2008 International Symposium on Parallel and Distributed Processing with Applications

# Fixed Point Decimal Multiplication using RPS Algorithm

Rekha K. James, Shahana T. K, K. Poulose Jacob,
*Cochin University of Science and Technology*
Kochi, Kerala, India
{rekhajames, shahanatk, kpj}@cusat.ac.in

Sreela Sasi
*Gannon University*
Erie, PA, USA
sasi001@gannon.edu

## Abstract

*Decimal multiplication is an integral part of financial, commercial, and internet-based computations. A novel design for single digit decimal multiplication that reduces the critical path delay and area for an iterative multiplier is proposed in this research. The partial products are generated using single digit multipliers, and are accumulated based on a novel RPS algorithm. This design uses* n *single digit multipliers for an* n × n *multiplication. The latency for the multiplication of two* n*-digit Binary Coded Decimal (BCD) operands is (*n + 1*) cycles and a new multiplication can begin every* n *cycle. The accumulation of final partial products and the first iteration of partial product generation for next set of inputs are done simultaneously. This iterative decimal multiplier offers low latency and high throughput, and can be extended for decimal floating-point multiplication.*

## 1. Introduction

Nowadays, decimal arithmetic is receiving significant attention in the financial, commercial, and internet-based applications. These applications often store data in decimal format. Currently, general purpose computers do decimal computations using binary arithmetic. But, a number of decimal numbers such as 0.2 cannot be represented precisely in binary. In this world of precision, such errors generated by conversion between decimal and binary formats are no more tolerable. Recently, support for decimal arithmetic has received increased attention due to the growing importance in financial analysis, banking, tax calculation, currency conversion, insurance, telephone billing and accounting which cannot tolerate such errors. This can be overcome by using a decimal arithmetic and logic unit. Decimal arithmetic operations are typically more complex, slower and occupy more area leading to more power and less speed when implemented in hardware. Hence, the major consideration while implementing decimal arithmetic is to enhance its speed and reduce area as much as possible. Due to the growing importance of decimal arithmetic, standard specifications are recently added to the draft revision of the IEEE 754 Standard for Floating-Point Arithmetic.

Decimal multipliers are typically implemented using an iterative approach because of their complexity. Usually, the entire multiplicand is multiplied by one multiplier digit to generate a partial product in each cycle. The partial product is added to an intermediate product register that holds the previously accumulated partial products. In an iterative decimal multiplier presented in [1], decimal partial products are generated by creating two partial products for each multiplier digit. Multiplying two *n* digit Binary Coded Decimal (BCD) numbers requires *n* iterations, where all iterations consist of two binary carry-save additions and three decimal corrections. After *n* iterations, the carry and sum are added using a decimal carry-propagate adder to produce the final product. The multiplier presented in [2] generates the partial products by the costly retrieval of product of BCD digits from look-up tables. Several existing designs for decimal multiplication generate and store multiples of the multiplicand before partial product generation, and then use the multiplier digits to select the appropriate multiple as the partial product [3, 4]. The multiplier presented in [4] makes use of a secondary set of multiples generated using combinational logic. Iterative additions are performed in two pipeline stages, which allows for a higher frequency of operation. The latency of this multiplier is (*n + 4*) cycles and a new multiplication can begin every (*n + 1*) cycle. The multiplier in [5] stores intermediate product digits in a less restrictive, redundant format called the overloaded decimal representation that reduces the delay of the iterative portion of the multiplier. These approaches are based on either slow accumulation of easy multiples or costly retrieval of product of BCD digits from look-up tables. An

978-0-7695-3471-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ISPA.2008.89

alternative approach is to generate the partial product as needed. Generating the partial products as needed is an ideal approach for three reasons as enumerated in [6]. The use of decimal digit-by-digit multipliers for partial product generation leads to less number of cycles, less wiring and do not require registers to store multiples of the multiplicand. The algorithm presented in [6] reduces the complexity of partial product generation by employing a recoding scheme to restrict the magnitude range of the operand digits. Further, by restricting the range of each digit in the partial product, the complexity of partial product accumulation is also significantly reduced. An integral building block of a decimal digit by digit multiplier is the single digit multiplier. The single digit multiplier in [7] uses a standard 4 × 4 unsigned binary multiplier that generates an 8-bit binary output which needs to be corrected to two BCD digits.

This paper presents a novel design for single digit decimal multiplication to reduce the critical path delay and area, which allows for a fast iterative multiplier design. The partial products are generated by single digit multipliers, and accumulated using a novel RPS algorithm. This iterative decimal multiplier offers low latency and high throughput.

The organization of the paper is as follows: Initially, a new approach for single digit multiplication is discussed. A decimal fixed point multiplier is then proposed using single digit multipliers. A novel RPS algorithm is also proposed to select the inputs for partial product generation and accumulation. Finally, the paper concludes by tabulating an area and delay analysis of the proposed design using logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library.

## 2. BCD Digit Multiplication

A key component of a fixed-point multiplier is a single digit multiplier that multiplies an *n*-digit multiplicand, *A*, by an *n*-digit multiplier, B producing a *2n*-digit product, P. The single digit multiplier accepts two BCD inputs (A, B) which can take a value [0-9]. It realizes a function F(A, B), giving a product in the range [0, 81] represented by two BCD digits. There are one hundred possible combinations of inputs for multiplication, out of which only 4 combinations require 4 × 4 multiplication, 64 combinations need 3 × 3 multiplication, and the remaining 32 combinations use either 3 × 4 or 4 × 3 multiplication. The proposed design makes use of this property. The single digit multiplier consists of two parts: a binary multiplier that gives a binary product $p_{(7-0)}$, and a binary to BCD converter.

### 2.1 Binary Multiplier

The binary multiplier consists of a 3 × 3 multiplier, a 4 × 3 multiplier and a 4 × 4 multiplier. Figures 1, 2 and 3 show the 3 × 3, 4 × 3 and 4 × 4 multiplication for BCD inputs respectively. In 4 × 3 multiplication for BCD inputs, one of the inputs is either $8(1000_2)$ or $9(1001_2)$. So, the 4 × 3 multiplier gets simplified to three 2-input AND gates. In 4 × 4 multiplication for BCD inputs, both inputs are either $8(1000_2)$ or $9(1001_2)$. So the 4 × 4 multiplier gets simplified to a half adder (a 2-input AND gate and a 2-input XOR gate).

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | $x_2$ | $x_1$ | $x_0$ | × |
| | | | $y_2$ | $y_1$ | $y_0$ | |
| | | | $x_2y_0$ | $x_1y_0$ | $x_0y_0$ | |
| | | $x_2y_1$ | $x_1y_1$ | $x_0y_1$ | | |
| | $x_2y_2$ | $x_1y_2$ | $x_0y_2$ | | | |
| $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ | |

**Figure 1 : 3 × 3 Multiplication**

| | | | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 0 | 0 | $x_0$ | × |
| | | | $y_2$ | $y_1$ | $y_0$ | |
| | | $y_0$ | 0 | 0 | $x_0y_0$ | |
| | $y_1$ | 0 | 0 | $x_0y_1$ | | |
| $y_2$ | 0 | 0 | $x_0y_2$ | | | |
| $y_2$ | $y_1$ | $y_0$ | $x_0y_2$ | $x_0y_1$ | $x_0y_0$ | |

**Figure 2: 4 × 3 Multiplication of BCD inputs**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | 1 | 0 | 0 | $x_0$ | × |
| | | | 1 | 0 | 0 | $y_0$ | |
| | | | $y_0$ | 0 | 0 | $x_0y_0$ | |
| 1 | 0 | 0 | $x_0$ | | | | |
| 1 | 0 | $x_0y_0$ | $x_0 \oplus y_0$ | 0 | 0 | $x_0y_0$ | |

**Figure 3: 4 × 4 Multiplication of BCD inputs**

The 3 × 3 and 4 × 3 multipliers give a 6-bit binary product, while a 4 × 4 multiplier produces a 7-bit binary result. The binary to BCD converter, following the binary multiplier, will be a 7-input converter. The design can be further simplified by using separate converters for 6-bit and 7-bit products. The 6-bit converter converts the binary output of the 3 × 3

multiplier or 4 × 3 multiplier outputs to its corresponding BCD. Instead of using a 7-bit binary to BCD converter, the 4 × 4 multiplier is designed to produce an 8-bit BCD output as shown in Figure 4. The 4 × 4 multiplier and the binary to BCD conversion circuit of its product now gets reduced to a 2-input AND, NAND, XOR and NOR gates.

| | | | |
|---|---|---|---|
| | 1 | 0 | 0 $x_0$ × |
| | 1 | 0 | 0 $y_0$ |
| --- | --- | --- | --- |
| $x_0y_0$ | $(x_0y_0)'$ | $(x_0y_0)'$ | $x_0 \oplus y_0$ |
| 0 | $(x_0 + y_0)'$ | $x_0 \oplus y_0$ | $x_0y_0$ |

**Figure 4: 4 × 4 Multiplication of BCD inputs generating 8-bit BCD output**

## 2.2 6-bit Binary to BCD converter

Binary product can be converted to an equivalent BCD by a six-input, eight-output combinational logic. Although the general binary-to-BCD conversion is extensively addressed in the literature [8–10], a special, simpler and faster, binary-to-BCD converter depicted in [7] for a 6-bit input is used in this proposed design. The first row in Figure 5 shows the BCD weights. The weights of $p_3$, $p_2$, $p_1$ and $p_0$ are the same as the corresponding weights in the original binary number $p_{(5\text{-}0)}$. But, weights 16 and 32 of $p_4$ and $p_5$ have been decomposed to (10, 4, 2) and (20, 10, 2) respectively. The three overloaded decimal digits in the right four columns may be added, by an overloaded decimal adder to get the overloaded sum ($D_3D_2D_1D_0$) and a carry. In a standard BCD representation, the bit combinations $A_{16}$ - $F_{16}$ are invalid BCD digits. The overloaded decimal representation allows 4-bit digits to have any value from $0_{16}$ - $F_{16}$, even though the base of the number is still ten. This reduces the overhead of correcting sum digits when the sum is an invalid BCD and sum correction is only performed when sixteen is exceeded. When the final digits are formed, each overloaded decimal digit is corrected back to BCD by adding six to the digit, if it is in the range $A_{16}$ - $F_{16}$. The carry is added to the two BCD digits ("0 0 $p_5$ $p_4$" and "0 0 0 $p_5$") in the left four columns leading to ($D_7D_6D_5D_4$).

The block diagram of the proposed single digit BCD multiplier is shown in Figure 6. The 4-bit 2:1 multiplexer selects the inputs to 4 × 3 multiplier depending on $x_3$ bit. The 6-bit 2:1 multiplexer does the selection of 3 × 3 multiplier output or the output of the 4 × 3 multiplier depending on the status of $x_3$ and $y_3$ bits. If $x_3$ and $y_3$ are different then the output of 4 × 3 multiplier is passed to the BCD converter, else the output of 3 × 3 multiplier is passed. After the 6-bit binary to BCD conversion the third multiplexer (8-bit 2:1 vector mux) selects the BCD converted output or the output of the 4 × 4 multiplier output (which gives an 8-bit BCD result) depending on $x_3$ and $y_3$ bits. If both are '1' then the 4 × 4 multiplier output is selected, else the BCD converter output is passed as the final product.

| ***80*** | ***40*** | ***20*** | ***10*** | ***8*** | ***4*** | ***2*** | ***1*** |
|---|---|---|---|---|---|---|---|
| 0 | 0 | $p_5$ | $p_4$ | $p_3$ | $p_2$ | $p_1$ | $p_0$ |
| 0 | 0 | 0 | $p_5$ | 0 | $p_4$ | $p_4$ | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | $p_5$ | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| $D_7$ | $D_6$ | $D_5$ | $D_4$ | $D_3$ | $D_2$ | $D_1$ | $D_0$ |

**Figure 5: The principle of 6-bit binary to overloaded BCD conversion**

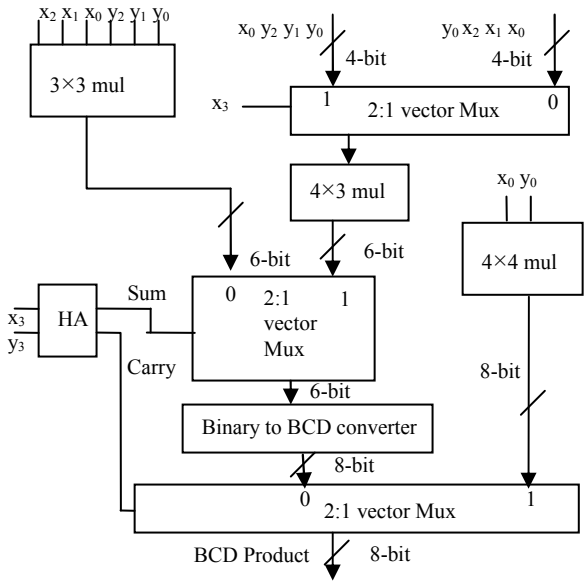


**Figure 6: Single digit BCD multiplier**

A comparison of the proposed design with the existing design in [7] in terms of area and critical path delay is done with the logic synthesis tool Leonardo Spectrum from Mentor Graphics Corporation using ASIC Library 0.18 micron, 1.8 V CMOS technology, and is tabulated in Table 1. The table shows that the proposed design has reduced area and delay compared to the existing one in [7].

**Table 1: Comparison of area and delay of single digit BCD multiplier implementations**

| Type of Multiplier | Area ($\mu m^2$) | % reduction | Delay (ns) | % reduction |
|---|---|---|---|---|
| Proposed multiplier | **489** | **8%** | **7.56** | **18%** |
| Multiplier in [7] | 532 | | 9.26 | |

## 3. Decimal Fixed Point Multiplication

The fixed point multiplier unit takes two *n*-digit operands, calculates $n^2$ partial products and returns their sum as a *2n*-digit integer. There are two main stages in the fixed-point multiplier design: generation of partial products and reduction of partial products. In the first stage of the process, the RPS algorithm selects appropriate inputs for generation of partial products using *n* single digit multipliers. After the generation of partial products they are reduced along with the carry of previous addition using multi-operand decimal adders. The process is repeated n times to generate a 2n digit product after the $(n+1)^{th}$ cycle. Many techniques have been developed to speed up the process of decimal addition. Direct decimal addition is one of the efficient techniques for two-operand decimal addition [11]. Erle and Schulte proposed a variant of direct decimal addition to produce intermediate results in a decimal carry-save format that can be used in an iterative decimal multiplier [12]. In another approach, proposed by Ohtsuki et al., a correction value of six is added to each digit of the first partial product using a binary carry-save adder [13]. Shirazi et al. proposed a technique for constant time decimal addition, called Redundant Binary Coded Decimal (RBCD) [14, 15]. Kenney and Schulte introduced three algorithms for performing fast decimal addition on multiple BCD operands: non-speculative tree, double correction speculation array and single correction speculation array [16]. The non-speculative tree algorithm that gives the minimum delay with same area of the three algorithms is the best suited for multi-operand decimal addition and is made use of in this research. In [17] a new scheme is proposed to obtain the sum of each decimal column via a network of carry-free adders and converting the sum into decimal format via a fast binary to decimal converter.

The complete process is formulated as RPS algorithm and is given below.

### 3.1 RPS Algorithm

The steps for RPS algorithm to multiply two *n*-digit numbers are as follows.

Step 1: Initialize i, j, k, m and a mod-*n* counter to 0. ( i - to select $A_i$, j - to select $B_j$)

Step 2: For iterations = 0 to *n*-1, repeat steps (3) to (6)

Step 3: For BCD-digit multipliers, c = 0 to *n*-1, repeat steps (4) to (5)

Step 4: Select $A_i$ and $B_j$ as inputs to the $c^{th}$ single digit multiplier

Step 5: If i = 0, then

If k = *n*, then i = k, m = m+1, j = m

Else k = k+1, i = k, j = 0, End

Else i = i-1, j = j+1, End

Step 6: Perform multiplication on *n* single digit multipliers and store the result.

Step 7: Increment counter, Enable multi-operand BCD addition

Multi-operand BCD addition:

$Adder_x = (\sum P_{yzL} + \sum P_{abH}) + C_{Ladder\ (x-1)} + C_{Hadder\ (x-2)}$

If x < *n*, then

y = x to 0, z = 0 to x, a = (x-1) to 0, b = 0 to ( x-1)

If x = *n*, then

y = (*n*-1) to x-(*n*-1), z = x-(*n*-1) to (*n*-1), a = (x-1) to 0, b = 0 to (x-1)

If x > *n*, then

y = (*n*-1) to x-(*n*-1), z = x-(*n*-1) to (*n*-1), a = (*n*-1) to (x-*n*), b = (x-*n*) to (*n*-1)

$C_{Ladder\ (x-1)}$ is the lower carry digit from the previous addition.

$C_{Hadder\ (x-2)}$ is the higher carry digit from the addition prior to previous addition.

The algorithm is explained considering a (7-digit × 7-digit) fixed point multiplication. This example is considered since it is an integral component of a 32-bit decimal floating point multiplication that has 7 significand digits. A (7-digit × 7-digit) multiplication results in 7 × 7 (49) partial products, each having 2 digits given by $P_{ijL}$ and $P_{ijH}$. The proposed design makes use of only seven single digit BCD multipliers. This design generates those partial products which are necessary for early accumulation using the RPS algorithm instead of generating the partial product digits of the complete multiplicand with the least significant digit of the multiplier. This is shown in Figure 7. The first iteration generates seven partial products as seen from the right most end of the partial product array of Figure 7.

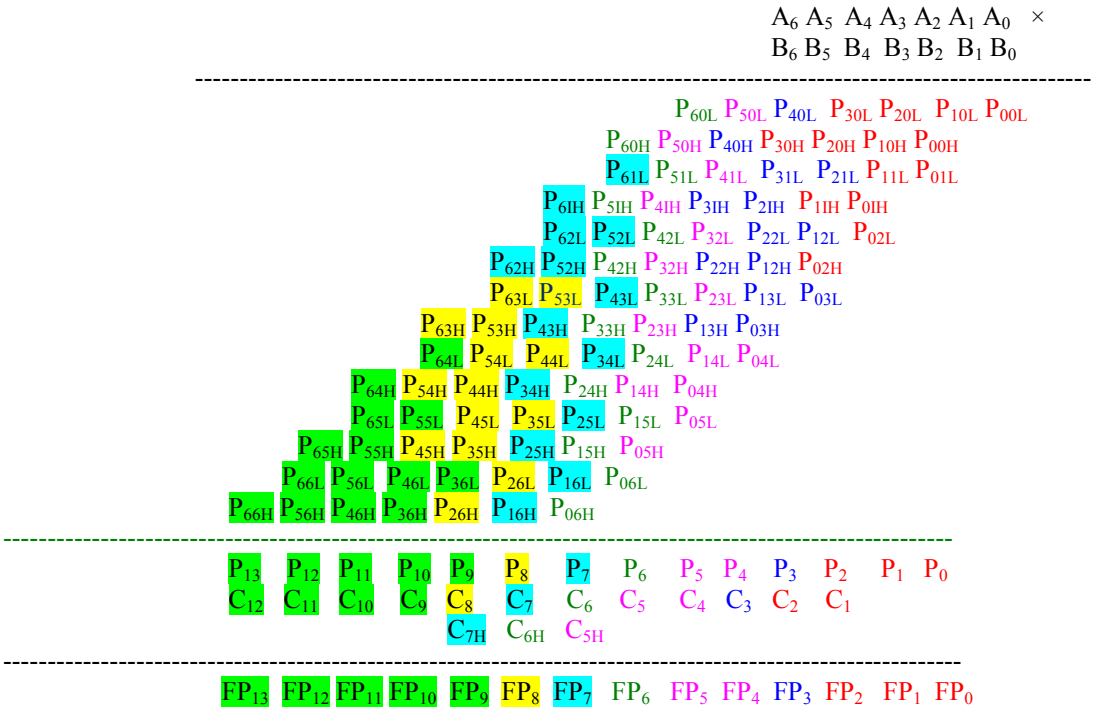


**Figure 7: Partial product generation and accumulation in different cycles**

Hence after the first cycle, the partial products generated are $P_{00}$, $P_{10}$, $P_{01}$, $P_{20}$, $P_{11}$, $P_{02}$, $P_{30}$ (both $P_L$ and $P_H$), as shown in red in the array.Similarly during the second cycle, seven more partial products are generated. Simultaneously those partial products which were generated in first cycle are added using multi-operand decimal adders to generate the final products $FP_0$, $FP_1$ and $FP_2$. This process is repeated for all seven cycles. The final product is ready after the eighth cycle. The block diagram of a 7-digit fixed point decimal multiplier is shown in Figure 8. The controller block uses the RPS algorithm to determine the flow of inputs to single digit multipliers for each cycle. The detailed design of the controller block is shown in Figure 9.

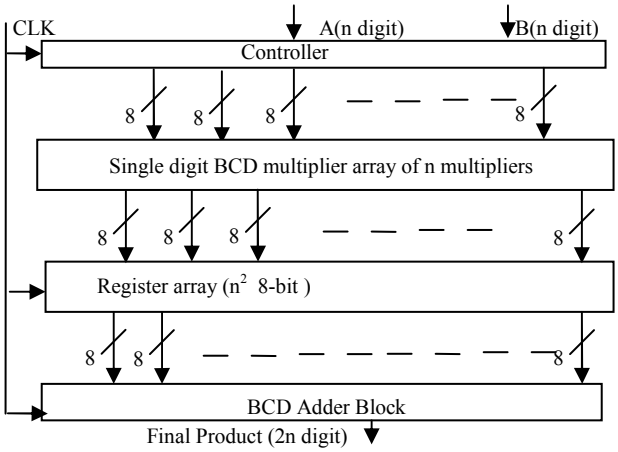


**Figure 8: Fixed point decimal multiplier**

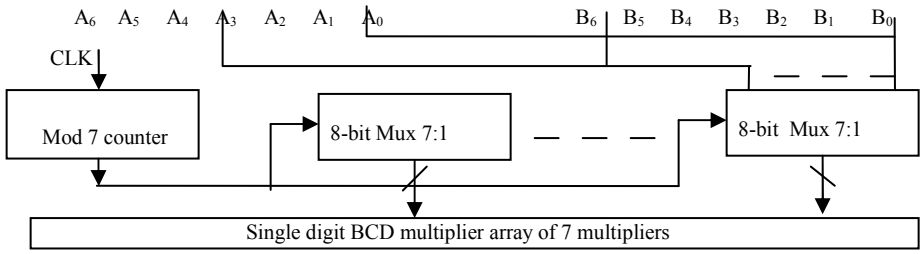


**Figure 9: Fixed point decimal multiplier controller block**

The controller block is followed by an array of n single digit BCD multipliers that give the first set of partial products. These are stored in *n* registers of an *n* × *n* 8-bit register array as shown in Figure 10. During the first cycle, data is stored in registers that are marked as red. The inputs to the multi-digit BCD adders are selected from this stored data using the RPS algorithm, as shown in Figure 11.

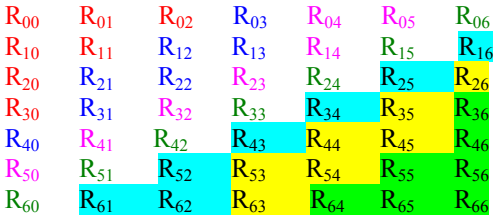


**Figure 10: Register array for storing output of BCD-digit multiplication**

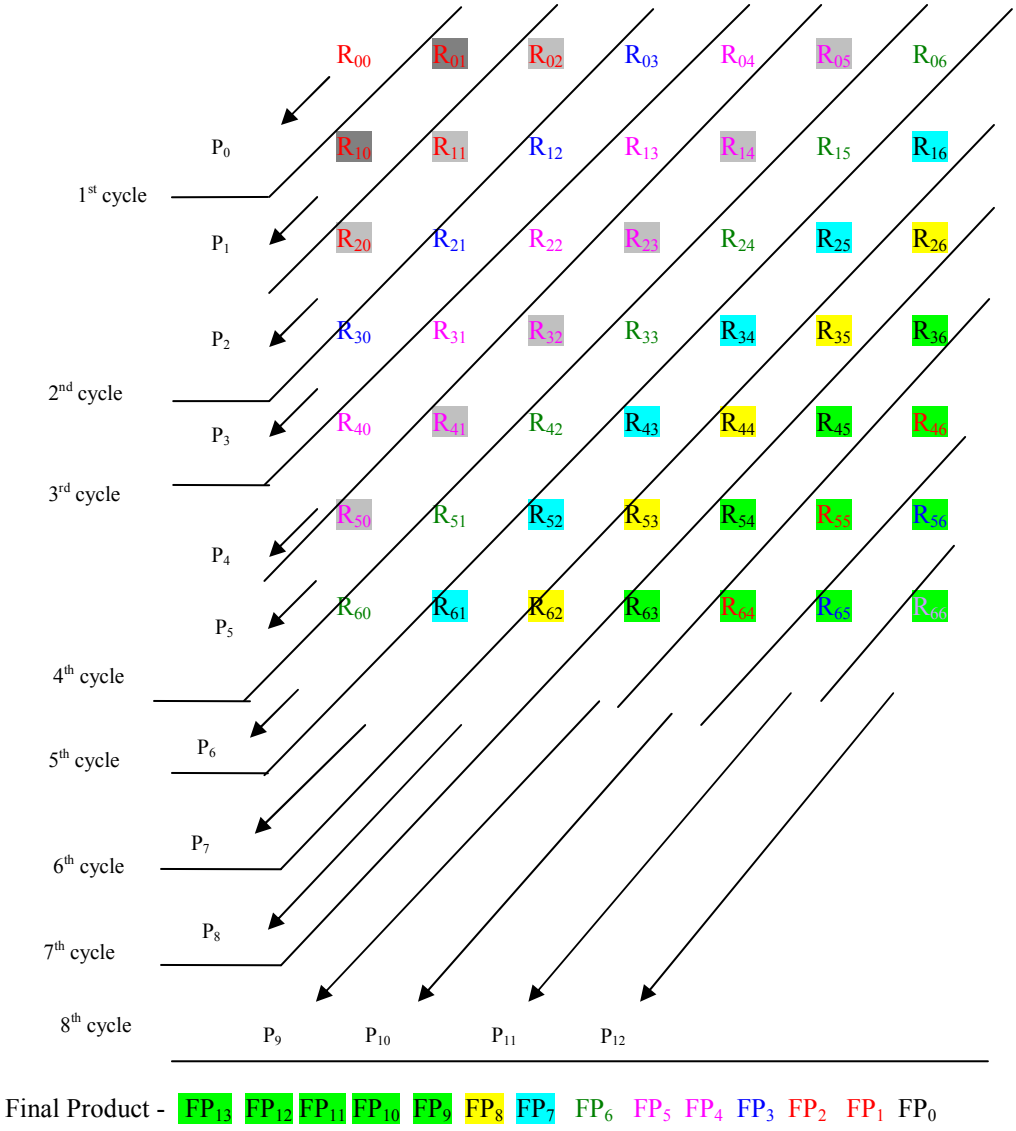


**Figure 11: Selecting inputs to multi-operand BCD addition**

The complete multi-operand BCD adder array is shown in Figure 12. During the eighth cycle $FP_9$, $FP_{10}$, $FP_{11}$, $FP_{12}$, and $FP_{13}$ are generated using 11-operand (9-partial products and the two carries ($C_8$ and $C_{7H}$) that are generated in previous cycles), 7-operand, 5-operand, 3-operand decimal adders, and a 4-digit high speed decimal adder. The 4-digit high speed decimal adder is used to add the carry generated. The maximum depth of addition occurs at the eighth cycle, and this determines the clock frequency. Using this approach, when multiplying two *n*-digit operands to produce a 2*n*-digit product, the worst-case latency is *n + 1* cycles, and initiation interval is *n* cycles.

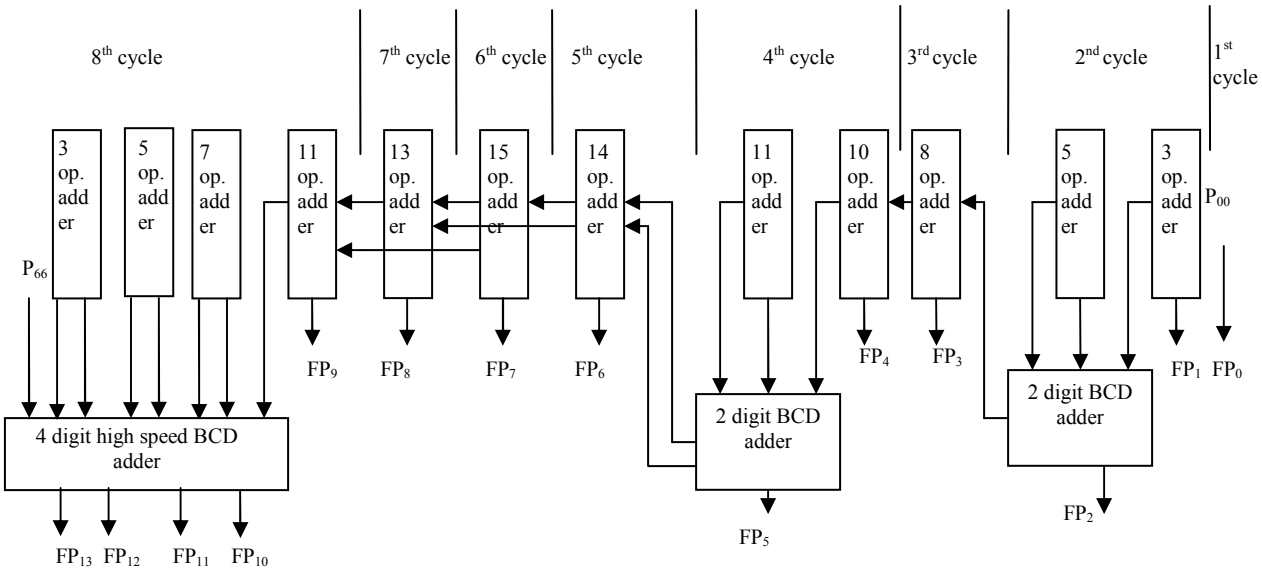


**Figure 12: BCD adder array**

## 4. Synthesis Results

The proposed decimal fixed point multiplier was coded for a (7-digit × 7-digit) multiplier in VHDL, and synthesized to evaluate the area and delay of the design. Synthesis was done using Leonardo Spectrum from Mentor Graphics Corporation with ASIC Library of 0.18 micron, 1.8 V CMOS technology. An area and delay breakdown for an approximate contribution of major components of the design shown in Figure 8 is given in Table 2. Even though the delay for the complete circuit is 29.95ns, the latency is less since the multi-operand BCD addition takes place simultaneously with the single digit multiplication of next set of inputs.

The proposed design differs from the iterative approach using easy multiples for partial product generation for hardware realization of BCD multipliers. The design in [7] uses similar approach, and so this is also synthesized in the same environment as the proposed multiplier. The proposed design of a single digit multiplier array block gives a reduction of 8% in area and 18% in delay compared to the multiplier in [7].

**Table 2: Area and Delay for different stages of Decimal Fixed Pont Multiplier (7-digit × 7-digit)**

| Component | Area | | Delay | |
|---|---|---|---|---|
| | $\mu m^2$ | % | ns | % |
| Controller | 2719 | 15.06 % | 4.85 | 16.19% |
| Single digit multiplier array | 3423 | 18.96 % | 7.56 | 25.24% |
| BCD adder array | 5791 | 32.08 % | 12.93 | 43.17% |
| Register array | 6114 | 33.87 % | 4.61 | 15.39% |
| Fixed point multiplier | 18047 | 100% | 29.95 | 100% |

## 5. Conclusion

This paper proposed a novel single digit BCD multiplier cell that can be used in iterative BCD multiplier circuits. It is demonstrated that this design gains an 8% savings in the area and 18% savings in delay compared to the existing design of [7]. This design leads to more regular VLSI implementation, and does not require special registers for storing easy multiples. A new RPS algorithm is used to generate and accumulate the partial products in an efficient manner for fixed point decimal multiplication. The design was validated using 7-digit × 7-digit fixed point decimal multiplication that is required for a 32-bit floating point decimal multiplication. The latency for the multiplication of two n-digit BCD operands is ($n$ + $1$) cycles, and a new multiplication can begin every '$n$' cycle. Inclusion of parallelism improved the throughput.

Future research focuses on incorporating pipelining in the design and for reducing the size of register arrays used for storing the partial products.

## 6. References


[1] T. Ohtsuki, et al., "Apparatus for Decimal Multiplication," U.S. Patent, June 1987, #4,677,583

[2] Ueda, T.: 'Decimal multiplying assembly and multiply module'.U.S. Patent 5379245, January 1995

[3] F. Y. Busaba, C. A. Krygowski, W. H. Li, E. M. Schwarz, and S. R. Carlough, "The IBM Z900 Decimal Arithmetic Unit," in *Asilomar Conference on Signals, Systems, and Computers*, vol. 2, pp. 1335– 1339, November 2001

[4] M. A. Erle and M. J. Schulte, "Decimal Multiplication Via Carry-Save Addition," IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, pp. 348-358, June 2003

[5] R. D. Kennedy, M. J. Schulte and M. A. Erle, "A High-Frequency Decimal Multiplier," IEEE 14th International IEEE international conference on Computer Design (ICCD'04), pp. 22-29, Oct 2004

[6] Erle, M.A. Schwarz, E.M. Schulte, M.J, "Decimal multiplication with efficient partial product generation**",** 17th IEEE Symposium on Computer Arithmetic, 2005. ARITH-17 2005.
pp. 21- 28, June 2005

[7] Jaberipur, G.; Kaivani, A, "Binary-coded decimal digit multipliers", Computers & Digital Techniques, IET Volume 1, Issue 4, July 2007 pp. 377 – 381

[8] Schmookler, M.: 'High-speed binary-to-decimal conversion', IEEE Trans. Comput., 1968, 17, (5), pp. 506–508

[9] Rhyne, V.T.: 'Serial binary-to-decimal and decimal-to-binary conversion', IEEE Trans. Comput., 1970, 19, (9), pp. 808–812

[10] Arazi, B., and Naccache, D.: 'Binary-to-decimal conversion based on the 282 1 by 5', Electron. Lett., 1992, 28, (23), pp. 2151–2152

[11] M. Schmookler and A. Weinberger, "High Speed Decimal Addition," IEEE Trans. Computers, vol. 20, no. 8, pp. 862-867, Aug. 1971

[12] M.A. Erle and M.J. Schulte, "Decimal Multiplication via Carry-Save Addition," Proc. IEEE 14th Int'l Conf. Application-Specific Systems, Architectures, and Processors, pp. 348-358, June 2003.

[13] T. Ohtsuki et al., "Apparatus for Decimal Multiplication," US Patent #4,677,583, June 1987.

[14] B. Shirazi, D.Y. Yun, and C.N. Zhang, "RBCD: Redundant Binary Coded Decimal Adder," IEE Proc.—Part E, vol. 136, no. 2, Mar. 1989.

[15] B. Shirazi, D.Y. Yun, and C.N. Zhang, "VLSI Designs for Redundant Binary-Coded Decimal Addition," Proc. Seventh Ann. Int'l Conf. Computers and Comm., pp. 52-56, Mar. 1988.

[16] R. D. Kenney and M. J. Schulte, 'High-Speed Multi-operand Decimal Adders' IEEE Transactions on Computers, vol. 54, No. 8, Aug 2005, pp. 953-963

[17] L. Dadda, 'Multioperand Parallel Decimal Adder:A Mixed Binary and BCD Approach', IEEE Transactions on Computers, Vol. 56, No. 9, Sept 2007