On High-Performance Parallel Fixed-Point Decimal Multiplier Designs

Ming Zhu
University of Nevada, Las Vegas, ming.z1989@gmail.com
ON HIGH- PERFORMANCE PARALLEL FIXED-POINT

DECIMAL MULTIPLIER DESIGNS

by

Ming Zhu

Bachelor of Engineering in Microelectronics

Shanghai Jiao Tong University

2011

A thesis submitted in partial fulfillment

of the requirements for the

Master of Science in Electrical Engineering – Electrical Engineering

Department of Electrical and Computer Engineering

Howard R. Hughes College of Engineering

The Graduate College

University of Nevada, Las Vegas

December 2013
THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Ming Zhu

entitled

On High-Performance Parallel Fixed-Point Decimal Multiplier Designs

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering
Department of Electrical and Computer Engineering

Yingtao Jiang, Ph.D., Committee Chair
Emma Regentova, Ph.D., Committee Member
Mei Yang, Ph.D., Committee Member
Hui Zhao, Ph.D., Graduate College Representative
Kathryn Hausbeck Korgan, Ph.D., Interim Dean of the Graduate College

December 2013
ABSTRACT

High-performance, area-efficient hardware implementation of decimal multiplication is preferred to slow software simulations in a number of key scientific and financial application areas, where errors caused by converting decimal numbers into their approximate binary representations are not acceptable.

Multi-digit parallel decimal multipliers involve two major stages: (i) the partial product generation (PPG) stage, where decimal partial products are determined by selecting the right versions of the pre-computed multiples of the multiplicand, followed by (ii) the partial product accumulation (PPA) stage, where all the partial products are shifted and then added together to obtain the final multiplication product. In this thesis, we propose a parallel architecture for fixed-point decimal multiplications based on the 8421-5421 BCD representation. In essence, we apply a hybrid 8421-5421 recoding scheme to help simplify the computation logic of the PPG. In the following PPA stage, these generated partial products are accumulated using 8421 carry-lookahead adders (CLAs) organized as a tree structure; this organization is a significant departure from the traditional carry-save-adder-based (CSA) approach, which suffers from the problems introduced by extra recoding logic and/or addition circuits needed. In addition to the proposed 8421-5421-based decimal multiplier, we also propose a 4221-based decimal multiplier that is built upon a novel full adder for 4221 BCD codes; in this design, expensive 4221-to-8421 conversions are no longer needed, and as a result, the operands of this 4221 multiplier can be directly represented in 4221 BCD.

The proposed 16×16 decimal multipliers are compared against other best known decimal multiplier designs in terms of delays and delay-area products with a TSMC
90nm technology. The evaluation results have confirmed that the proposed 8421-5421 multiplier achieves the lowest delay and is the most time-area efficient design among all the existing hardware-based BCD multipliers.
ACKNOWLEDGEMENT

I am grateful to my advisor, Dr. Yingtao Jiang, for his guidance during my 2-year study. He is like a mentor to me and offers me as many opportunities as he can. It was also him who gave me the idea of this research, encouraged me when the research did not go well and guided me with the paper writing. I also appreciate the advice and assistance from my thesis committee which includes Dr. Mei Yang, Dr. Emma E. Regentova and Dr. Hui Zhao.

Last but not least, I would like to thank my beloved family and friends for their support all along the way.
# TABLE OF CONTENTS

ABSTRACT............................................................................................................................ ii

ACKNOWLEDGEMENT .......................................................................................................... v

TABLE OF CONTENTS......................................................................................................... vi

LIST OF TABLES .................................................................................................................. vii

LIST OF FIGURES ................................................................................................................. viii

CHAPTER 1 DEFINITIONS AND ABBREVIATIONS .............................................................. 1

CHAPTER 2 INTRODUCTION ............................................................................................... 3

CHAPTER 3 LITERATURE OVERVIEW .............................................................................. 8

A. Decimal PPG .................................................................................................................. 8
   a. Decoding Algorithm for Multiplier Digit .................................................................. 8
   b. Partial Product Pre-Computations .......................................................................... 9

B. Decimal Addition and PPA ......................................................................................... 10

CHAPTER 4 PROPOSED 8421 BCD MULTIPLIER ............................................................ 12

A. 8421-5421 BCD PPG ................................................................................................... 12

B. 8421 BCD Addition and Partial Product Accumulation ............................................. 15

CHAPTER 5 PROPOSED 4221 MULTIPLIER ..................................................................... 21

A. 4221 BCD PPG ............................................................................................................. 21

B. 4221 BCD Addition and PPA ..................................................................................... 23

CHAPTER 6 EVALUATION AND COMPARISON ............................................................ 27

A. Decimal PPG ............................................................................................................... 27

B. Decimal Addition ....................................................................................................... 28

C. Decimal PPA ............................................................................................................... 30

D. Decimal Multiplication ............................................................................................ 30

CHAPTER 7 CONCLUSION ............................................................................................... 32

A. Summery ..................................................................................................................... 32

B. Future Work ................................................................................................................ 32

APPENDIX (VERILOG HDL CODES) ............................................................................. 33

REFERENCE ....................................................................................................................... 70

CV ....................................................................................................................................... 73
LIST OF TABLES

TABLE 1. BCD REPRESENTATIONS ................................................................. 2

TABLE 2. DECODING OF $bi$ ........................................................................ 8

TABLE 3. 16-DIGIT $2An$ AND $5An$ COMPARISON .................................. 27

TABLE 4. 16-DIGIT PPG COMPARISON .................................................... 28

TABLE 5. ADDER COMPARISON .................................................................. 29

TABLE 6. PPR AND PPA FOR 16-BY-16 DECIMAL MULTIPLICATION .......... 29

TABLE 7. 16-BY-16 DECIMAL MULTIPLICATIONS ........................................ 29

TABLE 8. AREA-DELAY FOR 16-BCD MULTIPLIERS ................................... 31
LIST OF FIGURES

Figure 1. A general structure for an $n$-by-$n$ BCD multiplier ................................................. 4
Figure 2. A pencil-and-paper approach for a 4-by-1 decimal PPG $An \times b$ ......................... 4
Figure 3. Optimized “Radix-5” PPG structure ........................................................................ 13
Figure 4. Partial product accumulation of an $2n$-by-$2n$ decimal multiplication ............. 16
Figure 5. Accumulation of partial products for 16-by-16 decimal multiplication .......... 20
Figure 6. 32 parallel 4221 32:2 CSA Trees for PPR ................................................................. 26
Figure 7. 32:2 4221 CSA Tree for PPR .................................................................................. 26
CHAPTER 1
DEFINITIONS AND ABBREVIATIONS

1) number representation: A binary number \( a \) is formally expressed as \((a)_2\), and a decimal number \( b \) can be expressed as \((b)_{10}\).

2) bit: Each bit has a value of either 0 or 1.

3) digit: Each Arabic number is a decimal digit, with a value ranging from 0 to 9. Unless explicitly stated otherwise, an \( m \)-by-\( n \) multiplication means an \( m \)-digit decimal multiplicand multiplies with an \( n \)-digit decimal multiplier.

4) MSB/LSB: Most/Least Significant Bit.

5) MSD/LSD: Most/Least Significant Digit.

6) BCD: Binary-Coded-Decimal. In this paper, we focus on four sets of BCD codes, namely, 8421, 5421, 4221, and 5211; in BCD, 4 bits is used to represent 1 digit. Unless explicitly stated otherwise, 8421 means 8421-BCD representation. An \( n \)-digit BCD \( A \) is given as:

\[
A_n = A_n[4n-1:0] = a_{n-1} \ldots a_0,
\]

where \( n \) is the digit length, and the total bit length is \( 4n \). The \( i \)-th digit \( a_i = a_i[3:0], i = 0, \ldots, n-1 \) can assume an integer value between 0 to 9, with a weight of \( 10^i \). The value of one digit \( (a_i) \) is determined as:

a) 8421: \( 8a_i[3] + 4a_i[2] + 2a_i[1] + a_i[0] \);

b) 5421: \( 5a_i[3] + 4a_i[2] + 2a_i[1] + a_i[0] \);

c) 4221: \( 4a_i[3] + 2a_i[2] + 2a_i[1] + a_i[0] \);

d) 5211: \( 5a_i[3] + 2a_i[2] + a_i[1] + a_i[0] \);
TABLE 1 tabulates all the valid representations of a digit for all four BCD coding schemes. Since one digit is mapped to a 4-bit representation, which can represent up to 16 values, the obvious redundancy complicates the decimal computations in binary systems.

TABLE 1. BCD REPRESENTATIONS

<table>
<thead>
<tr>
<th>Decimal Value</th>
<th>8421-BCD</th>
<th>5421-BCD</th>
<th>4221-BCD</th>
<th>5211-BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>0010</td>
<td>0010 0100</td>
<td>0011 0100</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>0011</td>
<td>0011 0101</td>
<td>0101 0110</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>0100</td>
<td>0110 1000</td>
<td>0111</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>1000</td>
<td>1001 0111</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>1001</td>
<td>1100 1010</td>
<td>1010 1101</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>1010</td>
<td>1101 1011</td>
<td>1100 1011</td>
</tr>
<tr>
<td>8</td>
<td>1000</td>
<td>1011</td>
<td>1110</td>
<td>1110 1101</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>1100</td>
<td>1111</td>
<td>1111</td>
</tr>
</tbody>
</table>
CHAPTER 2
INTRODUCTION

Decimal-based computer arithmetic has been around since the ENIAC era [1], but it was quickly replaced by binary arithmetic for obvious reasons: (i) representing the 10 decimal numbers but with a four-bit BCD is much less efficient than representing 16 binary numbers with the same four bits, especially when logic gates were such a precious hardware resource in early computers; (ii) decimal arithmetic operations are typically much more complex and slower than their counterpart binary arithmetic operations due to wider value range of each digit (from 0 to 9) and the representation redundancy. Even so, support for decimal arithmetic is still appreciated in niche financial and many other key applications, where errors caused by converting a decimal number to its approximate binary representation (e.g., the decimal number of 0.2 cannot be exactly represented in binary) are often unacceptable. As a result, data in these applications are still in BCD representations and they are processed using a software approach for decimal arithmetic computation, a process that is typically 100 to 1,000 times slower than hardware-based binary computations [2].

Thanks to the rapid advancement of VLSI technology, and partially driven by the recent release of the IEEE 754 Standard [3] with newly added specifications governing decimal computations, hardware-based implementations for decimal computations have enjoyed revived interest.

In general, an n-by-n BCD multiplication $A_n \times B_n$ can be performed as:

$$P_{2n} = A_n \times B_n = A_n \sum_{i=0}^{n-1} b_i(10)^i = \sum_{i=0}^{n-1} A_n b_i(10)^i = \sum_{i=0}^{n-1} PPi_{n+1}(10)^i,$$

(1)
Figure 1. A general structure for an $n$-by-$n$ BCD multiplier

$$
\begin{array}{c}
\times \\
\downarrow \\
\hline
a_3 \quad a_2 \quad a_1 \quad a_0 \\
\downarrow \\
\hline
pt_3 \quad pt_2 \quad pt_1 \quad pt_0 \\
\downarrow \\
\hline
+ \quad c_3 \quad c_2 \quad c_1 \quad c_0 \\
\downarrow \\
\hline
pp_1 \quad pp_3 \quad pp_2 \quad pp_1 \quad pp_0
\end{array}
$$

Figure 2. A pencil-and-paper approach for a 4-by-1 decimal PPG $A_n \times b$

where $PPi_{n+1} = A_n \times b_i$ is the $i$-th partial product, given by the multiplicand $A_n$ and $b_i$, the $i$-th digit of the multiplier, $B_n$.

A general BCD multiplier architecture is depicted in Figure 1, which consists of (i) the partial product generation (PPG) stage, including a decoder of $b_i$ to select the right multiples of the multiplicand as determined by the pre-computation logic, and (ii) the partial product accumulation (PPA) stage, where all the partial products are shifted and added together to obtain the final multiplication product.

A traditional pencil-and-paper approach for the decimal PPG (Figure 1) is depicted in Figure 2, where $a_i \times b = 10 \times c_i + pt_i$. Here, the single-digit multiplication results (i.e. $c_i$ and $pt_i$) are determined by searching a Look-Up Table (LUT), which
stores all the combinations of pre-computed single-digit multiplication results and is indexed by the multiplier digit \( b \); then the partial product is obtained by summing up all the \( c_i \)'s and \( pt_i \)'s. Since each digit has 10 values, such LUT has up to \( 10 \times 10 \) entries, or 55 entries if duplicate results due to multiplication’s commutative property \((a \times b = b \times a)\) are removed. The LUT of such a size has a negative implication on circuit size and delay [4]. As an alternative approach, decimal PPG can be performed as binary multiplications first, followed by a conversion step where the binary partial products are transformed back into their corresponding decimal representations [5-9]. One big problem associated with this approach is that the logic of long-digit binary-to-decimal conversions can be extremely complex.

When it comes to PPG (Figure 1) optimization, most existing works follow a similar idea by writing a multiplier digit as a summation of two decimal numbers, namely \( b_i = b0_i + b1_i \). By doing so, two intermediate partial products (e.g. \( A_n, 2A_n \) and \( 5A_n \)) that can be obtained through very simple logic operations like shifts are combined to generate the desired partial product (e.g. \( 7A_n = 2A_n + 5A_n \)) [4, 10-24] that otherwise is hard to compute directly. This idea can be better described in the following equation:

\[
P_{2n} = A_n \times B_n = \sum_{i=0}^{n-1} A_n b_i(10)^i
\]

\[
= \sum_{i=0}^{n-1} A_n(b0_i + b1_i)(10)^i
\]

\[
= \sum_{i=0}^{n-1} (A_n(b0_i) + A_n(b1_i))(10)^i
\]

\[
= \sum_{i=0}^{n-1} (PP0i_{n+1} + PP1i_{n+1})(10)^i.
\]
As $PPi_{n+1} = PP0i_{n+1} + PP1i_{n+1}$, $PP0i_{n+1}$ and $PP1i_{n+1}$ are hereinafter referred as the two intermediate partial products of $PPi_{n+1}$.

As a step after PPG, PPAs can be performed iteratively [10] [11], sequentially [25] or in parallel [15] [16]. Iterative architectures achieve high hardware utilization, but at a cost of high latency and low data throughput. Carry-save adders (CSAs) are generally used in parallel partial product reductions (PPRs) as an effective mean to reduce the delay induced by carry propagation, but this approach requires extra logic and/or additions to attain the final products [15]. All these techniques can be supplemented by adding pipelines to further improve the data throughput and the hardware utilization [15].

Although the operands and partial products in decimal multipliers (Figure 1) are typically represented in popular 8421 BCD, actually, there are a number of alternative data representations, i.e. 4221, 5211 and 5421 BCD [17] [26], that can also be employed, alone or along with 8421 codes, to simplify the PPG logic and achieve the partial products in 4221, which can be accumulated by using simple 4221 CSAs in the PPR. However, unsuitable mixing of BCDs, such as 8421-4221, could still cause excessive delay and/or hardware overhead because the PPR structures for 4221 partial products incur long carry propagation and an extra 4221-8421 conversion is required after the PPR to achieve the final 8421 product.

In this paper, we implement three 16-by-16 multipliers based on different combinations of 8421 and 4221 BCD codes. We propose a hybrid 8421-5421 recoding multiplier with simplified partial product pre-computation logic for PPG, and 8421 carry-lookahead adders (CLAs) organized as a parallel tree structure for PPA. We also design a 4221 BCD multiplier that has the same architecture as the proposed 8421 multiplier; this
4221 multiplier explores modified 4221-5211 recoding for partial product pre-computations in PPG and it uses novel 4221 CLAs for additions in PPA. The third design is another 4221 BCD multiplier that has the same PPG design as the previous 4221 BCD multiplier, but for the PPA stage, it uses 4221 CSAs for PPR and a 32-digit 4221 CLA to achieve the final product represented in 4221. No 4221-8421 conversions are required in either proposed 4221 multipliers. We synthesize all three designs, and compare them against the best known architectures in terms of delay and delay-area product. Reports confirm that our 8421-5421 multiplier achieves a significant speed-up and hardware overhead reduction, and outperforms all the existing BCD multipliers in both terms of delay and area-time efficiency.

In what follows, we will review previous work in Chapter 2. The proposed 8421 and 4221 BCD multipliers are detailed in Chapter 3 and 4, respectively. Performance results of various BCD multiplier designs are reported and analyzed in Chapter 5. Finally, the conclusion is drawn in Chapter 6.
CHAPTER 3

LITERATURE OVERVIEW

In this chapter, various design techniques, which are applicable to PPGs and PPAs in decimal multipliers (Figure 1) with different BCD representations, will be reviewed.

A. Decimal PPG

As alluded earlier, PPG contains several steps: the decoding of \( b_i \), the pre-computations of multiples of multiplicands, and the summation of the two intermediate partial products. Various techniques have been proposed for these steps.

a. Decoding Algorithm for Multiplier Digit

TABLE 2 summarizes various combinations of \( b_{0i} \) and \( b_{1i} \) to make up for \( b_i \) as in (2). In [10] [11], \( b_{0i} \) and \( b_{1i} \) take values from the set of \{ 0, 1, 2, 4, 5 \}, while in [4], they draw values from \{ 0, 1, 2, 5 \} to compute \( b_i \) ranging from 0 to 7, and the remaining 8\( A_n \) and 9\( A_n \) cases are obtained directly using the pencil-and-paper approach as shown in Figure 2. In the approaches adopted in [4] [10] [11], as both two intermediate partial products are positive, the partial product out of these two intermediate partial products is obtained through a simple addition.

<table>
<thead>
<tr>
<th>( b_i )</th>
<th>( b_{0i} + b_{1i} )</th>
<th>( b_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 + 0</td>
<td>0 + 0</td>
</tr>
<tr>
<td>1</td>
<td>1 + 0</td>
<td>1 + 0</td>
</tr>
<tr>
<td>2</td>
<td>2 + 0</td>
<td>2 + 0</td>
</tr>
<tr>
<td>3</td>
<td>1 + 2</td>
<td>1 + 2</td>
</tr>
<tr>
<td>4</td>
<td>4 + 0</td>
<td>2 + 2</td>
</tr>
<tr>
<td>5</td>
<td>0 + 5</td>
<td>0 + 5</td>
</tr>
<tr>
<td>6</td>
<td>1 + 5</td>
<td>1 + 5</td>
</tr>
<tr>
<td>7</td>
<td>1 + 5</td>
<td>2 + 5</td>
</tr>
<tr>
<td>8</td>
<td>4 + 4</td>
<td>Figure 2, ( b_i = 8 )</td>
</tr>
<tr>
<td>9</td>
<td>4 + 5</td>
<td>Figure 2, ( b_i = 9 )</td>
</tr>
</tbody>
</table>
The algorithm adopted in [15] is quite different from those in [4] [10] [11], where subtractions may be needed to combine the two intermediate products. In this case, each 8421 multiplier digit is computed as \( b_i = b_{Hi} \times 5 + b_{Li} \), where \( b_{Hi} \in \{0, 1, 2\} \) and \( b_{Li} \in \{\pm 2, \pm 1, 0\} \); as so, this is a “Radix-5” approach. In a simple word, this approach selects the first intermediate partial product from the set of \{10A_n, 5A_n, 0\}, while the second intermediate partial product is taken from \{0, \pm 1A_n, \pm 2A_n\}. 10A_n is achieved by simply left-shifting \( A_n \) by one digit, while \(-A_n\) and \(-2A_n\) are readily calculated through the 10’s complements of \( A_n \) and \( 2A_n \), respectively, by adding the 9’s complement and 1 [27]. A “Radix-4” algorithm [17] [26] is similar to “Radix-5”, but this time, \( b_i = b_{Hi} \times 4 + b_{Li} \), and a longer delay is observed when pre-computing \( 8A_n \), which is achieved by chaining three \( 2A_n \) modules.

The “Radix-10” decoding algorithm adopted in [17] [26] decodes \( b_i \) from an integer interval of \[6, 9\] to \[-4, 0\] and adds 1 (a decoding carry) to the next multiplier digit with a higher weight. Thus, \( b_i \) falls into the range of \[-4, 5\], and only \( 0A_n \sim 5A_n \) are needed to be computed, which are much easier to implement than \( 6A_n \sim 9A_n \). However, due to the decoding carry propagation, “Radix-10” introduces high decoding delay, and yet, it still requires the summation of two intermediate partial products to get one partial product, in the case of \( 3A_n = A_n + 2A_n \).

b. Partial Product Pre-Computations

Conventionally, the operands of the PPG (i.e. multiplicands, multipliers and generated partial products) are all in 8421 BCD representations. Based on the observation that the carries in \( 2A_n \) and \( 5A_n \) do not propagate more than one digit, [10] gives the logic for 8421 BCD operands for the generation of the \( n \)-th digit of the \( i \)-th partial product,
reproduced here as (3) and (4), respectively. This motivates one to use $2A_n$ and $5A_n$ to
derive the intermediate partial products.

$$ppi_n[0] = a_{n-1}[2]a_{n-1}[1]a_{n-1}[0] + a_{n-1}[3]a_{n-1}[0] + a_{n-1}[3]$$

$$ppi_n[1] = a_{n}[3]a_{n}[2]a_{n}[1]a_{n}[0] + a_{n}[2]a_{n}[1]a_{n}[0] + a_{n}[3]a_{n}[0]$$

$$ppi_n[2] = a_{n}[1]a_{n}[0] + a_{n}[2]a_{n}[1] + a_{n}[3]a_{n}[0]$$

$$ppi_n[3] = a_{n}[2]a_{n}[1]a_{n}[0] + a_{n}[3]a_{n}[0]$$

$$ppi_n[0] = a_{n}[0]a_{n-1}[3]a_{n-1}[1] + a_{n}[0]a_{n-1}[1] + a_{n}[0]a_{n-1}[3]$$

$$ppi_n[1] = a_{n}[0]a_{n-1}[2] + a_{n}[0]a_{n-1}[2]a_{n-1}[1] + a_{n-1}[2]a_{n-1}[1] + a_{n}[0]a_{n-1}[3]$$

$$ppi_n[2] = a_{n}[2]a_{n-1}[3]a_{n-1}[1] + a_{n}[0]a_{n-1}[2]a_{n-1}[1] + a_{n}[0]a_{n-1}[3]$$

$$ppi_n[3] = a_{n}[0]a_{n-1}[2]a_{n-1}[1] + a_{n}[0]a_{n-1}[3]$$

A different approach is adopted in [17] [23] [26], where the multiplicands and
multipliers are coded in 8421, while the generated partial products are coded in 4221
(Table 1). $2A_n$ can be achieved by first applying 8421-to-5211 recoding and left-shifting
the 5211-encoded $A_n$ by 1 bit to become $2A_n$ in 4221; and $5A_n$ by first left-shifting $A_n$
coded in 8421 by 3 bits into 5421-encoded $5A_n$ and then recoding this 5421-encoded $5A_n$
into its 4221 representation [23] [26]. There are two compelling reasons that make 4221 a
preferred choice to represent the partial products: (i) the 9’s complements of 4221 can be
obtained by simply inverting each bit of the digits, and (ii) 4221 CSAs for the subsequent
PPR are simpler than their 8421 counterparts.

B. Decimal Addition and PPA

[27] proposes a combinational logic for a single-digit 8421 full adder, and this
design has been admitted into almost every 8421 BCD arithmetic operation. In [27], it
adds 2 adjacent digits as a byte sequentially, and generates a carry-generation signal \( g_i = 1 \), if \( a_{2i+1}a_{2i} + b_{2i+1}b_{2i} \geq 100 \) and a carry-propagation signal \( p_i = 1 \), if \( a_{2i+1}a_{2i} + b_{2i+1}b_{2i} = 99 \) for each byte. For multi-digit adders, hierarchical group carry-generations and group carry-propagations can be generated to help reduce the critical path delay pertaining to the carry propagation [27] [28] [29].

The summation of 8421 partial products can be performed iteratively [10] [11], sequentially [25], or in parallel [13] [15]. Iteration architectures achieve high hardware utilization, but at a cost of high data latency and low data throughput. A pipeline is often added in sequential accumulation to improve data throughput. Delay could be much reduced in parallel computations, where CSAs are generally used. [15] applies 8421 CSAs for the PPR, but this scheme requires extra logic, such as 9:4 compressors and extra addition stages, to take care of the redundant carries that could not be absorbed in the PPR, leading to larger delay as well as higher hardware overhead.

For 4221 PPAs, [17] [26] reduce the number of 4221-coded partial products through 4221 CSAs. In the end, since no 4221 full adders or CLAs have been considered, the results of 4221 PPR have to be recoded back to 8421 so that a classical 8421 CLA can be employed to obtain the final 8421-coded product [17] [26]. However, as there remain carry propagation problems in CSA trees and an extra 4221-to-8421 conversion is required, the delay of the multiplication tends to be increased considerably.
CHAPTER 4

PROPOSED 8421 BCD MULTIPLIER

The proposed 8421-5421 multiplier follows the “Radix-5 algorithm”, but with a different structure from what was employed in [15]. In this proposed multiplier, the $2A_n$ and $5A_n$ are instead computed based on 8421-5421 conversion. Rather than using CSA trees as seen in most exiting multipliers, in this proposed design, the obtained partial products are accumulated through 8421 CLAs, which are organized as a truly parallel tree without intermediate carries.

A. 8421-5421 BCD PPG

The proposed 8421-5421 PPG structure is presented in Figure 3, where $b_i = b_{Hi} \times 5 + (-1)^{OP}b_{Li}$. We generate the first intermediate partial product $PP0i_{n+1}$ from $\{10A_n, 5A_n, 0A_n\}$ in Pre_Comp0, while for the second intermediate partial product, instead of selecting from $\{0A, \pm1A_n, \pm2A_n\}$ directly (i.e. 1 out of 5) [15], we select $PP1i_{n+1}$ from $\{0A_n, A_n, 2A_n\}$ in Pre_Comp1 first (i.e. 1 out of 3, simpler than that in [15]) and then determine the second intermediate partial product, $PP2i_{n+1}$, in CMP with the operation signal $OP$. In CMP, if $OP = 1$, $PP2i_{n+1}$ is equal to the 9’s complement of $PP1i_{n+1}$; otherwise, $PP2i_{n+1} = PP1i_{n+1}$. Once $PP0i_{n+1}$ and $PP2i_{n+1}$ are obtained, a CLA adds them up together, along with the carry-in $OP$, to generate $P Pi_{n+1}$. This proposed PPG requires one 9’s complement module fewer than that in [15].

Since Pre-Comp0 only deals with $10A_n$ (i.e. $b0_i = (1010)_2$), $5A_n$ (i.e. $b0_i = (0101)_2$) and 0 (i.e. $b0_i = (0000)_2$), the two least significant bits of $b0_i$ is sufficient for the determination of the corresponding partial product; that is,
Figure 3. Optimized “Radix-5” PPG structure

- if $b_0[i:0] = 10, PP0i_{n+1} = 10A_n$;
- if $b_0[i:0] = 01, PP0i_{n+1} = 5A_n$;
- If $b_0[i:0] = 00, PP0i_{n+1} = 0$;

and a similar strategy could also be applied to Pre-Comp1, which deals with $\{0A_n, A_n, 2A_n\}$, i.e.

- if $b1[i:0] = 10, PP1i_{n+1} = 2A_n$;
- if $b1[i:0] = 01, PP1i_{n+1} = A_n$;
- If $b1[i:0] = 00, PP1i_{n+1} = 0$.

The Boolean expressions for $b0_i$, $b1_i$ and $OP$ are listed in (5) ~ (7), respectively.
The pre-computation of $10A_n$ is obtained by left-shifting $A_n$ by 1 digit, $1A_n$ is the multiplicand itself, and $0A_n$ has a result of 0. $2A_n$ and $5A_n$ can be obtained by recoding from 8421 to 5421 as in (8), and from 5421 to 8421 as in (9), respectively [23]. More specifically, recoding $A_n$ from 8421 to 5421 codes and then left-shifting the 5421-encoded $A_n$ by 1 bit gives the 8421-encoded $2A_n$. In an opposite manner, if $A_n$ in 8421 is left shifted by 3 bits, it becomes $5A_n$ in 5421; recode the 5421-encoded $5A_n$ back to 8421 and we get the $5A_n$ in 8421. As one can see, (8) and (9) have slightly simpler logic than those used in [10], and shifting needs no extra logic gate.

\[ b_0[i][0] = b_i[2] + b_i[1]b_i[0] \]
\[ b_0[i][1] = b_i[3] \quad (5) \]
\[ b_1[i][0] = b_i[2]b_i[0] + b_i[2]b_i[1]b_i[0] \]
\[ b_1[i][1] = b_i[3]b_i[0] + b_i[1]b_i[0] + b_i[2]b_i[1] \quad (6) \]
\[ (7) \]


\[ (8) \]

\[ (9) \]
To summarize, although this proposed 8421-5421 PPG requires one extra CMP module and a full adder, as opposed to the PPGs in [10] [11] [13] that only require half adders and no CMP modules, each pre-computation module in our 8421-5421 PPG selects the intermediate partial product from the three possible cases, while each pre-computation unit in [10] [11] has to select one out of the four possible cases and in [13], it draws the result out of five. Due to the architectural and logic simplicity of the pre-computations, this proposed 8421-5421 PPG is more area-time efficient than the ones reported in the literature.

B. 8421 BCD Addition and Partial Product Accumulation

In light of additions and subtractions in [27], we utilize a similar logic to compute the sum digit and the carry-out bit; but this time, instead of using the byte structure in [27], we generate a carry-generation bit and a carry-propagation bit for each single-digit adder, so that we can break the carry propagation chain inside the byte structure for higher performance. The logic of carry-generation and the carry-propagation are given as:

\[
\text{carry-generation: } g_{digit} = K + L G_0, \\
\text{carry-propagation: } p_{digit} = L H_0,
\]

where

\[
K = \{1 \mid a_i[3:1], 0 + b_i[3:1], 0 \geq 10\},
\]

\[
L = \{1 \mid a_i[3:1], 0 + b_i[3:1], 0 \geq 8\},
\]

\[
G_0 = a_i[0] \& b_i[0](\text{AND}), \quad H_0 = a_i[0] \oplus b_i[0](\text{XOR}).
\]

For multi-digit addition, following the approach adopted in [27] [28], we group 4 single-digit full adders together, where a group carry-generation and a group carry-propagation can be generated.
Figure 4. Partial product accumulation of an $2^n$-by-$2^n$ decimal multiplication
In general, to implement a high speed $2^n$-by-$2^n$ multiplication, we generate all $2^n$ partial products in parallel, and accumulate them using aforementioned 8421 CLAs organized as a tree structure with $n$ stages (Figure 4), which are indexed as Stage 1, Stage 2, ..., Stage $n$ from the top to the bottom. In Stage $i$ $(1 \leq i \leq n)$, we use a $(2^n + 2^i)$-digit adder($k$) to add the two sum results from the adder($2k$) and adder($2k+1$) in Stage $(i-1)$, i.e. $TPP(i-1)(2k)_{2^n+2^{i-1}}$ and $TPP(i-1)(2k+1)_{2^n+2^{i-1}}$, and generate $TPP(i)(k)_{2^n+2^i}$ ($TPP(0)(k)_{2^n+1}$ is essentially $PP(k)_{2^n+1}$, and $TPP(n)(0)_{2^n+2^n}$ is the final product of the $2^n$-by-$2^n$ multiplication).

**Lemma 1:** The maximum effective digit length of a product $P_x = A_n \times B_m$ is $m+n$, or equivalently, $P_x < (10)^{m+n}$.

**Theorem:** In Stage $i$ $(1 \leq i \leq n)$, adder($k$) $(0 \leq k \leq 2^{n-k} - 1)$ is a $(2^n + 2^i)$-digit adder, where only a $(2^n + 2^{i-1})$-digit CLA is required, and the carry-out is always 0; the output of adder($k+1$) $(0 \leq k \leq 2^{n-k} - 2)$, i.e. $TPP(i)(k+1)_{2^n+2^i}$, is $2^i$-digit left-shifted with respect to the output of adder($k$), i.e. $TPP(i)(k)_{2^n+2^i}$

**Proof (by mathematical induction):**

**Step 1** ($i = 1$, initial step):

The adder($k$) completes the addition of

$$PP(2k+1)_{2^n+1} \times 10 + PP(2k)_{2^n+1}$$

$$= A_{2^n}b_{2k+1} \times 10 + A_{2^n}b_{2k}$$

$$= A_{2^n} \sum_{j=0}^{1} b_{2k+j}(10)^j$$

(10)
Since $PP(2k+1)_{2^n+1}$ is 1-digit left shifted with respect to $PP(2k)_{2^n+1}$, adder(k) only needs a $(2^n + 1)$-digit CLA to complete the addition of $PP(2k + 1)_{2^n+1}$ and the $2^n$ MSDs of $PP(2k)_{2^n+1}$ (i.e. $0, PP(2k)_{2^n+1}[(4 \times 2^n + 3): 4]$), and set the LSD of $PP(2k)_{2^n+1}$ (i.e. $PP(2k)_{2^n+1}[3:0]$) as the LSD of the sum output of adder(k) directly. As a result, the sum output of adder(k) grows to $(2^n + 2)$ digits. Besides, since the sum output of the adder has the same number of digits as that of the $2^n$-by-2 multiplication (10), which has a maximum length of $(2^n + 2)$ digits (Lemma 1), the carry-out of adder(k) in Stage 1 is 0.

In Stage 1, the output of adder(k+1) is 2-digit left shifted with respect to the output of adder(k), because the inputs of adder(k+1) are $PP(2k + 3)_{2^n+1}$ and $PP(2k + 2)_{2^n+1}$, which are 2-digit left shifted with respect to the inputs of adder(k), i.e. $PP(2k + 1)_{2^n+1}$ and $PP(2k)_{2^n+1}$, respectively.

**Step 2** (inductive step):

We assume that in Stage $i$ ($1 \leq i \leq n - 1$), the proposition holds. Then in Stage $(i+1)$, since the input of adder(k) $TPP(i)(2k + 1)_{2^n+2^i}$ is $2 \times 2^{i-1} = 2^i$ digits left-shifted with respect to the other input $TPP(i)(2k)_{2^n+2^i}$, adder(k) only needs a $(2^n + 2^i)$-digit CLA to add $TPP(i)(2k + 1)_{2^n+2^i}$ and the $2^n$ MSDs of $TPP(i)(2k)_{2^n+2^i}$, and leaves the $2^i$ LSDs of the $TPP(i)(2k)_{2^n+2^i}$ directly to the $2^i$ LSDs of the sum output. As a result, adder(k) has a result of $(2^n + 2^{i+1})$ digits long. Meanwhile, due to (11), the output of adder(k) in Stage $(i+1)$ has a maximum length of $(2^n + 2^{i+1})$ (Lemma 1), indicating that the carry-out is still 0.
\[
TPP(i+1)(k) = \sum_{j=0}^{1} TPP(i)(2k + j)_{2^n+2^i}(10)^j
\]
\[
= \sum_{j=0}^{3} TPP(i-1)(4k + j)_{2^n+2^i-1}(10)^j
\]
\[
= \ldots
\]
\[
= \sum_{j=0}^{2^{i+1}-1} TPP(0)(2^{i+1}k + j)_{2^n+1}(10)^j
\]
\[
= \sum_{j=0}^{2^{i+1}-1} PP(2^{i+1}k + j)_{2^n+1}(10)^j
\]
\[
= A_{2^n} \times \sum_{j=0}^{2^{i+1}-1} b_{(2^{i+1}k+j)}(10)^j
\] (11)

Since the two inputs of adder(k+1) are \(TPP(i)(2k + 2)_{2^n+2^i}\) and \(TPP(i)(2k + 3)_{2^n+2^i}\), which are \((2 \times 2^i)\)-digit left-shifted with respect to the inputs of adder(k), i.e. \(TPP(i)(2k)_{2^n+2^i}\) and \(TPP(i)(2k + 1)_{2^n+2^i}\), respectively, the output of adder(k+1) in Stage \((i+1)\) is \(2^{i+1}\)-digit left-shifted with respect to the adder(k) in the same stage.  

Since the carry-out of each adder is 0, adders in the same stage could operate independently and in parallel with exactly the same critical path delay. Based on above theorem, we implement a 4-stage 8421 PPA tree for the 16-by-16 decimal multiplication, and generate a 32-digit product (Figure 5). In addition, pipeline could be applied to each stage to further improve the data throughput.
Figure 5. Accumulation of partial products for 16-by-16 decimal multiplication
CHAPTER 5

PROPOSED 4221 MULTIPLIER

To take full advantage of the simple logic of 9’s complement and CSAs in 4221, we implement a novel 4221 full adder, so that the 8421-4221 conversion as required in [17] [23] [26] is no longer needed in our 4221 multipliers, and all the inputs and the outputs of the multipliers as well as the internal intermediate results can be represented directly in 4221 BCD. However, due to the redundant and discontinuous nature of 4221, we only use the codes listed in the left sub-column of the 4221 column in TABLE 1, when there are two representations for one decimal value. This way, we can avoid the so-called many-to-many 5211-4221 recoding and the high complexity of 4221 full adder logic.

A. 4221 BCD PPG

The PPG decoding algorithm and structure is similar to what is shown in our 8421-5421 BCD multiplier (Figure 3). Also similar to the 8421 multiplier, we can obtain $2A_n$ in the 4221 codes by recoding $A_n$ from 4221 to 5211 and then left-shifting the 5211-coded $A_n$ by 1 bit, and $5A_n$ by left-shifting the 4211-coded $A_n$ by 3 bits to get the 5211 coded in 5211 and recoding such 5211 from 5211 to 4221 [23] [26]. However, since we force each decimal value to be represented by one unique 4221 BCD code, we now can calculate $2A_n$ of 4221 directly by (12), which is much more complex than that in (8) for 8421-encoded $2A_n$, and $5A_n$ by recoding from 5211 to 4221 in (13). In addition, the Boolean expressions for $b0_i$, $b1_i$ and $OP$ (14)–(16) in 4221 are more complicated than those in 8421-5421 PPG. Even so, given the simpler 9’s complement in 4221, the PPG for 4221 may still hold its performance advantage.
For each 4221 PPG, after achieving OP and the two 4221-coded intermediate partial products (the second intermediate partial product may be negative), we can add them up, based on following two schemes, to ensure that the operands are positive for the following two different 4221 PPAs. One scheme uses a 4221 CLA, which is based on the 4221 full adder presented below, to generate one positive partial product for the tree-structured PPA, the same as what is shown in our 8421-5421 multiplier (Chapter 3). In another scheme, we use a multi-digit 4221 CSA [26] to generate another two positive 4221-coded intermediate partial products for each PPG module (32 intermediate partial products for the 16 PPG modules in the 16-by-16 multiplication); afterwards, we use a 32:2 PPR and a 32-digit 4221 CLA to accumulate the 32 positive intermediate partial products to obtain the final product represented in 4221 BCD codes.

\[
\begin{align*}
    a2_i[0] &= a_{i-1}[3] \\
    a2_i[1] &= \overline{a_{i-1}[3]}a_i[2] + a_i[4]a_i[1]a_i[0] + a_i[2]a_i[1]\overline{a_i[0]} + a_i[2]a_i[1]a_i[0] \\
    &\quad + a_{i-1}[3]a_i[2]a_i[1]\overline{a_i[0]} + a_{i-1}[3]a_i[2]a_i[1]a_i[0] \\
    a2_i[2] &= a_{i-1}[3]a_i[2]a_i[1]a_i[0] + a_{i-1}[3]a_i[2]a_i[1]\overline{a_i[0]} + a_i[1]a_i[0] \\
    &\quad + a_i[2]a_i[1] \\
    a2_i[3] &= a_{i-1}[3]a_i[2]a_i[1]a_i[0] + a_{i-1}[3]a_i[2]a_i[1]\overline{a_i[0]} + a_i[1]a_i[0] \\
    &\quad + a_i[2]a_i[1] \\
    Y[0] &= X[3]X[1]\overline{X[0]} + \overline{X[3]}X[1]X[0] + X[3]X[1]\overline{X[0]} + \overline{X[3]}X[1]\overline{X[0]} \\
\end{align*}
\]
\[ b_{0i}[0] = b_{i}[3]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1] + b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] \]
\[ + b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] \]
\[ b_{0i}[1] = b_{i}[3]b_{i}[2]b_{i}[1] + b_{i}[3]b_{i}[2]b_{i}[1] + b_{i}[2]b_{i}[1]b_{i}[0] \]
\[ + b_{i}[2]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] \]
\[ b_{1i}[0] = b_{i}[3]b_{i}[2] + b_{i}[3]b_{i}[1] + b_{i}[2]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1] \]
\[ + b_{i}[3]b_{i}[1]b_{i}[0] \]
\[ b_{1i}[1] = b_{i}[3]b_{i}[2]b_{i}[1] \]
\[ OP = b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1] \]
\[ + b_{i}[2]b_{i}[1]b_{i}[0] + b_{i}[3]b_{i}[2]b_{i}[1]b_{i}[0] \]

**B. 4221 BCD Addition and PPA**

We build a 1-digit 4221 full adder; that is, adding two 1-digit inputs, \(a[3:0]\) and \(b[3:0]\), and a single-bit carry-in, \(cin\), gives a 1-digit sum \(s[3:0]\), a single-bit carry-out, \(cout\), a single-bit carry-generation, \(gdigit\) and a single-bit carry-propagation, \(pdigit\). Let us define \(g[i] = a[i] \& b[i]\), \(p[i] = a[i] \mid b[i]\), \(h[i] = a[i] \^ \ b[i]\), for \(i = 0, 1, 2, 3\). Thus, we can get the logic for 4221 full adder, as in (17). Obviously, the logic of 4221 full adder is much more complicated than that of 8421, and so is the 4221 CLA even with the same structure as the 8421 CLA.

A PPR with 32 32:2 4221 CSA trees is shown in Figure 6. The 16 generated 4221-coded partial product pairs are separated by horizontal grids, and they read as \(PP0_{17}, PP1_{17}\), and so on, from the top to the bottom. Each partial product pair includes its first intermediate partial product, represented as a row with black dots, and its second intermediate partial product, represented as a row with grey dots. Each column represents
a 32:2 4221 CSA tree, as depicted in Figure 7, where each CSA represents a 3:2 4-bit binary CSA and each “×2” block is a 2a4 module [26]. The module “×1” is to ensure the correct 4221 representation for the 4221 full adder. Each CSA tree adds up the digits of the same weight from the 32 positive intermediate partial products, propagates the carries, and outputs two 4221 digits with the same weight; thus, the 32 32:2 CSA trees reduce the 32 positive intermediate partial products into two 32-digit intermediate products coded in 4221. Finally, a 32-digit 4221 CLA adds up the results of the PPR (the two 32-digit intermediate products) and achieves the final 4221 product. This architecture is similar to the PPA part of the “Radix-5” approach in [26], but with one major distinction: in this multiplier, no 4221-8421 recoding logic is needed.
\[ cout = gdigit + pdigit \cdot cin \]
\[ s[0] = h[0] \oplus cin \]
\[ + g[3]g[1](g[0] + p[0]cin) \]
\[ + p[2]g[1] + p[2]p[0]cin + g[1](g[0] + p[0]cin + \overline{p[0]} \overline{cin}) \]
\[ + p[1](g[0] \overline{cin} + h[0] \overline{cin}) \right. \]
\[ + \text{cout}(\text{cin}g[3]p[1]g[0] + p[1]p[0] \overline{p[0]} \overline{cin}) \]
\[ + p[1]g[0]cin) \]
\[ + g[3]g[1] + p[1]g[2](p[0] + cin) + g[2]g[0]cin \]
\[ + p[2]p[1]g[0]cin) \]
Figure 6. 32 parallel 4221 32:2 CSA Trees for PPR

Figure 7. 32:2 4221 CSA Tree for PPR
CHAPTER 6
EVALUATION AND COMPARISON

In the latest IEEE 754-2008 standard [3], the significand of a double-precision decimal floating-point could represent the value of a 16-digit decimal number, which is long enough for most real applications. Therefore, in this paper, we implement 16-by-16 decimal multipliers for both 8421 and 4221 operands, and compare them against those in [4] [15] [26], the known most area-time efficient and high performance designs in the literature. All the designs are synthesized with the 90nm technology from TSMC. We use delay and delay-area product as the merits for performance comparisons.

A. Decimal PPG

We first evaluate the combination logic of $2A_n$ and $5A_n$, as listed in TABLE 3. Then we synthesize the PPGs of the three multipliers respectively, as demonstrated in TABLE 4.

It shows that, 8421-5421 recoding is the most area-time efficient for $2A_n$ and $5A_n$ pre-computations, and our 8421-5421-recoding-based PPG is the second best among all the PPGs in terms of delay and delay-area product. Since the 4221 PPG, PP5, uses simple 9’s compliment and 3:2 CSAs for the two intermediate partial products, and it involves no carry propagation, this design achieves the lowest delay and circuit area.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Delay (ns)</th>
<th>Area ($\mu m^2$)</th>
<th>Delay $\times$ Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2A_n$</td>
<td>0.03</td>
<td>622.34</td>
<td>18.67</td>
</tr>
<tr>
<td>8421 [10]</td>
<td>0.03</td>
<td>600.47</td>
<td>18.01</td>
</tr>
<tr>
<td>8421-5421[26]</td>
<td>0.03</td>
<td>1281.37</td>
<td>60.07</td>
</tr>
<tr>
<td>4221-5211 (12)</td>
<td>0.05</td>
<td>1450.01</td>
<td>45.50</td>
</tr>
<tr>
<td>$5A_n$</td>
<td>0.03</td>
<td>751.46</td>
<td>22.54</td>
</tr>
<tr>
<td>5421-8421[26]</td>
<td>0.03</td>
<td>975.84</td>
<td>29.28</td>
</tr>
<tr>
<td>5211-4221 (13)</td>
<td>0.03</td>
<td>975.84</td>
<td>29.28</td>
</tr>
</tbody>
</table>
TABLE 4. 16-DIGIT PPG COMPARISON

<table>
<thead>
<tr>
<th>PPG</th>
<th>Delay (ns)</th>
<th>Area ((\mu m^2))</th>
<th>Delay \times Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP0 (8421 [10])</td>
<td>0.50</td>
<td>8185.7</td>
<td>4092.8</td>
</tr>
<tr>
<td>PP1 (8421 [4])</td>
<td>0.50</td>
<td>10524.7</td>
<td>5262.4</td>
</tr>
<tr>
<td>PP2 (8421 [15])</td>
<td>0.70</td>
<td>9687.5</td>
<td>6781.3</td>
</tr>
<tr>
<td>PP3 (8421-5421, Chapter 3)</td>
<td>0.50</td>
<td>6313.0</td>
<td>3156.5</td>
</tr>
<tr>
<td>PP4 (4221-5211 w/ 4221 CLA, Chapter 4)</td>
<td>0.50</td>
<td>9439.5</td>
<td>4719.8</td>
</tr>
<tr>
<td>PP5 (4221-5211 w/ 4221 CSA, Chapter 4)</td>
<td>0.30</td>
<td>4114.4</td>
<td>1234.3</td>
</tr>
</tbody>
</table>

B. Decimal Addition

To verify the improvement of our proposed multi-digit 8421 BCD CLA, we first construct multi-digit adders of different word lengths for each of the 8421 BCD adders and the 4221 BCD adder. The synthesis results are shown in TABLE 5. Five designs are compared.

1) norm 8421: Baseline design. Each digit is corrected after the binary addition if it is larger than 9; this process continues until all the multiplier digits are exhausted.

2) cla 8421: [27]; 2 bytes are clustered for the generation of group carry-lookahead.

3) mcla 8421: Proposed 8421 CLA as described in Chapter 3.

4) zcla 8421: In this design, we replace the L and K in [27] with the aforementioned carry-propagation and carry-generation (Chapter 3), respectively; 4 digits are grouped together with the creation of a group carry-lookahead.

5) 4221: Proposed 4221 adder as described in Chapter 4.

One can see that, for the 1-digit 8421 adder, cla is the most area-time efficient because there involves no extra carry-lookahead logic; but when the digit length increases, the mcla has the lowest delay-area product, because the carry-lookahead for each digit makes the mcla tend to have lower carry delay than that in cla, while it keeps the simplest combination logic for computing the sum digit and carry-out bit. On the other hand, due to their complex logic, 4221 full adders show a consistent high delay-area product value.
### TABLE 5. ADDER COMPARISON

<table>
<thead>
<tr>
<th>digit</th>
<th>BCD adders</th>
<th>Delay (ns)</th>
<th>Area ($\mu m^2$)</th>
<th>Delay × Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Norm 8421</td>
<td>0.11</td>
<td>433.2</td>
<td>47.7</td>
</tr>
<tr>
<td></td>
<td>Cla 8421</td>
<td>0.1</td>
<td>194.0</td>
<td>19.4</td>
</tr>
<tr>
<td></td>
<td>Mcla 8421</td>
<td>0.1</td>
<td>282.2</td>
<td>28.2</td>
</tr>
<tr>
<td></td>
<td>Zcla 8421</td>
<td>0.1</td>
<td>298.5</td>
<td>29.8</td>
</tr>
<tr>
<td></td>
<td>4221</td>
<td>0.11</td>
<td>823.4</td>
<td>90.6</td>
</tr>
<tr>
<td>4</td>
<td>Norm 8421</td>
<td>0.22</td>
<td>1888.2</td>
<td>415.4</td>
</tr>
<tr>
<td></td>
<td>Cla 8421</td>
<td>0.17</td>
<td>1119.8</td>
<td>190.4</td>
</tr>
<tr>
<td></td>
<td>Mcla 8421</td>
<td>0.15</td>
<td>1258.1</td>
<td>188.7</td>
</tr>
<tr>
<td></td>
<td>Zcla 8421</td>
<td>0.16</td>
<td>1941.8</td>
<td>310.7</td>
</tr>
<tr>
<td></td>
<td>4221</td>
<td>0.22</td>
<td>1771.1</td>
<td>389.6</td>
</tr>
<tr>
<td>8</td>
<td>Norm 8421</td>
<td>0.3</td>
<td>2409.6</td>
<td>722.9</td>
</tr>
<tr>
<td></td>
<td>Cla 8421</td>
<td>0.21</td>
<td>2309.4</td>
<td>485.0</td>
</tr>
<tr>
<td></td>
<td>Mcla 8421</td>
<td>0.2</td>
<td>2187.4</td>
<td>437.5</td>
</tr>
<tr>
<td></td>
<td>Zcla 8421</td>
<td>0.21</td>
<td>3029.9</td>
<td>636.3</td>
</tr>
<tr>
<td></td>
<td>4221</td>
<td>0.26</td>
<td>3026.3</td>
<td>786.8</td>
</tr>
<tr>
<td>12</td>
<td>Norm 8421</td>
<td>0.4</td>
<td>3595.0</td>
<td>1438.0</td>
</tr>
<tr>
<td></td>
<td>Cla 8421</td>
<td>0.25</td>
<td>2957.9</td>
<td>739.5</td>
</tr>
<tr>
<td></td>
<td>Mcla 8421</td>
<td>0.25</td>
<td>2867.6</td>
<td>716.9</td>
</tr>
<tr>
<td></td>
<td>Zcla 8421</td>
<td>0.25</td>
<td>3212.6</td>
<td>803.2</td>
</tr>
<tr>
<td></td>
<td>4221</td>
<td>0.27</td>
<td>4786.8</td>
<td>1292.4</td>
</tr>
<tr>
<td>16</td>
<td>Norm 8421</td>
<td>0.5</td>
<td>4491.9</td>
<td>2245.9</td>
</tr>
<tr>
<td></td>
<td>Cla 8421</td>
<td>0.3</td>
<td>3778.5</td>
<td>1133.5</td>
</tr>
<tr>
<td></td>
<td>Mcla 8421</td>
<td>0.3</td>
<td>3389.0</td>
<td>1016.7</td>
</tr>
<tr>
<td></td>
<td>Zcla 8421</td>
<td>0.3</td>
<td>3752.4</td>
<td>1125.7</td>
</tr>
<tr>
<td></td>
<td>4221</td>
<td>0.3</td>
<td>6236.1</td>
<td>1870.8</td>
</tr>
</tbody>
</table>

### TABLE 6. PPR AND PPA FOR 16-BY-16 DECIMAL MULTIPLICATION

<table>
<thead>
<tr>
<th>Module</th>
<th>Delay (ns)</th>
<th>Area ($\mu m^2$)</th>
<th>Delay × Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>8421 PPA</td>
<td>1</td>
<td>63,149</td>
<td>63,149</td>
</tr>
<tr>
<td>4221 PPA1 w/ 4221 CLAs</td>
<td>1.23</td>
<td>98,487</td>
<td>121,139</td>
</tr>
<tr>
<td>4221 32:2 CSA</td>
<td>0.73</td>
<td>5,810</td>
<td>3,548</td>
</tr>
<tr>
<td>4221 PPR</td>
<td>1</td>
<td>91,990</td>
<td>91,990</td>
</tr>
<tr>
<td>4221 PPA2 w/ PPR</td>
<td>1.26</td>
<td>125,280</td>
<td>157,853</td>
</tr>
</tbody>
</table>

### TABLE 7. 16-BY-16 DECIMAL MULTIPLICATIONS

<table>
<thead>
<tr>
<th>BCD Multipliers</th>
<th>Delay (ns)</th>
<th>Area ($\mu m^2$)</th>
<th>Delay × Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>8421 Multiplier [15]</td>
<td>2.65</td>
<td>300,000</td>
<td>795,000</td>
</tr>
<tr>
<td>8421-5421 Multiplier</td>
<td>1.46</td>
<td>181,873</td>
<td>265,535</td>
</tr>
<tr>
<td>4221 Multiplier1 w/ PPA1</td>
<td>1.70</td>
<td>263,089</td>
<td>447,251</td>
</tr>
<tr>
<td>4221 Multiplier2 w/ PPA2</td>
<td>1.55</td>
<td>205,103</td>
<td>317,909</td>
</tr>
</tbody>
</table>
C. Decimal PPA

Here we evaluate the delays and circuit areas of the proposed 8421 PPA (Chapter 3) and two 4221 PPAs, where the design of 4221 PPA1 uses 4221 CLAs organized as the same tree structure as the 8421 PPA, and the 4221 PPA2 uses 32:2 4221 CSA trees and a 32-digit 4221 full adder proposed in Chapter 4. The synthesis results are presented in Table 6. Due to the simplicity of 8421 CLAs, 8421 PPA has lower delay as well as delay-area product than those of 4221 PPA1. On the other hand, although each 32:2 CSA tree consumes a small delay and circuit area, the long carry propagation among them increases the delay significantly, and the 32-digit 4221 CLA has a very negative impact on the performance. Overall speaking, 8421 PPA has the lowest delay and is the most area-time efficient among all the designs.

D. Decimal Multiplication

Synthesis results of the proposed multipliers using 8421 and 4221 BCD codes are demonstrated in Table 7. Due to simplified pre-computation logic with 8421-5421 recoding and PPG structure, and the high-performance area-efficient 4-stage tree structure 8421 PPA, our 8421-5421 decimal multiplier outperforms all the others. Its delay and time-area product are only 55% and 33.4% of those in [15], respectively.

Besides, following the scaling methodology suggested in [26], we compare the ratios of our multipliers and the architectures proposed in [26] over the one in [15], (TABLE 8). It shows that our 8421-5421 multiplier outperforms the “Radix-10” multiplier in [26] by 42.26% in delay and 32% in area-time efficiency, and the “Radix-5” multiplier in [26] by 27.5% and 44.6%, respectively. Our 4221 multiplier using 4221
CSAs outperforms the “Radix-10” [26] by 34.87% in delay and 10% in delay-area product, and the “Radix-5” [26] by 20% and 20.81%, respectively.

Putting everything together, one can see that the proposed 8421-5421 decimal multiplier has the lowest delay and the highest area-time efficiency among all the BCD multiplier designs.

TABLE 8. AREA-DELAY FOR 16-BCD MULTIPLIERS

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Delay Ratio</th>
<th>Area Ratio</th>
<th>Delay-Area Product Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dec. radix-5</td>
<td>1.3</td>
<td>1.1</td>
<td>1.43</td>
</tr>
<tr>
<td>Dec. radix-10</td>
<td>1.45</td>
<td>0.9</td>
<td>1.31</td>
</tr>
<tr>
<td>Proposed in [15]</td>
<td>1.85</td>
<td>1.6</td>
<td>2.96</td>
</tr>
<tr>
<td>8421-5421 Multiplier</td>
<td>1.01</td>
<td>0.97</td>
<td>0.98</td>
</tr>
<tr>
<td>4221 Multiplier1 w/ PPA1</td>
<td>1.19</td>
<td>1.4</td>
<td>1.67</td>
</tr>
<tr>
<td>4221 Multiplier2 w/ PPA2</td>
<td>1.08</td>
<td>1.09</td>
<td>1.18</td>
</tr>
</tbody>
</table>
CHAPTER 7

CONCLUSION

A. Summary

In this thesis, we presented several decimal multipliers using different combinations of 8421, 5421, 4221 and 5211 BCD representations. The proposed 8421-5421-based multiplier has been optimized at both algorithm and architecture levels. In particular, the 8421 multiplier explores a “Radix-5” algorithm for partial product generation, takes advantage of the recoding between 8421 and 5421 to improve the pre-computations of $2A_n$ and $5A_n$, and organizes the 8421 CLAs as a tree structure for parallel partial product accumulation. We also designed the best 4221 multiplier with a novel 4221 full adder where the 4221-to-8421 recoding as required by the existing 4221 multipliers can be totally eliminated. The proposed multipliers were synthesized and compared against each other as well as against the known best designs in the literature, and the results have confirmed that our 8421-5421 decimal multiplier outperforms all the existing BCD multipliers in terms of both delay and area-time efficiency.

B. Future Work

We are working on exploring other decimal representations, such as exceed-3 BCD and multi-radix representations, to further optimize the decimal multiplication. We would do some power consumption analysis on these designs as well.

We also plan to explore other relevant topics such as decimal division or floating-point arithmetic.
APPENDIX (VERILOG HDL CODES)

Hereby, we attach the Verilog HDL codes for some essential modules in our proposed 8421-5421 multiplier and 4221 BCD multipliers.

| 2a (8421 [10]) | module dec_1digit_2a( 
| a, 
| a2 ) ; |
| --- | --- |
| | input [7:0] a; |
| | output [3:0] a2; |
| | assign a2[0] = a[2]&a[1]&~a[0] | a[2]&a[0] | a[3]; |
| endmodule |

| 5a (8421 [10]) | module dec_1digit_5a( 
| a, 
| a5 ) ; |
| --- | --- |
| | input [7:0] a; |
| | output [3:0] a5; |
| endmodule |

| 2a (8421-5421) | module dec_8421to5421( 
| in, 
| out ) ; |
| --- | --- |
| | input [3:0] in; |
| | output [3:0] out; |
|  | endmodule |
| 5a (5421-8421) | module dec_5421to8421( in, out);
|  | input [3:0] in;
|  | output [3:0] out;
|  | assign out[0] = ( in[3] & ~in[0] ) | ( ~in[3] & in[0] );
|  | endmodule |
| 2a (4221) | module dec_1digit_2a(  a,  a2);
|  | output [3:0] a2;
|  | assign a2[0] = a[0];
|  | endmodule |
| 5a (5211-4221) | module dec_5211to4221(  in,  out);
|  | input [3:0] in;
|  | output [3:0] out;
|  | assign out[3] = in[3];
| PPG0 ( [10]) | module dec_16digit_multi_pp_0( a, b, p ) |
| | input [63:0] a; |
| | input [3:0] b; |
| | output [67:0] p; |
| | reg [3:0] btmp0, btmp1; |
| | wire [67:0] ptmp0, ptmp1; |
| | always @( a or b ) begin |
| | case(b) |
| | /* 4'd1: begin |
| | btmp0 = 4'd1; |
| | btmp1 = 4'd0; |
| | end */ |
| | 4'd2: begin |
| | btmp0 = 4'd0; |
| | btmp1 = 4'd2; |
| | end |
| | 4'd3: begin |
| | btmp0 = 4'd1; |
| | btmp1 = 4'd2; |
| | end |
| | 4'd4: begin |
| | btmp0 = 4'd0; |
| | btmp1 = 4'd4; |
| | end |
| | /* 4'd5: begin |
| | btmp0 = 4'd5; |
| | btmp1 = 4'd0; |
| | end */ |
| | 4'd6: begin |
| | btmp0 = 4'd4; |
| | btmp1 = 4'd2; |
| | end |
| | 4'd7: begin |
| | btmp0 = 4'd5; |
begin
  btmp0 = 4'd2;
end

4'd8: begin
  btmp0 = 4'd4;
  btmp1 = 4'd4;
end

4'd9: begin
  btmp0 = 4'd5;
  btmp1 = 4'd4;
end

default: begin
  btmp0 = b;
  btmp1 = 4'd0;
end
endcase
end

dec_16digit_multi_lut_2_0 lut0(.a(a),
  .b(btmp0),
  .p(ptmp0))
);

dec_16digit_multi_lut_2_1 lut1(.a(a),
  .b(btmp1),
  .p(ptmp1))
);

dec_17digit_adder pp(.a(ptmp0),
  .b(ptmp1),
  .cin(1'b0),
  .s(p),
  .cout());
endmodule

PPG1 ( [4] )
module dec_16digit_multi_pp_1(a, b, p);
input [63:0] a;
input [3:0] b;
output [67:0] p;
wire [67:0] p;
reg [2:0] btmp0, btmp1;
wire [67:0] ptmp0, ptmp1;
always @(a or b)
begin
    case(b)
    /* 4'd1:  begin
        btmp0 = 4'd1;
        btmp1 = 4'd0;
    end*/
    4'd2:  begin
        btmp0 = 4'd0;
        btmp1 = 4'd1;
    end
    4'd3:  begin
        btmp0 = 4'd1;
        btmp1 = 4'd1;
    end
    4'd4:  begin
        btmp0 = 4'd2;
        btmp1 = 4'd1;
    end
    4'd5:  begin
        btmp0 = 4'd0;
        btmp1 = 4'd2;
    end
    4'd6:  begin
        btmp0 = 4'd1;
        btmp1 = 4'd2;
    end
    4'd7:  begin
        btmp0 = 4'd2;
        btmp1 = 4'd2;
    end
    4'd8:  begin
        btmp0 = 4'd3;
        btmp1 = 4'd3;
    end
    4'd9:  begin
        btmp0 = 4'd4;
        btmp1 = 4'd4;
    end
    default: begin
...
Pre-Comp0
(8421-5421)

module dec_16digit_lu0_0(
    a,
    b,
    p
);
input [63:0] a;
input [1:0] b;
output [67:0] p;
wire [67:0] p5;
dec_16digit_5a tmp( .a(a),
    .a5(p5)
);
assign p = ( {a, 4'b0} & {68{b[1]}} ) | ( p5 & {68{b[0]}} );

endmodule

btmp0 = b[2:0];
btmp1 = 3'd0;
endcase
end
dec_16digit_multi_lut_1_0 lut0( .a(a),
    .b(btmp0),
    .p(ptmp0)
);
dec_16digit_multi_lut_1_1 lut1( .a(a),
    .b(btmp1),
    .p(ptmp1)
);
dec_17digit_adder pp(.a(ptmp0),
    .b(ptmp1),
    .cin(1'b0),
    .s(p),
    .cout() );
| Pre-Comp1  (8421-5421) | module dec_16digit_lut0_1( a,  
|                      |     b,  
|                      |     p  
|                      | );  
|                      | input [63:0] a;  
|                      | input [1:0] b;  
|                      | output [67:0] p;  
|                      | wire [67:0] p2;  
|                      | dec_16digit_2a tmp( .a(a),  
|                      |     .a2(p2)  
|                      | );  
|                      | assign p = ( p2 & {68{b[1]}} ) | ( {4'b0, a} & {68{b[0]}} ) );  
|                      | endmodule |
| PPG3  (8421-5421 PPG) | module dec_16digit_pp3( a,  
|                      |     b,  
|                      |     p  
|                      | );  
|                      | input [63:0] a;  
|                      | input [3:0] b;  
|                      | output [67:0] p;  
|                      | wire [1:0] btmp0, btmp1;  
|                      | wire [67:0] ptmp0, ptmp1, ptmp2;  
|                      | wire [67:0] ptmp3;  
|                      | wire op;  
|                      | assign btmp0[0] = b[2] | ( b[1] & b[0] );  
|                      | assign btmp0[1] = b[3];  
|                      | assign btmp1[0] = ( b[2] & ~b[0] ) | ( ~b[2] & ~b[1] & b[0] );  
|                      | assign ptmp3 = ( {68{op}} & ptmp2 ) | ( {68{~op}} & ptmp1 ); |
```verilog
module dec_16digit_pp4(    
a,     
b,     
p );
input [63:0] a;
input [3:0] b;
output [67:0] p;
wire [1:0] b0tmp, b1tmp;
wire [67:0] p0tmp, p1tmp, p2tmp;
wire op;

    ~b[3]&b[2]&b[1]&~b[0];  // b[2] | ( b[1] & b[0] );
    b[2]&~b[1]&b[0] | ~b[2]&b[1]&b[0] | b[3]&b[2]&b[1]&~b[0];  //b[3];
16-digit 4221 PPG5 module dec_16digit_pp5( a, b, p0, p1 );
input [63:0] a;
input [3:0] b;
output [67:0] p0, p1;
wire [1:0] b0tmp, b1tmp;
wire [67:0] p0tmp, p1tmp, p2tmp;
wire op;
assign p2tmp = {68{op}} ^ p1tmp;
dec_16digit_lut0_0 lut0(.a(a),
    .b(b0tmp),
    .p(p0tmp));
dec_16digit_lut0_1 lut1(.a(a),
    .b(b1tmp),
    .p(p1tmp));
dec_17digit_4221adder ad(
    .a(p0tmp),
    .b(p2tmp),
    .cin(op),
    .s(p),
    .cout());
endmodule


assign b1tmp[1] = b[3]&b[2]&b[1];


assign p2tmp = {68{op}} ^ p1tmp;

dec_16digit_lut0_0 lut0(.a(a),
    .b(b0tmp),
    .p(p0tmp)
);

dec_16digit_lut0_1 lut1(.a(a),
    .b(b1tmp),
    .p(p1tmp)
);

dec_17digit_4221csa csa(
    .a(p0tmp),
    .b(p2tmp),
    .cin(op),
    .s0(p0),
    .s1(p1)
);

dec_adder_compare

module dec_adder_compare( a, b, cin, s, cout );

input [3:0] a, b;
input cin;
output [3:0] s;
output cout;

wire [4:0] stmp;

assign stmp = a + b + cin;
assign s = ( stmp > 4'd9 )? ( stmp[3:0] + 4'd6 ) : stmp[3:0];
assign cout = ( stmp > 4'd9 );
endmodule

cla 8421 [27]

module dec_2digit_cla(
  a,  
b, 
cin, 
bg, 
bp,  
s, 
cout 
);

input [7:0] a, b;
input cin;

output [7:0] s;
// bg is byte carry generate
// bp is byte carry propagate
output bg, bp;
output cout;

wire [7:0] s;
wire cout;
wire [1:0] k, l;
wire [1:0] g0, p0;
wire bg, bp;
wire citmp;

assign citmp = k[0] | ( l[0] & g0[0] ) | ( l[0] & p0[0] & cin );
assign bp = l[1] & l[0] & p0[1] & p0[0];

dec_cla a0(  .a(a[3:0]),
          .b(b[3:0]),
          .cin(cin),
          )
endmodule
module dec_mcla(a, b, cin, pdigit, gdigit, s, cout);
input [3:0] a, b;
input cin;
output [3:0] s;
output cout;
output pdigit;
output gdigit;
wire [3:0] s;
wire cout;
// pdigit is propagation for multi-digit
// pdigit = 1 only when s = 9
wire pdigit;
wire gdigit;
wire c1;
//wire cout_tmp; // cout_tmp_bar;
wire [3:0] p, g, h;
wire k, l;
// wire [3:0] s_tmp;
assign p[1] = a[1] | b[1];
assign p[0] = a[0] | b[0];
assign g[1] = a[1] & b[1];
assign g[0] = a[0] & b[0];
assign h[1] = a[1] ^ b[1];
assign h[0] = a[0] ^ b[0];
assign cout = k | ( 1 & c1 );
// assign cout = gdigit | ( pdigit & cin );
assign c1 = ( a[0] & b[0] ) | ( a[0] & cin ) | ( b[0] & cin );
assign s[0] = h[0] ^ cin;
// a + b = 9, pdigit = 1; a + b >= 10, gdigit = 1;
zcla 8421

module dec_zcla(a, b, cin, pdigit, gdigit, cout, s);

input [3:0] a, b;
input cin;

output [3:0] s;
output cout;
output pdigit, gdigit;

wire [3:0] s;
wire cout;

wire [3:0] p, g, h;
wire pdigit, gdigit;
wire l;

assign p[1] = a[1] | b[1];
assign p[0] = a[0] | b[0];

assign g[1] = a[1] & b[1];
assign g[0] = a[0] & b[0];

assign h[1] = a[1] ^ b[1];
assign h[0] = a[0] ^ b[0];

assign cout = gdigit | ( pdigit & cin );
assign c1 = ( a[0] & b[0] ) | ( a[0] & cin ) | ( b[0] & cin );

// a + b = 9, pdigit = 1; a + b >= 10, gdigit = 1;

assign s[0] = h[0] ^ cin;

endmodule

4221 full adder

/*
  0000  0
  0001  1
  0010  2
  0011  3
  0110  4
  1001  5
  1100  6
  1101  7
  1110  8
  1111  9
*/
module dec_1digit_4221adder(
  a,
  b,
cin, 
gdigit, 
pdigit, 
s, 
cout 
);

input [3:0] a, b;
input cin;

output [3:0] s;
output gdigit, pdigit, cout;

wire [3:0] s;
wire gdigit, pdigit, cout;

wire [3:0] g, p, h, c;

assign g[0] = a[0] & b[0];
assign g[1] = a[1] & b[1];

assign p[0] = a[0] | b[0];
assign p[1] = a[1] | b[1];

assign h[0] = a[0] ^ b[0];
assign h[1] = a[1] ^ b[1];

assign c[3:0] = 4'b0;

assign cout = gdigit | ( pdigit & cin );

assign s[0] = h[0] ^ cin;


endmodule

module csa32to2(
    in,
    cin,
    cout,
    h,
    s
    );

input [127:0] in;
input [10:0] cin;
output [10:0] cout;
output [3:0] h, s;
wire [3:0] h00, h01, h02, h03, h04, h05, h06, h07, h08, h09, h10, h11, h12, h13, h14, h15, h16, h20, h21, h22, h23, h24, h30, h31, h32, h33, h34, h35, h36, h37;
wire [3:0] s00, s01, s02, s03, s04, s05, s06, s07, s08, s09, s10, s11,
s12, s13, s14, s15, s16, s20, s21, s22, s23, s24, s30, s31, s32, s33, s34, s35, s36, s37;
wire [3:0] mo0, mo1, mo2, mo3, mo4, mo5, mo6, mo7, mo8, mo9;
csa3to2 csa00(
    .a(in[3:0]),
    .b(in[7:4]),
    .c(in[11:8]),
    .h(h00),
    .s(s00)
);

csa3to2 csa01(
    .a(in[15:12]),
    .b(in[19:16]),
    .c(in[23:20]),
    .h(h01),
    .s(s01)
);

csa3to2 csa02(
    .a(in[27:24]),
    .b(in[31:28]),
    .c(in[35:32]),
    .h(h02),
    .s(s02)
);

csa3to2 csa03(
    .a(in[39:36]),
    .b(in[43:40]),
    .c(in[47:44]),
    .h(h03),
    .s(s03)
);

csa3to2 csa04(
    .a(in[51:48]),
    .b(in[55:52]),
    .c(in[59:56]),
    .h(h04),
    .s(s04)
);

csa3to2 csa05(
    .a(in[63:60]),
csw3to2 csa06(
    .a(in[75:72]),
    .b(in[79:76]),
    .c(in[83:80]),
    .h(h06),
    .s(s06)
);


csw3to2 csa07(
    .a(in[87:84]),
    .b(in[91:88]),
    .c(in[95:92]),
    .h(h07),
    .s(s07)
);


csw3to2 csa08(
    .a(in[99:96]),
    .b(in[103:100]),
    .c(in[107:104]),
    .h(h08),
    .s(s08)
);


csw3to2 csa09(
    .a(in[111:108]),
    .b(in[115:112]),
    .c(in[119:116]),
    .h(h09),
    .s(s09)
);


csw3to2 csa10(
    .a(h00),
    .b(h01),
    .c(h02),
    .h(h10),
    .s(s10)
);


csa3to2 csa11(
  .a(s00),
  .b(s01),
  .c(s02),
  .h(h11),
  .s(s11)
);

  csa3to2 csa12(
  .a(h03),
  .b(h04),
  .c(h05),
  .h(h12),
  .s(s12)
);

  csa3to2 csa13(
  .a(s03),
  .b(s04),
  .c(s05),
  .h(h13),
  .s(s13)
);

  csa3to2 csa14(
  .a(h06),
  .b(h07),
  .c(h08),
  .h(h14),
  .s(s14)
);

  csa3to2 csa15(
  .a(s06),
  .b(s07),
  .c(s08),
  .h(h15),
  .s(s15)
);

  csa3to2 csa16(
  .a(s09),
  .b(in[123:120]),
  .c(in[127:124]),
  .h(h16),
  .s(s16))
cسا3تو2 cسا20(
       .a(h10),
       .b(h12),
       .c(h14),
       .h(h20),
       .s(s20)
    );

cسا3تو2 cسا21(
       .a(s10),
       .b(h11),
       .c(s12),
       .h(h21),
       .s(s21)
    );

cسا3تو2 cسا22(
       .a(h13),
       .b(s14),
       .c(h15),
       .h(h22),
       .s(s22)
    );

cسا3تو2 cسا23(
       .a(s13),
       .b(s15),
       .c(s16),
       .h(h23),
       .s(s23)
    );

cسا3تو2 cسا24(
       .a(h23),
       .b(h09),
       .c(h16),
       .h(h24),
       .s(s24)
    );

cسا3تو2 cسا30(
       .a(s20),
       .b(h21),
       .c(s20),
       .h(h21),
       .s(s20)
    );
.c(h22),
 .h(h30),
 .s(s30)
);
csa3to2 csa31(
 .a(s21),
 .b(s22),
 .c(s24),
 .h(h31),
 .s(s31)
);
csa3to2 csa32(
 .a(s30),
 .b(h31),
 .c(h24),
 .h(h32),
 .s(s32)
);
csa3to2 csa33(
 .a(h20),
 .b(h30),
 .c(h32),
 .h(h33),
 .s(s33)
);
dec_1digit_2a m0(
 .a({h33, cin[0]}),
 .a2(mo0)
);
assign cout[0] = h33[3];
dec_1digit_2a m1(
 .a({mo0, cin[1]}),
 .a2(mo1)
);
assign cout[1] = mo0[3];
dec_1digit_2a m2(
 .a({s33, cin[2]}),
 .a2(mo2)
);
assign cout[2] = s33[3];
csa3to2 csa34(
    .a(mo1),
    .b(mo2),
    .c(s32),
    .h(h34),
    .s(s34)
);

dec_1digit_2a m3(
    .a({h34, cin[3]}),
    .a2(mo3)
);
assign cout[3] = h34[3];

dec_1digit_2a m4(
    .a({mo3, cin[4]}),
    .a2(mo4)
);
assign cout[4] = mo3[3];

dec_1digit_2a m5(
    .a({s34, cin[5]}),
    .a2(mo5)
);
assign cout[5] = s34[3];

dec_1digit_2a m6(
    .a({mo5, cin[6]}),
    .a2(mo6)
);
assign cout[6] = mo5[3];

csa3to2 csa35(
    .a(mo6),
    .b(s11),
    .c(s23),
    .h(h35),
    .s(s35)
);

csa3to2 csa36(
    .a(mo4),
    .b(s31),
    .c(h35),
    .h(h36),
    .s(s36)
);
module dec_ppa(
  pp00, pp01, pp10, pp11, pp20, pp21, pp30, pp31, pp40,
  pp41, pp50, pp51, pp60, pp61, pp70, pp71,
  .s(s36)
);

dec_1digit_2a m7(
  .a({h36, cin[7]}),
  .a2(mo7)
);
assign cout[7] = h36[3];

dec_1digit_2a m8(
  .a({mo7, cin[8]}),
  .a2(mo8)
);
assign cout[8] = mo7[3];

dec_1digit_2a m9(
  .a({s36, cin[9]}),
  .a2(mo9)
);
assign cout[9] = s36[3];
csa3to2 csa37(
  .a(mo8),
  .b(mo9),
  .c(s35),
  .h(h37),
  .s(s37)
);

dec_1digit_2a m10(
  .a({h37, cin[10]}),
  .a2(h)
);
assign cout[10] = h37[3];
dec_1digit_recode m11(
  .a(s37),
  .b(s)
);
endmodule

`timescale 1ns/100ps

module dec_ppa(
  pp00, pp01, pp10, pp11, pp20, pp21, pp30, pp31, pp40,
  pp41, pp50, pp51, pp60, pp61, pp70, pp71,
input [67:0] pp80, pp81, pp90, pp91, pp100, pp101, pp110, pp111, pp120, pp121, pp130, pp131, pp140, pp141, pp150, pp151;

output [127:0] p;

wire [127:0] p0, p1;

wire [10:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15;
wire [10:0] c16, c17, c18, c19, c20, c21, c22, c23, c24, c25, c26, c27, c28, c29, c30, c31;

csa32to2 csa0(
    .in( { pp00[3:0], pp01[3:0], 120'b0 } ),
    .cin( 11'b0 ),
    .cout( c0 ),
    .h( p1[3:0] ),
    .s( p0[3:0] )
);

csa32to2 csa1(
    .in( { pp00[7:4], pp01[7:4], pp10[3:0], pp11[3:0], 112'b0 } ),
    .cin( c0 ),
    .cout( c1 ),
    .h( p1[7:4] ),
    .s( p0[7:4] )
);

csa32to2 csa2(
    .in( { pp00[11:8], pp01[11:8], pp10[7:4], pp11[7:4], pp20[3:0], pp21[3:0], 104'b0 } ),
    .cin( c1 ),
    .cout( c2 ),
    .h( p1[11:8] ),
    .s( p0[11:8] )
);
csa32to2 csa3(
    .in( { pp00[15:12], pp01[15:12], pp10[11:8], pp11[11:8], pp20[7:4], pp21[7:4], pp30[3:0], pp31[3:0], 96'b0 } ),
    .cin( c2 ),
    .cout( c3 ),
    .h( p1[15:12] ),
    .s( p0[15:12] )
);  
csa32to2 csa4(
    .in( { pp00[19:16], pp01[19:16], pp10[15:12], pp11[15:12], pp20[11:8], pp21[11:8], pp30[7:4], pp31[7:4], pp40[3:0], pp41[3:0], 88'b0 } ),
    .cin( c3 ),
    .cout( c4 ),
    .h( p1[19:16] ),
    .s( p0[19:16] )
);  
csa32to2 csa5(
    .in( { pp00[23:20], pp01[23:20], pp10[19:16], pp11[19:16], pp20[15:12], pp21[15:12], pp30[11:8], pp31[11:8], pp40[7:4], pp41[7:4], pp50[3:0], pp51[3:0], 80'b0 } ),
    .cin( c4 ),
    .cout( c5 ),
    .h( p1[23:20] ),
    .s( p0[23:20] )
);  
csa32to2 csa6(
    .cin( c5 ),
    .cout( c6 ),
    .h( p1[27:24] ),
    .s( p0[27:24] )
);  
csa32to2 csa7(
    .cin( c6 ),
    .cout( c7 ),
.h( p1[31:28] ),
.s( p0[31:28] )
);

csa32to2 csa8(  
.in( { pp00[35:32], pp01[35:32], pp10[31:28], pp11[31:28],  
  pp20[27:24], pp21[27:24], pp30[23:20], pp31[23:20], pp40[19:16],  
  pp41[19:16], pp50[15:12], pp51[15:12], pp60[11:8], pp61[11:8],  
  pp70[7:4], pp71[7:4], pp80[3:0], pp81[3:0], 56'b0 } ),  
.cin( c7 ),
.cout( c8 ),
.h( p1[35:32] ),
.s( p0[35:32] )
);

csa32to2 csa9(  
.in( { pp00[39:36], pp01[39:36], pp10[35:32], pp11[35:32],  
  pp20[31:28], pp21[31:28], pp30[27:24], pp31[27:24], pp40[23:20],  
  pp41[23:20], pp50[19:16], pp51[19:16], pp60[15:12], pp61[15:12],  
  pp70[11:8], pp71[11:8], pp80[7:4], pp81[7:4], pp90[3:0],  
  pp91[3:0], 48'b0 } ),
.cin( c8 ),
.cout( c9 ),
.h( p1[39:36] ),
.s( p0[39:36] )
);

csa32to2 csa10(  
.in( { pp00[43:40], pp01[43:40], pp10[39:36], pp11[39:36],  
  pp20[35:32], pp21[35:32], pp30[31:28], pp31[31:28], pp40[27:24],  
  pp41[27:24], pp50[23:20], pp51[23:20], pp60[19:16], pp61[19:16],  
  pp70[15:12], pp71[15:12], pp80[11:8], pp81[11:8], pp90[7:4],  
  pp91[7:4], pp100[3:0], pp101[3:0], 40'b0 } ),
.cin( c9 ),
.cout( c10 ),
.h( p1[43:40] ),
.s( p0[43:40] )
);

csa32to2 csa11(  
.in( { pp00[47:44], pp01[47:44], pp10[43:40], pp11[43:40],  
  pp20[39:36], pp21[39:36], pp30[35:32], pp31[35:32], pp40[31:28],  
  pp41[31:28], pp50[27:24], pp51[27:24], pp60[23:20], pp61[23:20],  
  pp70[19:16], pp71[19:16], pp80[15:12], pp81[15:12], pp90[11:8],  
  pp91[11:8], pp100[7:4], pp101[7:4], pp110[3:0], pp111[3:0],  
  32'b0 } ),
.cin( c10 ),
.cout( c11 ),
.h( p1[47:44] ),
.s( p0[47:44] )
);

csa32to2 csa12(
   .in( { pp00[51:48], pp01[51:48], pp10[47:44], pp11[47:44],
         pp20[43:40], pp21[43:40], pp30[39:36], pp31[39:36], pp40[35:32],
         pp41[35:32], pp50[31:28], pp51[31:28], pp60[27:24], pp61[27:24],
         pp70[23:20], pp71[23:20], pp80[19:16], pp81[19:16], pp90[15:12],
         pp91[15:12], pp100[11:8], pp101[11:8], pp110[7:4], pp111[7:4],
         pp120[3:0], pp121[3:0], 24'b0 } ),
   .cin( c11 ),
   .cout( c12 ),
   .h( p1[51:48] ),
   .s( p0[51:48] )
);

csa32to2 csa13(
   .in( { pp00[55:52], pp01[55:52], pp10[51:48], pp11[51:48],
         pp20[47:44], pp21[47:44], pp30[43:40], pp31[43:40], pp40[39:36],
         pp41[39:36], pp50[35:32], pp51[35:32], pp60[31:28], pp61[31:28],
         pp70[27:24], pp71[27:24], pp80[23:20], pp81[23:20], pp90[19:16],
         pp91[19:16], pp100[15:12], pp101[15:12], pp110[11:8],
         pp111[11:8], pp120[7:4], pp121[7:4], pp130[3:0], pp131[3:0],
         16'b0 } ),
   .cin( c12 ),
   .cout( c13 ),
   .h( p1[55:52] ),
   .s( p0[55:52] )
);

csa32to2 csa14(
   .in( { pp00[59:56], pp01[59:56], pp10[55:52], pp11[55:52],
         pp20[51:48], pp21[51:48], pp30[47:44], pp31[47:44], pp40[43:40],
         pp41[43:40], pp50[39:36], pp51[39:36], pp60[35:32], pp61[35:32],
         pp70[31:28], pp71[31:28], pp80[27:24], pp81[27:24], pp90[23:20],
         pp91[23:20], pp100[19:16], pp101[19:16], pp110[15:12],
         pp111[15:12], pp120[11:8], pp121[11:8], pp130[7:4], pp131[7:4],
         pp140[3:0], pp141[3:0], 8'b0 } ),
   .cin( c13 ),
   .cout( c14 ),
   .h( p1[59:56] ),
   .s( p0[59:56] )
);
csa32to2 csa15(
    .in( { pp00[63:60], pp01[63:60], pp10[59:56], pp11[59:56],
          pp20[55:52], pp21[55:52], pp30[51:48], pp31[51:48], pp40[47:44],
          pp41[47:44], pp50[43:40], pp51[43:40], pp60[39:36], pp61[39:36],
          pp70[35:32], pp71[35:32], pp80[31:28], pp81[31:28], pp90[27:24],
          pp91[27:24], pp100[23:20], pp101[23:20], pp110[19:16],
          pp111[19:16], pp120[15:12], pp121[15:12], pp130[11:8],
          pp131[11:8], pp140[7:4], pp141[7:4], pp150[3:0], pp151[3:0] } ),
    .cin( c14 ),
    .cout( c15 ),
    .h( p1[63:60] ),
    .s( p0[63:60] )
);  

csa32to2 csa16(
    .in( { pp00[67:64], pp01[67:64], pp10[63:60], pp11[63:60],
          pp20[59:56], pp21[59:56], pp30[55:52], pp31[55:52], pp40[51:48],
          pp41[51:48], pp50[47:44], pp51[47:44], pp60[43:40], pp61[43:40],
          pp70[39:36], pp71[39:36], pp80[35:32], pp81[35:32], pp90[31:28],
          pp91[31:28], pp100[27:24], pp101[27:24], pp110[23:20],
          pp111[23:20], pp120[19:16], pp121[19:16], pp130[15:12],
          pp131[15:12], pp140[11:8], pp141[11:8], pp150[7:4],
          pp151[7:4] } ),
    .cin( c15 ),
    .cout( c16 ),
    .h( p1[67:64] ),
    .s( p0[67:64] )
);  

csa32to2 csa17(
    .in( { 8'b0, pp10[67:64], pp11[67:64], pp20[63:60],
          pp21[63:60], pp30[59:56], pp31[59:56], pp40[55:52], pp41[55:52],
          pp50[51:48], pp51[51:48], pp60[47:44], pp61[47:44], pp70[43:40],
          pp71[43:40], pp80[39:36], pp81[39:36], pp90[35:32], pp91[35:32],
          pp100[31:28], pp101[31:28], pp110[27:24], pp111[27:24],
          pp120[23:20], pp121[23:20], pp130[19:16], pp131[19:16],
    .cin( c16 ),
    .cout( c17 ),
    .h( p1[71:68] ),
    .s( p0[71:68] )
);  

csa32to2 csa18(
    .in( { 16'b0, pp20[67:64], pp21[67:64], pp30[63:60],
          pp31[63:60], pp40[59:56], pp41[59:56], pp50[55:52], pp51[55:52],
          pp60[51:48], pp61[51:48], pp70[47:44], pp71[47:44], pp80[43:40],
          pp81[43:40], pp90[39:36], pp91[39:36], pp100[35:32], pp101[35:32],
          pp110[31:28], pp111[31:28], pp120[27:24], pp121[27:24], pp130[23:20],
          pp131[23:20], pp140[19:16], pp141[19:16], pp150[15:12], pp151[15:12],
    .cin( c18 ),
    .cout( c19 ),
    .h( p1[75:68] ),
    .s( p0[75:68] )
);
   .cin( c17 ),
   .cout( c18 ),
   .h( p1[75:72] ),
   .s( p0[75:72] )
);

csa32to2 csa19(
   .cin( c18 ),
   .cout( c19 ),
   .h( p1[79:76] ),
   .s( p0[79:76] )
);

csa32to2 csa20(
   .cin( c19 ),
   .cout( c20 ),
   .h( p1[83:80] ),
   .s( p0[83:80] )
);

csa32to2 csa21(
   .in( { 40'b0, pp50[67:64], pp51[67:64], pp60[63:60], pp61[63:60], pp70[59:56], pp71[59:56], pp80[55:52], pp81[55:52], pp90[51:48], pp91[51:48], pp100[47:44], pp101[47:44], pp110[43:40], pp111[43:40], pp120[39:36], pp121[39:36], pp130[35:32], pp131[35:32], pp140[31:28], pp141[31:28], pp150[27:24], pp151[27:24] } ),
.cin( c20 ),
.cout( c21 ),
.h( p1[87:84] ),
.s( p0[87:84] )
);

csa32to2 csa22(  
.in( { 48'b0, pp60[67:64], pp61[67:64], pp70[63:60], 
pp71[63:60], pp80[59:56], pp81[59:56], pp90[55:52], pp91[55:52],
pp100[51:48], pp101[51:48], pp110[47:44], pp111[47:44],
pp120[43:40], pp121[43:40], pp130[39:36], pp131[39:36],
pp140[35:32], pp141[35:32], pp150[31:28], pp151[31:28] } ),
.cin( c21 ),
.cout( c22 ),
.h( p1[91:88] ),
.s( p0[91:88] )
);

csa32to2 csa23(  
.in( { 56'b0, pp70[67:64], pp71[67:64], pp80[63:60],
pp81[63:60], pp90[59:56], pp91[59:56], pp100[55:52],
pp101[55:52], pp110[51:48], pp111[51:48], pp120[47:44],
pp121[47:44], pp130[43:40], pp131[43:40], pp140[39:36],
pp141[39:36], pp150[35:32], pp151[35:32] } ),
.cin( c22 ),
.cout( c23 ),
.h( p1[95:92] ),
.s( p0[95:92] )
);

csa32to2 csa24(  
.in( { 64'b0, pp80[67:64], pp81[67:64], pp90[63:60],
pp91[63:60], pp100[59:56], pp101[59:56], pp110[55:52],
pp111[55:52], pp120[51:48], pp121[51:48], pp130[47:44],
pp131[47:44], pp140[43:40], pp141[43:40], pp150[39:36],
pp151[39:36] } ),
.cin( c23 ),
.cout( c24 ),
.h( p1[99:96] ),
.s( p0[99:96] )
);

csa32to2 csa25(  
.in( { 72'b0, pp90[67:64], pp91[67:64], pp100[63:60],
pp101[63:60], pp110[59:56], pp111[59:56], pp120[55:52],
pp121[55:52], pp130[51:48], pp131[51:48], pp140[47:44],
pp141[47:44], pp150[43:40], pp151[43:40] ),
    .cin( c24 ),
    .cout( c25 ),
    .h( p1[103:100] ),
    .s( p0[103:100] )
);

csa32to2 csa26(
    .in( { 80'b0, pp100[67:64], pp101[67:64], pp110[63:60],
        pp111[63:60], pp120[59:56], pp121[59:56], pp130[55:52],
        pp131[55:52], pp140[51:48], pp141[51:48], pp150[47:44],
        pp151[47:44] } ),
    .cin( c25 ),
    .cout( c26 ),
    .h( p1[107:104] ),
    .s( p0[107:104] )
);

csa32to2 csa27(
    .in( { 88'b0, pp110[67:64], pp111[67:64], pp120[63:60],
        pp121[63:60], pp130[59:56], pp131[59:56], pp140[55:52],
        pp141[55:52], pp150[51:48], pp151[51:48] } ),
    .cin( c26 ),
    .cout( c27 ),
    .h( p1[111:108] ),
    .s( p0[111:108] )
);

csa32to2 csa28(
    .in( { 96'b0, pp120[67:64], pp121[67:64], pp130[63:60],
        pp131[63:60], pp140[59:56], pp141[59:56], pp150[55:52],
        pp151[55:52] } ),
    .cin( c27 ),
    .cout( c28 ),
    .h( p1[115:112] ),
    .s( p0[115:112] )
);

csa32to2 csa29(
    .in( { 104'b0, pp130[67:64], pp131[67:64], pp140[63:60],
        pp141[63:60], pp150[59:56], pp151[59:56] } ),
    .cin( c28 ),
    .cout( c29 ),
    .h( p1[119:116] ),
    .s( p0[119:116] )
);
module dec_16digit_multi( a, b, p);

input [63:0] a, b;
output [127:0] p;
wire [127:0] p;
wire [71:0] ptmp00, ptmp01, ptmp02, ptmp03, ptmp04, ptmp05, ptmp06, ptmp07;
wire [79:0] ptmp10, ptmp11, ptmp12, ptmp13;
wire [95:0] ptmp20, ptmp21;

dec_16digit_pp0 mpp0(.a(a), .b(b[3:0]), .p(pp0)));

endmodule
dec_16digit_pp0 mpp1(.a(a),
  .b(b[7:4]),
  .p(pp1)));

dec_16digit_pp0 mpp2(.a(a),
  .b(b[11:8]),
  .p(pp2)));

dec_16digit_pp0 mpp3(.a(a),
  .b(b[15:12]),
  .p(pp3)));

dec_16digit_pp0 mpp4(.a(a),
  .b(b[19:16]),
  .p(pp4)));

dec_16digit_pp0 mpp5(.a(a),
  .b(b[23:20]),
  .p(pp5)));

dec_16digit_pp0 mpp6(.a(a),
  .b(b[27:24]),
  .p(pp6)));

dec_16digit_pp0 mpp7(.a(a),
  .b(b[31:28]),
  .p(pp7)));

dec_16digit_pp0 mpp8(.a(a),
  .b(b[35:32]),
  .p(pp8)));

dec_16digit_pp0 mpp9(.a(a),
  .b(b[39:36]),
  .p(pp9)));

66
dec_16digit_pp0 mpp10(.a(a),
        .b(b[43:40]),
        .p(pp10)
    );

dec_16digit_pp0 mpp11(.a(a),
        .b(b[47:44]),
        .p(pp11)
    );

dec_16digit_pp0 mpp12(.a(a),
        .b(b[51:48]),
        .p(pp12)
    );

dec_16digit_pp0 mpp13(.a(a),
        .b(b[55:52]),
        .p(pp13)
    );

dec_16digit_pp0 mpp14(.a(a),
        .b(b[59:56]),
        .p(pp14)
    );

dec_16digit_pp0 mpp15(.a(a),
        .b(b[63:60]),
        .p(pp15)
    );

dec_17digit_ppa ma00(.a({4'b0, pp0[67:4]}),
        .b(pp1),
        .s(ptmp00[71:4])
    );
assign ptmp00[3:0] = pp0[3:0];

dec_17digit_ppa ma01(.a({4'b0, pp2[67:4]}),
        .b(pp3),
        .s(ptmp01[71:4])
    );
assign ptmp01[3:0] = pp2[3:0];

dec_17digit_ppa ma02(.a({4'b0, pp4[67:4]}),
        .b(pp5),
        .s(ptmp02[71:4])
    );
assign ptmp02[3:0] = pp4[3:0];

dec_17digit_ppa ma03(.a({4'b0, pp6[67:4]}),
    .b(pp7),
    .s(ptmp03[71:4])
);
assign ptmp03[3:0] = pp6[3:0];

dec_17digit_ppa ma04(.a({4'b0, pp8[67:4]}),
    .b(pp9),
    .s(ptmp04[71:4])
);
assign ptmp04[3:0] = pp8[3:0];

dec_17digit_ppa ma05(.a({4'b0, pp10[67:4]}),
    .b(pp11),
    .s(ptmp05[71:4])
);
assign ptmp05[3:0] = pp10[3:0];

dec_17digit_ppa ma06(.a({4'b0, pp12[67:4]}),
    .b(pp13),
    .s(ptmp06[71:4])
);
assign ptmp06[3:0] = pp12[3:0];

dec_17digit_ppa ma07(.a({4'b0, pp14[67:4]}),
    .b(pp15),
    .s(ptmp07[71:4])
);
assign ptmp07[3:0] = pp14[3:0];

dec_18digit_ppa ma10(.a({8'b0, ptmp00[71:8]}),
    .b(ptmp01),
    .s(ptmp10[79:8])
);
assign ptmp10[7:0] = ptmp00[7:0];

dec_18digit_ppa ma11(.a({8'b0, ptmp02[71:8]}),
    .b(ptmp03),
    .s(ptmp11[79:8])
);
assign ptmp11[7:0] = ptmp02[7:0];

dec_18digit_ppa ma12(.a({8'b0, ptmp04[71:8]}),
    .b(ptmp05),
assign ptmp12[7:0] = ptmp04[7:0];

dec_18digit_ppa ma13(.a({8'b0, ptmp06[71:8]}),
                             .b(ptmp07),
                             .s(ptmp13[79:8]))
);
assign ptmp13[7:0] = ptmp06[7:0];

dec_20digit_ppa ma20(.a({16'b0, ptmp10[79:16]}),
                             .b(ptmp11),
                             .s(ptmp20[95:16]))
);
assign ptmp20[15:0] = ptmp10[15:0];

dec_20digit_ppa ma21(.a({16'b0, ptmp12[79:16]}),
                             .b(ptmp13),
                             .s(ptmp21[95:16]))
);
assign ptmp21[15:0] = ptmp12[15:0];

dec_24digit_ppa ma30(.a({32'b0, ptmp20[95:32]}),
                             .b(ptmp21),
                             .s(p[127:32]))
);
assign p[31:0] = ptmp20[31:0];
endmodule
REFERENCE


[12] Alvaro Vazquez and Florent de Dinechin, "Efficient implementation of Parallel BCD


CV

Ming Zhu

Degrees
2013 University of Nevada, Las Vegas (UNLV)
    Master of Science (M.S.) in Electrical Engineering
    Major in Electrical and Computer Engineering

2011 Shanghai Jiao Tong University (SJTU), Min Hang Campus, China
    Bachelor of Science (B.S.) in Engineering
    Major in Microelectronics

Honors and Awards
    The Electrical and Computer Engineering 2012 Outstanding Graduate Student (2013)
    Second Place of “Best Thesis” in College of Engineering (2013)

Work Experience
2011~2013 Graduate Teaching Assistant, University of Nevada Las Vegas

Publications


Thesis title
    On High-Performance Parallel Fixed-Point Decimal Multiplier Designs

Thesis Examination Committee
    Chair, Yingtao Jiang, Ph.D
    Committee member, Mei Yang, Ph.D
    Committee member, Mei Yang, Ph.D
    Graduate College Faculty Representative, Hui Zhao, Ph.D