A High-Frequency Decimal Multiplier

Robert D. Kenney and Michael J. Schulte
Department of ECE
University of Wisconsin – Madison
Madison, WI 53706, USA
rdkenney@uwalumni.com, schulte@engr.wisc.edu

Mark A. Erle
Systems and Technology Group
International Business Machines
Poughkeepsie, NY 12601
merle@us.ibm.com

Abstract

Decimal arithmetic is regaining popularity in the computing community due to the growing importance of commercial, financial, and Internet-based applications, which process decimal data. This paper presents an iterative decimal multiplier, which is operates at high clock frequencies and scales well to large operand sizes. The multiplier uses a new decimal representation for intermediate products, which allows for a very fast two-stage iterative multiplier design. Decimal multipliers, which are synthesized using a 0.11 micron CMOS standard cell library, operate at clock frequencies close to 2 GHz. The latency of the proposed design to multiply two n-digit BCD operands is (n + 8) cycles with a new multiplication able to begin every (n + 1) cycles.

1. Introduction

Recently, support for decimal arithmetic has received increased attention due to the growing importance of financial, commercial, and Internet-based applications, which often cannot tolerate errors from converting between decimal and binary formats. Since many decimal numbers, such as 0.2, cannot be exactly represented in binary, these applications often store data in decimal format and process data using decimal arithmetic software [1]. Although decimal arithmetic software eliminates conversion errors, it is typically 100 to 1,000 times slower than binary arithmetic implemented in hardware [1]. Due to the growing importance of decimal arithmetic, specifications for it have recently been added to the draft revision of the IEEE 754 Standard for Floating-Point Arithmetic [2].

This paper presents an iterative decimal multiplier, which operates at high clock frequencies and scales well to large operand sizes. The multiplier uses a novel decimal representation for intermediate products, which allows for a very fast two-stage iterative multiplier

design. Section 2 discusses previous approaches to decimal multiplication. Section 3 presents our proposed approach to decimal multiplication. Section 4 provides area and delay estimates for decimal multipliers with various operand sizes. Section 5 presents our conclusions. Additional information on decimal arithmetic is available from mesa.ece.wisc.edu and www2.hursley.ibm.com/decimal/.

In the remainder of this paper, decimal numbers are assumed to be in Binary Coded Decimal (BCD) format. Subscripts next to constants are used to denote the base of the constant. For example, 1001_2 represents the binary number equal to 9_{10}. An upper case variable (e.g., B) denotes an entire operand word. A lower case variable with an associated subscript (e.g. b_i) denotes a single digit in that operand. A digit referenced with brackets (e.g. $b_i[4]$) denotes a single bit in that digit. The multiplication of an *n*-digit multiplicand, *A*, by an n-digit multiplier, B, produces a *2n*-digit product, P.

2. Previous decimal multipliers

Decimal arithmetic units are inherently more complex than binary arithmetic units, since they need to handle a wider range of digits, carries across both bit and digit boundaries, and invalid result digits. For example, adding two BCD numbers using binary addition can produce invalid BCD digits in the range A_{16} - F_{16}. When this happens, each invalid BCD digit is corrected by adding six, which produces a valid BCD digit in the range 0_{10} - 5_{10} and a digit carry. Because of their increased complexity, decimal multipliers are typically implemented using an iterative approach. Each iteration, the entire multiplicand is multiplied by one multiplier digit to generate a partial product. The partial product is added to an intermediate product register that holds the previously accumulated partial products.

An iterative decimal multiplier is presented in [3]. In this multiplier, decimal partial products are generated by creating two partial products, *DH* and *DL*, for each multiplier digit. When the entire multiplicand is multiplied by a single multiplier digit, *DH* contains the higher order digits and *DL* contains the lower order digits. For example, the partial products generated with a BCD multiplicand of 3548_{10} and a multiplier digit of 6_{10} are:

$$
\begin{array}{r}
3548 \\
\underline{\times 6} \\
08048 \rightarrow \text{DL} \\
13240 \rightarrow \text{DH}
\end{array}
$$

During a single iteration, a first round of decimal corrections is performed on the intermediate product, which is stored in carry-save format. These new values, along with *DH*, are sent into a binary carry-save adder and the resulting sum is corrected. The new sum, new carry, and *DL* are then added using binary carry-save addition and the resulting sum is corrected yet again. Finally, this sum and carry are right-shifted and stored in intermediate registers, to be added to the new *DH* and *DL* generated in the next cycle. Multiplying two *n*-digit BCD numbers requires *n* iterations, where each iteration consists 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.

An alternate approach to iterative decimal multiplication, which uses decimal carry-save addition, is presented in [4]. The multiplier in [4] uses a unique form of decimal partial product generation. When generating the partial product, $A \times b_i$, the multiplier digit, b_i, is used to select values from two sets of secondary multiples. The sum of the selected secondary multiples equals the partial product $A \times b_i$. During the first cycle of operation, the secondary multiples *2A*, *4A*, and *5A* are generated using simple combinational logic. Since the multiplicand is latched and unaltered during operation, the secondary multiples need not be latched, which saves area. The secondary multiples are divided into two sets, *SM1* ∈ *{0A, 1A, 4A, 5A}* and *SM2* ∈ *{0A, 2A, 4A}*, so that any multiple of *A*, from *0A* to *9A*, can be produced by adding appropriate values of *SM1* and *SM2*. For example, if $b_i = 6$, *4A* and *2A* are selected for *SM1* and *SM2*, respectively.

The multiplier presented in [4] performs iterative additions in two pipeline stages, which allows for a higher clock frequency than the design proposed in [3]. The first stage uses a simplified decimal 3:2 counter to add the secondary multiples, *SM1* and *SM2*, to produce *n* 4-bit BCD sum digits and *n* 1-bit carry digits. The second stage uses a decimal 4:2 compressor to add the sum and carry digits from the previous stage, along with the carry and sum digits from the previous 4:2 compression. The output of the decimal 4:2 compressor corresponds to the intermediate product in decimal carry-save format. The decimal 3:2 counter and 4:2 compressor use a variant of *direct decimal addition*, introduced in [5]. The critical path for this multiplier consists of ten levels of logic to perform the decimal 4:2 compression, where each logic level corresponds to one complex gate delay.

After *n* iterations, the 4-bit sum and 1-bit carry digits are added using a simplified form of decimal carry-propagate addition. The latency of this multiplier is $(n + 4)$ cycles and a new multiplication can begin every $(n + 1)$ cycles.

3. Proposed Decimal Multiplier

The multiplier presented in [4] stores intermediate product digits in a BCD carry-save format. Our multiplier stores these digits in a less restrictive, redundant format, called the *overloaded decimal representation*, which reduces the delay of the iterative portion of the multiplier. In a standard BCD representation, the bit combinations A_{16} - F_{16} correspond to invalid BCD digits. Our 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 10_{10}. This allows decimal numbers to have multiple representations. For example, the number 120_{10} can be represented as 120_{10} or $0C0_{10}$. The overload decimal representation reduces the overhead of correcting sum digits during the iterative portion of the multiplier, since sum correction is only performed when sixteen is exceeded. When the final product 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}.

Figure 1 is a diagram of our proposed multiplier design. The secondary multiple generation and selection, and decimal carry-propagate addition are identical to those used in [4]. The overloaded decimal adder takes two cycles to add the secondary multiples to the intermediate product, which is stored using our overloaded decimal representation. When digits leave the overloaded decimal adder, clean-up logic is used to convert the overloaded decimal digits back to BCD digits. The intermediate product register is also cleaned-up before the final carry-propagate addition is performed. As digits enter the final product shift register and at the end of the multiplication, 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 major portions of the multiplier are explained below.

Figure 2 shows a block diagram for one digit of a two-stage overloaded decimal adder. The first stage inputs three 4-bit digits to a 4-bit binary carry-save adder (CSA). Two inputs, $sm1_i$ and $sm2_i$, are BCD digits from the secondary multiples, *SM1* and *SM2*, and the third digit, pr_{i+2}, is an overloaded decimal digit from the intermediate product register, *PR*. With BCD addition, six is added to the sum each time ten is surpassed. In our overloaded decimal representation, when a carry is generated out of a 4-bit digit, it corresponds to a value of sixteen. Thus, when a carry is generated, it is known that the sum of the digits added is at least sixteen, which is greater than ten, so a correction factor of six must be added. To reduce the worst case delay of the overloaded decimal adder, six is added in the next iteration. Since the secondary multiple digits are in BCD form (0 - 9), their digit values plus six (6 - F) can be found with simple two-level logic. In our adder, the carry-outs from additions in the previous iterations (labeled *co_top* and *co_bot* in Figure 2) select whether to add the secondary multiple digit or the secondary multiple digit plus six. In the first stage of the overloaded decimal adder, the two secondary multiple digits, $sm1_i$ and $sm2_i$, are conditionally increased by six and added, along with the intermediate product digit, pr_{i+2}, using a binary carry-save adder. In the second stage, the overloaded decimal sum and carry digits from the first stage are compressed using a 4-bit binary carry-propagate adder (CPA) to produce the new overloaded decimal product digit, pr_i.
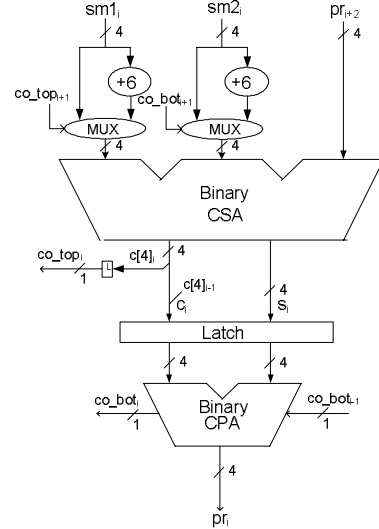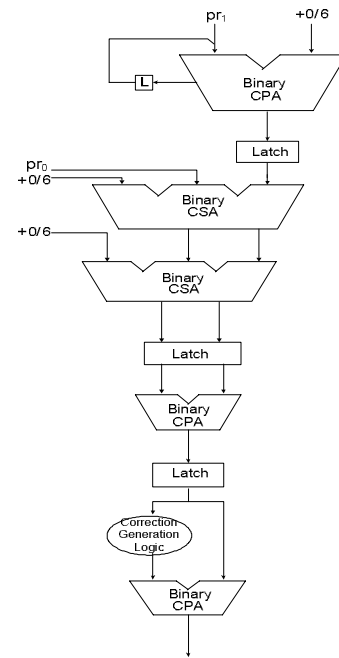

Figure 2. 1-digit, 2-stage overloaded decimal adder
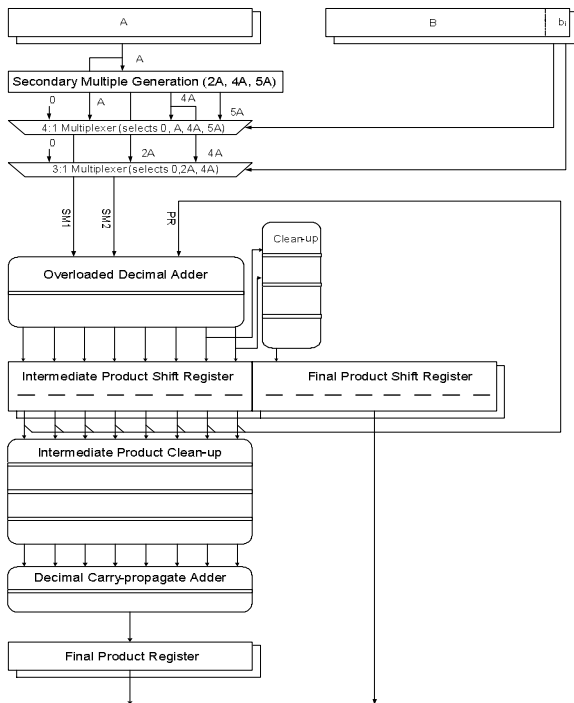

Figure 3. 1-digit, 4-stage clean-up block

Since the overloaded decimal adder uses two stages, it holds two intermediate products. While one is in the intermediate product shift register, the other is in the latch half-way through the overloaded decimal adder. In order to produce a new final product digit each cycle, the least significant intermediate product digits, pr_1 and pr_0, are summed and corrected into proper BCD digits, by adding six to each digit if it is in the range A_{16} - F_{16}. This is done in a 1-digit. 4-stage clean-up block, which is shown in Figure 3. The first stage adds pr_1 and a correction factor of +0/6. The second stage adds pr_0, the result of the first clean-up stage, and two more +0/6

Figure 1: Proposed decimal multiplier

correction factors using binary carry-save adders. The third stage adds the sum and carry from stage two, and the fourth stage corrects the result of stage three.

When the iterative phase of multiplication completes, the digits in the intermediate product shift register also need to be cleaned up. This is done by an intermediate product clean-up block, which consists of an array of n modified 1-digit clean-up blocks. The intermediate product clean-up block merges the two intermediate products for each digit, corrects invalid sum digits, and produces a 1-bit carry and a 4-bit BCD sum for each digit. The sum and carry digits are sent to a simplified decimal carry-propagate adder, which uses a two-stage pipeline and is identical to the one presented in [4].

4. Synthesis Results

The decimal multiplier design proposed in Section 3 is constructed in Verilog for 8, 16, and 34-digit wide operands. All multipliers are synthesized using Synopsys Design Compiler and LSI Logic's gflxp 0.11 micron CMOS standard cell library. Table 1 shows the results of synthesis runs when the designs are optimized for delay. The delays for each operand size are very close, since the delay of the design is not very dependent on the width of the operands. The small differences in delay can be attributed to increased fanout with increased operand size. The critical path for our multiplier is in the first stage of the overloaded decimal adder. The combinational delay for this stage corresponds to eight logic levels. It is composed of a 4-to-1 multiplexer delay to select SM1 and SM2, two-level logic to find the +6 values of SM1 and SM2, and a binary carry-save addition. Simplified decimal carry-propagate addition is performed in two cycles, so that it not on the critical delay path.

Table 1. Multiplier synthesis results

Operand Size (n)	Delay (ns)	Frequency (GHz)	Area (mm^2)
8	0.49	2.04	0.093
16	0.50	2.00	0.199
34	0.51	1.96	0.373

The multiplier discussed in Section 2 and [4] is also synthesized in the same environment as the multipliers we designed. Table 2 gives a comparison of the 34-digit multipliers for each technique when the multipliers are optimized for delay. Our design is able to operate at a 14% higher clock frequency than the multiplier in [4]. It requires 77% more area because of the clean-up blocks needed to merge and correct the two intermediate products. In [4], the latency for an n-digit

multiplication is $(n + 4)$ cycles and a new multiplication can begin every $(n + 1)$ cycles. The latency for our multiplier is $(n + 8)$ cycles and a new multiplication can begin every $(n + 1)$ cycles.

Table 2: Comparison of 34-digit multipliers

Multiplier Technique	Delay (ns)	Frequency (GHz)	Area (mm^2)
Decimal Carry-Save Addition [4]	0.58	1.72	0.210
Overloaded Decimal Addition	0.51	1.96	0.373

6. Conclusions

In this paper, we have given motivation for implementing decimal arithmetic in hardware. Two previous implementations of decimal multipliers are discussed. A decimal multiplier design that operates at high clock frequencies is proposed. The intermediate product is stored in an overloaded decimal representation, which allows the invalid BCD digits, A_{16} - F_{16}, to be used. Decimal multipliers are constructed in Verilog for 8, 16, and 34-digit operands. Synthesis results show that the circuits operate at clock frequencies in the vicinity of 2 GHz when implemented using a 0.11 micron CMOS standard cell library. A performance comparison with the decimal multiplier design presented in [9] shows that the proposed multiplier achieves a 14% higher clock frequency.

Acknowledgements

References

[1] M. F. Cowlishaw, "Decimal Floating-Point: Algorithm for Computers," *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pp. 104-111, June 2003.

[2] *Draft IEEE Standard for Floating-Point Arithmetic*, IEEE, inc., New York, 2003. Available from: http://754r.ucbtest.org/drafts/754r.pdf.

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

[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] M. Schmookler and A. Weinberger, "High Speed Decimal Addition," *IEEE Transactions on Computers*, Vol. C-20, No. 8, pp. 862-866, August 1971.