecoded from the XS-3 representation to the ODDS representation by just adding a constant factor into the partial product reduction tree. The resultant partial product reduction tree is implemented using regular structures of binary carry-save adders or compressors. The 4-bit binary encoding of ODDS operands allows a more efficient mapping of decimal algorithms into binary techniques. By contrast, signed-digit radix-10 and BCD carry-save redundant representations require specific radix-10 digit adders.

II. HIGH-LEVEL ARCHITECTURE

The high-level block diagram of the proposed parallel architecture for _d_ _d_-digit BCD decimal integer and fixed-point multiplication is shown in Fig. 1. This architecture accepts conventional (non-redundant) BCD inputs _X, Y_ , generates redundant BCD partial products PP, and computes the BCD product _P_ = _X_ _Y_. It consists of the following three stages:

1. Parallel generation of partial products coded in XS-3, including generation of multiplicand and recoding of the multiplier operand.
2. Recoding of partial products from XS-3 to the ODDS representation and subsequent reduction.
3. Final conversion to a non-redundant 2d-digit BCD product.

![Fig 1 Combinational Radix-10 Architecture](image)

Stage 1. Decimal partial product generation: A SD radix-10 recoding of the BCD multiplier has been used. This recoding produces a reduced number of partial products that leads to a significant reduction in the overall multiplier area [29]. Therefore, the recoding of the _d_-digit multiplier _Y_ into SD radix-10 digits _Yb_d_ _1; \ldots; Yb_0 produces _d_ partial products PP[0:d] _1; \ldots; PP[0:d] _0_, one per digit; note that each Ybk recoded digit is represented in a 6-bit hot-one code to be used as control input of the multiplexers for selecting the proper multiplicand multiple, [-5X; -1X; 0X; 1X; \ldots; 5X]. An additional partial product PP[0:d] _0_ is produced by the most significant multiplier digit after the recoding, so that the total number of partial products generated is _d_ + 1.
Stage 2) Decimal partial product reduction. In this stage, the array of $d + 1$ ODDS partial products are reduced to two $2d$-digit words $(A, B)$. Our proposal relies on a binary carriesaveadder tree to perform carry-free additions of the decimal partial products. The array of $d + 1$ ODDS partial products can be viewed as adjacent digit columns of height $H_{d + 1}$. Since ODDS digits are encoded in binary, the rules for binary arithmetic apply within the digit bounds, and only carries generated between radix-10 digits (4-bit columns) contribute to the decimal correction of the binary sum. That is, if a carry out is produced as a result of a 4-bit (modulo 16) binary addition, the binary sum must be incremented by 6 at the appropriate position to obtain the correct decimal sum (modulo 10 addition).

Stage 3) Conversion to (non-redundant) BCD. We consider the use of a BCD carry-propagate adder [29] to perform the final conversion to a non-redundant BCD product $P = A + B$. The proposed architecture is a 2d-digit hybrid parallel prefix/carry-select adder, the BCD Quaternary Tree adder (see Section 6). The sum of input digits $A$, $B$ at each position $i$ has to be in the range $\frac{1}{10}$; $18$, so that at most one decimal carry is propagated to the next position $i + 1$ [22]. Furthermore, to generate the correct decimal carry, the BCD addition algorithm implemented requires $A_i + B_i$ to be obtained in excess-6. Several choices are possible. We opt for representing operand $A$ in BCDexcess-6 ($A_i \in \{0; 9\}$), and $B$ coded in BCD ($B_i \in \{0; 9\}$).

### III. DECIMAL PARTIAL PRODUCT GENERATION

The partial product generation stage comprises the recoding of the multiplier to a SD radix-10 representation, the calculation of the multiplicand multiples in XS-3 code and the generation of the ODDS partial products. The SD radix-10 encoding produces $d$ SD radix-10 digits $Y_{bk} \in \{-5; 5\}$, with $k = 0; \ldots; d - 1$, $Y_{d - 1}$ being the most significant digit (MSD) of the multiplier [29]. Each digit $Y_{bk}$ is represented with a 5-bit hot-one code ($Y_{1k}; Y_{2k}; Y_{3k}; Y_{4k}; Y_{5k}$) to select the appropriate multiple $f_1X; \ldots; 5X$ with a 5:1 mux and a sign bit $Y_{sk}$ that controls the negation of the selected multiple. The negative multiples are obtained by ten's complementing the positive ones. This is equivalent to taking the nine's complement of the positive multiple and then adding

1. As we have shown in Section 2, the nine's complement can be obtained simply by bit inversion. This needs Fig. 1. Combinational SD radix-10 architecture.

As we have shown in Section 2, the nine’s complement can be obtained simply by bit inversion. This needs Fig. 1. Combinational SD radix-10 architecture.

![Fig 1. Combinational SD radix-10 architecture.](image)

Fig 1. Combinational SD radix-10 architecture.

![Fig 2. Radix-10 generation of partial products.](image)

Fig 2. Radix-10 generation of partial products.

The $d$ least significant partial products $PP_{d-1}; \ldots; PP_0$ are generated from digits $Y_{bk}$ by using a set of 5:1 muxes, as shown in Fig. 2. The xor gates at the output of the mux invert the multiplicand multiple, to obtain its 9’s complement, if the SD radix-10 digit is negative ($Y_{sk} = 1$). On the other hand, if the signals ($Y_{1k}; Y_{2k}; Y_{3k}; Y_{4k}; Y_{5k}$) are all zero then $PP_{5k} = 0$, but it has to be coded in XS-3 (bit encoding 0011). Then, to set the two least significant bits to 1, the input to the XOR gate is $Y_{sk}$. In addition, the partial product signs are encoded into their MSDs (see Section 4.2). The generation of the most significant partial product $PP_{sd}$ is described in Section 4.4, and only depends on $Y_{sd-1}$, the sign of the most significant SD radix-10 digit.
IV. Generation of the Multiplicand Multiples

We denote by \( N_X = \{ 1 \times X; 2 \times X; 3 \times X; 4 \times X; 5 \times X \} \), the set of multiplicand multiples coded in the XS-3 representation, with digits \( N_{X_i} \in \{ 0; 1; 2; 3; 4; 5; 6; 7; 8; 9 \} \) being \( \frac{1}{2} N_{X_i} \in \{ 0; 1; 2; 3; 4; 5; 6; 7; 8; 9 \} \) the corresponding value of the 4-bit binary encoding of \( N_{X_i} \) in the high-level block diagram of the multiples generation with just one carry propagation. This is performed in two steps:

1) digit recoding of the BCD multiplicand digits \( X_i \) into a decimal carry \( 0 \leq T_i \leq T_{max} \) and a digit \( 3 < D_i < 12 - T_{max} \), such as

\[
D_i + 10 \times T_i = \left( N - X_i \right) + 3;
\]

(5) being \( T_{max} \) the maximum possible value for the decimal carry.

2) The decimal carries transferred between adjacent digits are assimilated obtaining the correct 4-bit representation of XS-3 digits \( N_{X_i} \), that is

\[
\text{Fig 3} \quad \text{Generation of decimal multiples } N_X
\]

Most-Significant Digit Encoding

The MSD of each PP\(_{0\ldots k} \), PP\(_{d\ldots k} \), is directly obtained in the ODDS representation. Note that these digits store the carries generated in the computation of the multiplicand multiples and the sign bit of the partial product. For positive partial products we have

\[
PP_d[k] = T_{d-1}
\]

Correction Term

The resultant partial product sum has to be corrected of the critical path by adding a pre-computed term, \( f(d) \) which only depends on the format precision \( d \). This term has to gather: (a) the -8 constants that have not been included in the MSD encoding and (b) a -3 constant for every XS-3 partial product digit (introduced to simplify the nine's complement operation). Actually, the addition of these -3 constants is equivalent to convert the XS-3 digits of the partial products to the ODDS representation.

\[
f_c(d) = -8 \sum_{k=0}^{d-1} 10^{k+d} - 3 \times \left( \sum_{i=0}^{d-1} (i+1)10^i + \sum_{i=0}^{d-2} (d-1-i)10^{i+d} \right).
\]

Partial Product Array

In each column several components can be observed. Digits labeled with O represent the redundant excess-3BCD digits in the set \( \{ 0; 1; 2; 3; 4; 5; 6; 7; 8; 9 \} \). Digits labeled with \( Sk \) represent the MSD of each partial product, PP\(_{d\ldots k} \) (see Section 4.2). The 16 least significant digits of the correction term \( f(d) \) are placed at the least significant position of each row after being added to \( Y_k \), to complete the 10's complement in case of a negative partial product; thus \( H_k \equiv Y_k + f_0; 3; 7 \) (digitwise addition, out of the critical path), so that \( H_k \equiv 2 \times O \). Note that the negative bit -1032 is canceled with the carry out of the partial product sum in excess. The 16 leading digits of the correction term, \( \frac{1}{2} f((16)) \cdot d \), are added to the most significant partial product PP\(_{d\ldots d} \). Thus, in parallel with the evaluation of the multiplicand products we compute \( X F \equiv \frac{1}{2} X \equiv [f(16)] \cdot d \) in the ODDS representation (note that this computation does not involve a carry propagation and it is out of the critical path). Digits labeled as F in Fig. 4, represent the most significant partial product, PP\(_{d\ldots d} \), where

\[
PP_d[d] = X F \text{ if } Y_{s_{d-1}} = 1 \text{ and } PP[d] = [f_c(16)]_d \text{ if } Y_{s_{d-1}} = 0.
\]
DECIMAL PARTIAL PRODUCT REDUCTION

The PPR tree consists of three parts: (1) a regular binary CSA tree to compute an estimation of the decimal partial product sum in a binary carry-save form \((S, C)\), (2) a sum correction block to count the carries generated between the digit columns, and (3) a decimal digit 3:2 compressor which increments the carry-save sum according to the carry count to obtain the final double-word product \((A; B)\), \(A\) being represented with excess-6 BCD digits and \(B\) being represented with BCD digits. The PPR tree can be viewed as adjacent columns of \(h\) ODDS digits each, \(h\) being the column height (see Fig. 4), and \(h \leq d + 1\).

The high-level architecture of a column of the PPR tree (the \(i\)th column) with \(h\) ODDS digits in \([0, 15]\) (4 bits per digit). Each digit column of the binary CSA tree reduces the \(h\) input digits and \(n\) in input carry bits, transferred from the previous.

![Fig 4 Decimal partial product generator](image)

V. FINAL CONVERSION TO BCD

The selected architecture is a 2d-digit hybrid parallel prefix/carry-select adder, the BCD Quaternary Tree adder [29]. The delay of this adder is slightly higher than the delay of a binary adder of \(8d\) bits with a similar topology. The decimal carries are computed using a carry prefix tree, while two conditional BCD digit sums are computed out of the critical path using 4-bit digit adders which implement \(\frac{1}{2}A_i \cdot \bar{B}_i \cdot \bar{B}_j\) and \(\frac{1}{2}A_i \cdot \bar{B}_i \cdot B_j\). These conditional sums correspond to each one of the carry input values. If the conditional carry out from a digit is one, the digit adder performs a _6 subtraction. The selection of the appropriate conditional BCD digit sums is implemented with a final level of 2:1 multiplexers. To design the carry prefix tree we analyzed the signal arrival profile from the PPRT tree, and considered the use of different prefix tree topologies to optimize the area for the minimum delay adder.

![Fig 5 High level architecture for radix Multiplier](image)
TWO'S COMPLIMENTS

One of the most commonly used schemes is radix-4 MBE, which it allows for the reduction of the size of the partial product array by almost half, and it is very simple to generate the multiples of the multiplicand. The approach is general and, for the sake of clarity, will be explained through the practical case of $8 \times 8$ multiplication (as in the previous figures). As briefly outlined in the previous sections, the main goal of our approach is to produce a partial product array with a maximum height of $\lceil n/2 \rceil$ rows, without introducing any additional delay. Let us consider, as the starting point, the form of the simplified array as reported in Fig. 2, for all the partial product rows except the first one. As depicted in Fig. 3a, the first row is temporarily considered as being split into two subrows, the first one containing the partial product bits (from right to left) from pp00 to pp80 and the second one with two bits set at “one” in positions 9 and 8. Then, the bit neg3 related to the fourth product row, is moved to become a part of the second subrow. The key point of this “graphical” transformation is that the second subrow containing also the bit neg3, can now be easily added to the first subrow, with a constant short carry propagation of three positions (further denoted as “3-bits addition”), a value which is easily shown to be general, i.e., independent of the length of the operands, for square multipliers.

Generation of the three most significant bit weights of the first row, plus addition of the last neg bit, possible implementations can use a replication of three times cascaded by the circuit of Fig. 5 to add the neg signal. Parallel generation of the other bits of the first row; possible implementations can use instances of the circuitry depicted in Fig. 6, for each bit of the first row, except for the three most significant: Parallel generation of the bits of the other rows; possible implementations can use the circuitry replicated for each bit of the other rows.
Conclusion
A fast radix-10 multiplier is implemented using BCD and 2’s Compliment codes where the fast multipliers are very much essential in fast DSP and Micro-processor architectures. In conventional Multipliers are overheads of architecture in VLSI circuits. So a comparison was made with both the codes using BCD and Two’s compliment. A post routed simulation shows a latency reduction and performance improvement by 24%. This shows the efficient radix multiplier design.
References

[1] “Fast Radix-10 Multiplication Using Redundant BCD Codes” Alvaro Vazquez, Member, IEEE, Elisardo Antelo, and Javier D. Bruguera, Member, IEEE.


