Partial Product Generation Utilizing the Sum of Operands for Reduced Area Parallel Multipliers

HIROTAKA KAWASHIMA†1 and NAOFUMI TAKAGI†2

We propose a novel method to generate partial products for reduced area parallel multipliers. Our method reduces the total number of partial product bits of parallel multiplication by about half. We call partial products generated by our method Compound Partial Products (CPPs). Each CPP has four candidate values: zero, a part of the multiplicand, a part of the multiplier, and a part of the sum of the operands. Our method selects one from the four candidates according to a pair of a multiplicand bit and a multiplier bit. Multipliers employing the CPPs are approximately 30% smaller than array multipliers without radix-4 Booth’s method, and approximately up to 10% smaller than array multipliers with radix-4 Booth’s method. We also propose an acceleration method of the multipliers using CPPs.

1. Introduction

Multiplication is a fundamental arithmetic operation used in various applications. To achieve fast multiplication, recent processors and ASICs are often equipped with parallel multipliers. Since requirements for multiplier design differ among applications, it is necessary to offer various construction of parallel multipliers for various performances. One of the most significant goals of parallel multiplier design is to reduce the circuit area. In this paper, we propose a novel method of partial product generation for reducing the area of a multiplier.

In general a parallel multiplier consists of a partial product generator, a partial product compressor and a final adder. Basic multiplication generates partial products, each of which is a product of a whole multiplicand and a bit of a multiplier. Radix-4 Booth’s method is well known as a method for reducing the number of partial products. The radix-4 Booth’s method recodes the multiplier represented in binary into radix-4 signed digit representation whose digit set is \{-2, -1, 0, 1, 2\}. Since the recoded multiplier is represented in about half of the number of digits, the number of partial products is reduced by about half. In the cell based design using recent cell libraries, the area of multipliers often becomes smaller by using the radix-4 Booth’s method. When small area multipliers are required, array multipliers using the radix-4 Booth’s method are widely employed.

We propose a new method to reduce the total number of partial product bits by about half. Our method utilizes the sum of the operands (multiplicand and multiplier) to generate the partial products. We call each partial product generated by our method Compound Partial Product (CPP). A CPP has four candidate values: zero, a part of the multiplicand, a part of the multiplier and a part of the sum of the operands. Each CPP is obtained by selecting one from the four candidates. The total number of CPP bits is about half of that of basic partial product bits. We also propose an acceleration method of the multipliers with the proposed method. We divide the addition of the operands into multiple sections to generate CPPs faster and parallelize CPP compression.

Our evaluation shows that the multipliers employing the CPPs are smaller than array multipliers by approximately 30%, and smaller than array multipliers with the radix-4 Booth’s method by approximately 10%. Delay of these multipliers are comparable with that of array multipliers with the radix-4 Booth’s method.

The remainder of this paper is organized as follows: Section 2 proposes partial product generation utilizing the sum of the operands. Section 3 shows reduced area multipliers with the proposed method. Section 4 discusses acceleration of the multipliers with the proposed method. Section 5 evaluates the multipliers with the proposed method. Section 6 concludes this paper.

2. Partial Product Generation Utilizing the Sum of Operands

First, we describe our method for n-bit unsigned multiplication. We assume the multiplicand \(X\) and the multiplier \(Y\) are \(n\)-bit unsigned integers and expressed as \(x_0 x_1 \cdots x_{n-1} x_n\) and \(y_0 y_1 \cdots y_{n-1} y_n\), respectively. The values of \(X\) and \(Y\) are \(\sum_{i=0}^{n-1} 2^i x_i\) and \(\sum_{i=0}^{n-1} 2^i y_i\), respectively. We define \(X_i\) as \(x_i x_{i-1} \cdots x_0\)
and \( Y_i = [y_i y_{i-1} \cdots y_1 y_0] \) for \( 0 \leq i \leq n-1 \). Values of \( X_i \) and \( Y_i \) are \( \sum_{j=0}^{i} 2^{j}x_j \) and \( \sum_{j=0}^{i} 2^{j}y_j \), respectively.

We can transform \( X \times Y \) as follows:

\[
X \times Y = \left( \sum_{i=0}^{n-1} 2^{i}x_i \right) \times \left( \sum_{i=0}^{n-1} 2^{i}y_i \right)
\]

\[
= (2^{n-1}x_{n-1} + X_{n-2}) \times (2^{n-1}y_{n-1} + Y_{n-2})
\]

\[
= X_{n-2} \times Y_{n-2} + 2^{n-1}(2^{n-1}x_{n-1}y_{n-1} + x_{n-1}Y_{n-2} + y_{n-1}X_{n-2})
\]

By transforming iteratively, \( X \times Y \) is calculated as follows:

\[
X \times Y = x_0y_0 + \sum_{i=1}^{n-1} 2^{i}(2^{i}x_iy_i + x_iY_{i-1} + y_iX_{i-1})
\]

This equation is used for serial-serial multiplication\(^3\)\(-\)\(^5\). We call \( 2^{i}x_iy_i + x_iY_{i-1} + y_iX_{i-1} \) the \( i \)-th Compound Partial Product (CPP). The \( i \)-th CPP is denoted by \( P_i \). We can obtain the CPPs efficiently utilizing the sum of the multiplicand and the multiplier. There are four candidates of CPP values according to the combination of the values of \( x_i \) and \( y_i \). \( P_i \) is calculated as follows:

\[
P_i = \begin{cases} 
0 & \text{if (} x_i, y_i \text{) = (0, 0)} \\
X_{i-1} & \text{if (} x_i, y_i \text{) = (0, 1)} \\
Y_{i-1} & \text{if (} x_i, y_i \text{) = (1, 0)} \\
2^i + X_{i-1} + Y_{i-1} & \text{if (} x_i, y_i \text{) = (1, 1)} 
\end{cases}
\]

We need to calculate \( X_{i-1} + Y_{i-1} \) in the case of \( x_i, y_i = (1, 1) \). We define \( S \) as \( X + Y \). \( S \) is an \((n+1)\)-bit unsigned integer and expressed as \([s_n s_{n-1} \cdots s_2 \cdots s_1 s_0]\). We define \( S_i \) as \([s_is_{i-1} \cdots s_2 \cdots s_1 s_0] \). We can express \( P_i \) in an \((i+2)\)-bit unsigned binary representation as follows:

\[
P_i = \begin{cases} 
[0 0 0 0 0 \cdots 0 0] & \text{if (} x_i, y_i \text{) = (0, 0)} \\
[0 0 x_{i-1} x_{i-2} \cdots x_1 x_0] & \text{if (} x_i, y_i \text{) = (0, 1)} \\
[0 0 y_{i-1} y_{i-2} \cdots y_1 y_0] & \text{if (} x_i, y_i \text{) = (1, 0)} \\
[s_i \overline{s_i} s_{i-1} s_{i-2} \cdots s_1 s_0] & \text{if (} x_i, y_i \text{) = (1, 1)} 
\end{cases}
\]

\( \overline{s_i} \) indicates the inversion of \( s_i \). Note that when \( x_i, y_i = (1, 1) \), \( X_{i-1} + Y_{i-1} = S_i \). Since we can obtain all CPPs from \( S \), we calculate \( S \) only once. Our method generates only \( \frac{1}{2}n^2 + \frac{3}{2}n - 1 \) bits in total, while a basic \( n \)-bit multiplier generates \( n^2 \) partial product bits. Our method can reduce the total number of partial product bits by about half. Finally multiplication is performed as

\[
X \times Y = x_0y_0 + \sum_{i=1}^{n-1} 2^{i}P_i.
\]

We show an example of 8-bit unsigned multiplication in Fig. 1. We assume that \( X \) is \([01001110]\) and \( Y \) is \([11100101]\). \( S \) is represented as \([10010011]\). \( S \) is represented as \([00010010]\). \( P_4 \) is represented as \([00000000]\), \( [002x3x2x1x0] = [00011010]\), and \([s_6 s_5 s_4 s_3 s_2 s_1 s_0] = [01110011]\), respectively. The other CPPs are obtained in the same way.

We can apply the proposed method to signed multiplication by slight modification. In signed multiplication, \( X \) and \( Y \) are expressed in two’s complement representation as \([x_{n-1} x_{n-2} \cdots x_1 x_0]\) and \([y_{n-1} y_{n-2} \cdots y_1 y_0]\), respectively. The values of \( X \) and \( Y \) are \(-2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i\) and \(-2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} 2^i y_i\), respectively. For \( i \leq n-2 \), \( X_i \) and \( Y_i \) are expressed in unsigned binary representation as \([x_ix_{i-1} \cdots x_1 x_0]\) and \([y_i y_{i-1} \cdots y_1 y_0]\), respectively. The values of them are \( \sum_{i=0}^{i} 2^i x_i\) and \( \sum_{i=0}^{i} 2^i y_i\), respectively. We define \( S \) as the sum of the operands \( X + Y \). \( S \) is represented in \((n+1)\)-bit two’s complement binary.
Signed multiplication is shown as:

\[ X \times Y = \left( -2^{n-1}x_{n-1} + \sum_{i=0}^{n-2} 2^i x_i \right) \left( -2^{n-1}y_{n-1} + \sum_{i=0}^{n-2} 2^i y_i \right) = X_{n-2} \times Y_{n-2} + 2^{n-1}(2^{n-1}x_{n-1}y_{n-1} - x_{n-1}Y_{n-2} - y_{n-1}X_{n-2}) \]

Since \( x_0y_0 \) and the CPPs for \( i = 1 \) to \( n-2 \) are the same as those of the unsigned multiplication, \( X_{n-2} \times Y_{n-2} \) can be calculated in the same way as the unsigned multiplication. We define the \((n-1)\)-th CPP for signed multiplication \( P_{n-1} \) as \( 2^{n-1}x_{n-1}y_{n-1} - x_{n-1}Y_{n-2} - y_{n-1}X_{n-2} \). There are four candidate values of \( P_{n-1} \) according to \((x_{n-1}, y_{n-1})\) as follows:

\[
P_{n-1} = \begin{cases} 
0 & \text{if } (x_{n-1}, y_{n-1}) = (0,0) \\
-X_{n-2} & \text{if } (x_{n-1}, y_{n-1}) = (0,1) \\
-Y_{n-2} & \text{if } (x_{n-1}, y_{n-1}) = (1,0) \\
2^{n-1} - (X_{n-2} + Y_{n-2}) & \text{if } (x_{n-1}, y_{n-1}) = (1,1) 
\end{cases}
\]

\( P_{n-1} \) is also calculated using the sum of the operands. \( S \) is required only when \((x_{n-1}, y_{n-1}) = (1,1)\). Note that when \((x_{n-1}, y_{n-1}) = (1,1)\), \( X_{n-2} + Y_{n-2} = S_{n-1} \), and that \(-S_{n-1} = -2^n + S_{n-1} + 1\), where \( S_{n-1} \) indicates the bitwise inversion of \( S_{n-1} \). We treat \( S_{n-1} \) as an \( n \)-bit unsigned integer. Then \( P_{n-1} = -2^n + 2^{n-1} + S_{n-1} + 1 \). We define \( P^{*}_{n-1} \) as \( P_{n-1} - 1 \). Then, \( P^{*}_{n-1} \) is expressed in an \((n+1)\)-bit two’s complement binary representation as follows:

\[
P^{*}_{n-1} = \begin{cases} 
\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} & \text{if } (x_{n-1}, y_{n-1}) = (0,0) \\
\begin{bmatrix} 1 & 1 & x_{n-2} & x_{n-3} & \cdots & x_0 \end{bmatrix} & \text{if } (x_{n-1}, y_{n-1}) = (0,1) \\
\begin{bmatrix} 1 & 1 & y_{n-2} & y_{n-3} & \cdots & y_0 \end{bmatrix} & \text{if } (x_{n-1}, y_{n-1}) = (1,0) \\
\begin{bmatrix} s_{n-1} & s_{n-2} & s_{n-3} & \cdots & s_0 \end{bmatrix} & \text{if } (x_{n-1}, y_{n-1}) = (1,1) 
\end{cases}
\]

Finally, signed multiplication is performed as

\[ X \times Y = x_0y_0 + \left( \sum_{i=1}^{n-2} 2^i P^*_i \right) + 2^{n-1}P_{n-1} \]
propagate adder for calculating the sum of the operands. The operand reconder generates \( x_i \land y_i, \overline{x_i} \land y_i \) and \( x_i \land \overline{y_i} \) for \( 0 \leq i \leq n - 1 \). The CPP generator consists of \( \frac{n^2}{2} \) selector cells. Each selector cell generates a CPP bit from \( (x_j, y_j, s_j) \) and \( (x_i \land y_i, \overline{x_i} \land y_i, x_i \land \overline{y_i}) \). The CPP compressor compresses the CPPs into two numbers by carry save additions. Structures of partial product compressors such as array structure and Wallace tree\(^6\) are applicable for the CPP compressor. The final adder is a carry propagate adder, which sums up the two numbers.

Here, we show the function of the selector cells for unsigned multiplication. We define \( p_{i,j} \) as the \( j \)-th least significant bit of \( P_i \). The selector cells calculate \( p_{i,j}, p_{i,i} \) and \( p_{i,i+1} \) where \( 1 \leq i \leq n - 1 \) and \( 0 \leq j \leq i - 1 \) as follows:

\[
\begin{align*}
    p_{i,j} &= ((\overline{x_i} \land y_i) \land x_j) \lor ((x_i \land \overline{y_i}) \land y_j) \lor ((x_i \land y_i) \land s_j) \\
    p_{i,i} &= (x_i \land y_i) \lor \overline{y_i} \\
    p_{i,i+1} &= (x_i \land y_i) \lor s_i
\end{align*}
\]

In Fig. 3, white, gray and black squares indicate the selector cells for \( p_{i,j}, p_{i,i} \) and \( p_{i,i+1} \), respectively. For signed multiplication, we define \( p_{n-1,j}^* \) as the \( j \)-th least significant bit of \( P_{n-1}^* \). \( P_{n-1}^* \) is calculated as follows, where \( 0 \leq j \leq n - 2 \):

\[
\begin{align*}
    p_{n-1,j}^* &= ((\overline{x_{n-1}} \land y_{n-1}) \land x_j) \lor ((x_{n-1} \land y_{n-1}) \land y_j) \lor ((x_{n-1} \land y_{n-1}) \land s_j) \\
    p_{n-1,n-1}^* &= s_{n-1} \lor (x_{n-1} \land y_{n-1}) \\
    p_{n-1,n}^* &= p_{n-1,n-1}^*
\end{align*}
\]

We discuss the delay of the multipliers employing the CPPs. Both the basic partial product generation and the radix-4 Booth’s method generate partial products in constant time. The delay of partial product generation does not depend on the bit width of the operands. On the other hand, CPP generation requires a carry propagate addition, whose delay depends on the bit width of the operands. It could be a disadvantage of the delay of multipliers. Generally \( S \) is obtained in order from the least to the most significant bit. For \( 1 \leq i \leq n - 1 \) and for \( 0 \leq j \leq i - 1 \), \( p_{i,j} \) depend on \( s_j \). Therefore CPP bits are also generated in order from the least to the most significant bit, i.e., from the right side to the left side of each CPP in Fig. 1 (b) and Fig. 2. On the other hand, array-type CPP compressor works from the top to the bottom of the figure. We rearrange the alignment of CPP bits to overcome the disadvantage. We place the CPPs diagonally so that CPP bits are generated from the top to the bottom of the figure. The alignment of CPP bits of unsigned and signed multiplication are shown in Fig. 4 (a) and Fig. 4 (b), respectively. When we place CPPs diagonally, the \( j \)-th least significant bits of all CPPs are placed on the \( j \)-th row. The \( j \)-th least significant bit of CPPs depends on \( s_j \), and therefore all CPP bits generated simultaneously are
in the same row. As indicated above, the array-type CPP compressors compress CPP bits in order from the 0th row to the \((n - 1)\)-th row. When the array-type CPP compressor is used, CPP bits are compressed as soon as they are generated. Therefore, the delay of the operand addition has little impact on the delay of the entire multiplier. Finally, the delay of the multiplier using diagonal alignment and array-type CPP compressor is expected to be comparable with that of array multipliers.

We show the effectiveness of placing the CPPs diagonally. We design multipliers using the proposed method and optimize them under area constraint and no delay constraint. Design conditions are the same as the conditions that we will show in Section 5. We compare two constructions of multipliers, i.e., the one placing CPPs diagonally and the other horizontally. We name them diagonal and horizontal, respectively. We show the area and the delay of diagonal and horizontal optimized under area constraint in Table 1. diagonal are faster than horizontal by approximately 13% and 20% for Rohm 0.18 \(\mu\)m and STARC 90 nm process, respectively. diagonal are slightly smaller than horizontal. For signed multipliers, we have almost the same results as the unsigned multipliers. In the rest of this paper, we use diagonal placement for the multipliers using the CPPs.

4. Acceleration of the Multiplier Employing the CPPs

In this section we show a method for accelerating the multipliers described in Section 3. We accelerate the multipliers by dividing the operand addition into multiple sections and parallelize CPP compression. As an example, we explain the case of dividing the operand addition into two sections. Here, we assume that

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Comparison between horizontal and diagonal placement of CPPs for unsigned multipliers.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rohm 0.18 (\mu)m</td>
</tr>
<tr>
<td></td>
<td>area delay</td>
</tr>
<tr>
<td>16-bit</td>
<td></td>
</tr>
<tr>
<td>horizontal</td>
<td>20,070.1</td>
</tr>
<tr>
<td>diagonal</td>
<td>19,459.4</td>
</tr>
<tr>
<td>32-bit</td>
<td></td>
</tr>
<tr>
<td>horizontal</td>
<td>73,484.7</td>
</tr>
<tr>
<td>diagonal</td>
<td>72,165.1</td>
</tr>
<tr>
<td>64-bit</td>
<td></td>
</tr>
<tr>
<td>horizontal</td>
<td>280,449.5</td>
</tr>
<tr>
<td>diagonal</td>
<td>279,052.3</td>
</tr>
</tbody>
</table>

Fig. 5 An example of CPPs using the acceleration.

X and Y are \(n\)-bit unsigned integers and \(n\) is even for simplicity of explanation. We define \(X_L, X_H, Y_L\) and \(Y_H\) as follows:

\[
X_L = [x_{n-1}x_{n-2} \cdots x_1x_0]
\]

\[
X_H = [x_{n-1}x_{n-2} \cdots x_1x_0]
\]

\[
Y_L = [y_{n-1}y_{n-2} \cdots y_1y_0]
\]

\[
Y_H = [y_{n-1}y_{n-2} \cdots y_1y_0]
\]

\(X_L, X_H, Y_L\) and \(Y_H\) have values \(\sum_{j=0}^{n-1} 2^j x_j\), \(\sum_{j=0}^{n-1} 2^j y_j\), respectively. We define \(S_L\) and \(S_H\) as follows:

\[
S_L = X_L + Y_L
\]

\[
S_H = X_H + Y_H
\]

\(S_L\) and \(S_H\) are represented as \([l_{2l-1} \cdots l_0]\) and \([l_{2l-1} \cdots l_100]\), respectively. Note that \(S_L + 2S_H = S\). We generate the CPP bits with higher weight using the bits of \(S_H\) instead of the higher bits of \(S\). Then, for \(i \geq \frac{n}{2}\), two shorter CPPs, \(P_Li\) and \(P_Hi\), corresponding to \(P_i\) are generated. \(P_Li\) depends on \(S_L\) and consists of \(\frac{n}{2} + 1\) bits. \(P_Hi\) depends on \(S_H\) and consists of \(i + 2 - \frac{n}{2}\) bits. Note that \(P_Li + 2P_Hi = P_i\). A bit diagram of CPPs in this case is shown in Fig. 5 (a). The bits above the dashed line belong to \(P_Li\)'s and those below the dashed line belong to \(P_Hi\)'s. Thus, \(P_Hi\) and \(P_Li\) are generated in parallel. We
divide CPP compressor into two sections, i.e., one for $P_{Hi}$’s and the other for $P_{Lo}$’s, and parallelize CPP compression. The total number of CPP bits increases by $\frac{n}{2}$ bits due to the acceleration method. Since the acceleration method divides the operand addition, an additional bit $l_{\frac{n}{2}}$, i.e., the carry-out bit of $S_{L}$, appears in the sum of the operands. CPP bits corresponding to each row of Fig. 5 (a) depend on each bit of the sum of the operands. Therefore, there are additional CPP bits corresponding to $l_{\frac{n}{2}}$. The number of additional CPP bits is $\frac{n}{2}$. The additional CPP bits are shown on the fifth row in Fig. 5 (a).

We discuss the delay and the area of multipliers with the acceleration. When operand addition is divided into two sections, input width of each operand addition is $\frac{n}{2}$. Therefore the widths of divided CPPs are also smaller than $\frac{n}{2} + 1$. We can place the entire CPP bits in $\left\lfloor \frac{n}{2} \right\rfloor + 1$ rows for unsigned multiplication.

We show the alignment of CPP bits in Fig. 5 (b). For signed multiplication, the number of rows is $\left\lfloor \frac{n}{2} \right\rfloor + 2$. CPP bits are generated in order from the first row to the $\left\lfloor \frac{n}{2} \right\rfloor + 1$-th row. As discussed in Section 3, when array-type CPP compressor is used, the delay of the operand addition has little impact on the delay of the entire multiplier. Therefore the delay of the entire multiplier is mostly determined by the delay of the CPP compressor and the final adder. The delay of the multipliers using CPPs and the acceleration method is expected to be comparable by the delay of the CPP compressor and the final adder. The delay of the multiplier is mostly determined by the delay of the CPP compressor and the final adder. The delay of the multipliers using CPPs and the acceleration method is expected to be comparable by the delay of the CPP compressor and the final adder.

We compare these multipliers with the following two multipliers.

array : array multipliers without radix-4 Booth’s method

array+Booth : array multipliers with radix-4 Booth’s method

Table 2 shows the area and the delay of multipliers optimized under only area constraint. In this condition, we minimize the area of the multipliers and use no delay constraint. As a result, all adders used in the multipliers are ripple carry adders. Table 3 shows the area and the delay of multipliers optimized under both area and delay constraint. In this condition, we try to minimize the delay of the multipliers. However, if we minimize the delay exactly, the area of the multipliers increases explosively. Therefore, we relax the delay constraint slightly from the minimum delay so that the area can take a reasonable value. All adders used in the multipliers are the adder module DW01_add from DesignWare IP Library, Synopsys, Inc.

Our experimental results show the following two features. One is that the area of multipliers employing the CPPs are smaller than that of conventional multipliers under area constraint. The area of CPP+array is approximately 20%, 30% and 35% smaller than array for 16-bit, 32-bit and 64-bit, respectively. Compared with array+Booth, the area of CPP+array is approximately 10% smaller for both Rohm 0.18 $\mu$m and STARC 90 nm. The other feature is that the multipliers employing the CPPs are smaller and slightly slower than the array multipliers using the radix-4 Booth’s method under area and delay constraint. As described above, CPP+div2+array is slightly slower than array+Booth. CPP+div2+array is up to 22.7% and 12.2% smaller than array+Booth for Rohm 0.18 $\mu$m and STARC 90 nm, respectively. These features show the useful-
Table 2  Area (µm²) and delay (ns) of multipliers under area constraint.

<table>
<thead>
<tr>
<th></th>
<th>Rohm 0.18 µm</th>
<th>STARC 90 nm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>area</td>
<td>delay</td>
</tr>
<tr>
<td>unsigned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit array</td>
<td>24,946.8</td>
<td>11.47</td>
</tr>
<tr>
<td>array+Booth</td>
<td>22,902.3</td>
<td>13.37</td>
</tr>
<tr>
<td>CPP+array</td>
<td>19,459.4</td>
<td>11.35</td>
</tr>
<tr>
<td>CPP+div2+array</td>
<td>20,470.9</td>
<td>11.42</td>
</tr>
<tr>
<td>32-bit array</td>
<td>103,775.1</td>
<td>23.67</td>
</tr>
<tr>
<td>array+Booth</td>
<td>84,552.3</td>
<td>22.37</td>
</tr>
<tr>
<td>CPP+array</td>
<td>72,165.1</td>
<td>22.20</td>
</tr>
<tr>
<td>CPP+div2+array</td>
<td>74,397.7</td>
<td>22.58</td>
</tr>
<tr>
<td>64-bit array</td>
<td>283,645.9</td>
<td>43.41</td>
</tr>
<tr>
<td>signed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16-bit array</td>
<td>23,753.1</td>
<td>11.61</td>
</tr>
<tr>
<td>array+Booth</td>
<td>21,618.5</td>
<td>13.08</td>
</tr>
<tr>
<td>CPP+array</td>
<td>19,615.9</td>
<td>11.40</td>
</tr>
<tr>
<td>CPP+div2+array</td>
<td>21,652.1</td>
<td>11.42</td>
</tr>
<tr>
<td>32-bit array</td>
<td>103,362.3</td>
<td>23.83</td>
</tr>
<tr>
<td>array+Booth</td>
<td>81,821.4</td>
<td>22.36</td>
</tr>
<tr>
<td>CPP+array</td>
<td>72,483.0</td>
<td>22.29</td>
</tr>
<tr>
<td>CPP+div2+array</td>
<td>76,746.4</td>
<td>22.24</td>
</tr>
<tr>
<td>64-bit array</td>
<td>322,622.9</td>
<td>41.93</td>
</tr>
</tbody>
</table>

Here, we discuss the effect of the acceleration method proposed in Section 4. The acceleration method is not effective under the area constraint and no delay constraint in Table 2. In this condition, it is considered for the final adders in the multipliers to take the ripple carry structures, which is the most simple, the smallest and the slowest structure of a parallel adder. The final adders are considered to be critical for the delay. Even if the CPP compressors are accelerated, it does not lead to the speed-up of the entire multipliers. On the other hand, Table 3 shows that the acceleration method works effectively and all CPP+div2+array’s are faster than CPP+array’s. In Table 3, we set delay constraint, therefore the final adders in the multipliers take the faster structures than ripple carry structure. The array-type CPP compressors are considered to be critical for the delay. Therefore, accelerating the array-type CPP compressors leads to the speed-up of the entire multipliers.

Although the purpose of this paper is proposing the reduced area multipliers, readers may also have an interest in the delay of the multiplier. In fast multipliers...
Wallace tree is widely used. We show the experimental results of the following multipliers using Wallace tree.

**CPP+Wallace** : Multipliers using a tree-type CPP compressor. The operand addition is not divided.

**Wallace** : Wallace multipliers without radix-4 Booth’s method

**Wallace+Booth** : Wallace multipliers with radix-4 Booth’s method

Note that the result using Wallace tree is a reference information and we show only the simple constructions listed above. Table 4 shows the area and the delay of the multipliers using Wallace tree. In Table 4(a), the similar trend holds as Table 2. In Table 4(b), **CPP+Wallace** is slower and smaller than **Wallace** and **Wallace+Booth**. **Wallace** and **Wallace+Booth** generates partial products in constant time. **CPP+Wallace** does not generate the CPPs in constant time because **CPP+Wallace** requires a carry propagate addition in the operand addition. The carry propagate addition in CPP generation becomes disadvantageous for the delay.

### 6. Conclusion

We have proposed a novel partial product generation method for reduced area parallel multipliers. The proposed method reduces the total number of partial product bits by using the sum of the operands effectively. We call partial products generated by the proposed method the compound partial products (CPP). The total number of CPP bits is about half of the number of basic partial product bits. We have also proposed a method to accelerate the multipliers using CPPs. Our method for acceleration divides the operand addition into multiple sections and generates CPPs simultaneously. CPP compression is also parallelized into multiple sections. We designed the multipliers using CPPs and compared them with conventional multipliers. Our experimental results show the usefulness of the proposed method. The multipliers using CPPs can be an alternative to the array multipliers using radix-4 Booth’s method.

In this paper, we have evaluated only the case of dividing the operand addition into two sections. The multipliers employing the CPPs can take many other constructions according to the number of sections and the positions to divide. Finding the optimal construction is one of the future tasks. Our method
performs addition and multiplication simultaneously and therefore can be used more efficiently for applications where the sum and the product are required simultaneously.

Acknowledgments This work is supported by Core Research for Evolutional Science and Technology (CREST) from Japan Science and Technology Agency and VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc., Cadence Design Systems, Inc., and Rohm Corporation.

References

(Received November 9, 2010)
(Revised February 28, 2011)
(Accepted April 24, 2011)
(Released August 10, 2011)

(Recommended by Associate Editor: Yutaka Tamiya)

Hirotaka Kawashima received his B.E. and Master degree of information science from Nagoya University, Nagoya, Japan, in 2002 and 2007, respectively. He joined Center for Embedded Computing Systems, Nagoya University as a researcher in 2010 and currently working toward Ph.D. degree. His current interests include low power system design and computer arithmetic.

Naofumi Takagi received his B.E., M.E., and Ph.D. degrees in information science from Kyoto University, Kyoto, Japan, in 1981, 1983, and 1988, respectively. He joined Kyoto University as an instructor in 1984 and was promoted to an associate professor in 1991. He moved to Nagoya University, Nagoya, Japan, in 1994, and promoted to a professor in 1998. He returned to Kyoto University in 2010. His current interests include computer arithmetic, hardware algorithms, and logic design. He received Japan IBM Science Award and Sakai Memorial Award of the Information Processing Society of Japan in 1995, and The Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology of Japan in 2005.