IPSJ Transactions on System LSI Design Methodology Vol. 4 131–139 (Aug. 2011)

Regular Paper

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 [1]. 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 [2] 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 $[x_{n-1}x_{n-2}\cdots x_1x_0]$ and $[y_{n-1}y_{n-2}\cdots y_1y_0]$, 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 $[x_ix_{i-1}\cdots x_1x_0]$

†1 The Center for Embedded Computing Systems, Graduate School of Information Science, Nagoya University
†2 Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

and Y_i as $[y_i y_{i-1} \cdots y_1 y_0]$ for $0 \le i \le n-1$. 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:

$$\begin{aligned} X \times Y &= \left(\sum_{i=0}^{n-1} 2^i x_i\right) \times \left(\sum_{i=0}^{n-1} 2^i y_i\right) \\ &= \left(2^{n-1} x_{n-1} + X_{n-2}\right) \times \left(2^{n-1} y_{n-1} + Y_{n-2}\right) \\ &= X_{n-2} \times Y_{n-2} + 2^{n-1}(2^{n-1} x_{n-1} y_{n-1} + x_{n-1} Y_{n-2} + y_{n-1} X_{n-2}) \end{aligned}$$

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

$$X \times Y = x_0 y_0 + \sum_{i=1}^{n-1} 2^i \left(2^i x_i y_i + x_i Y_{i-1} + y_i X_{i-1}\right)$$

This equation is used for serial-serial multiplication [3)–5)]. We call $2^i x_i y_i + x_i Y_{i-1} + y_i X_{i-1}$ 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:

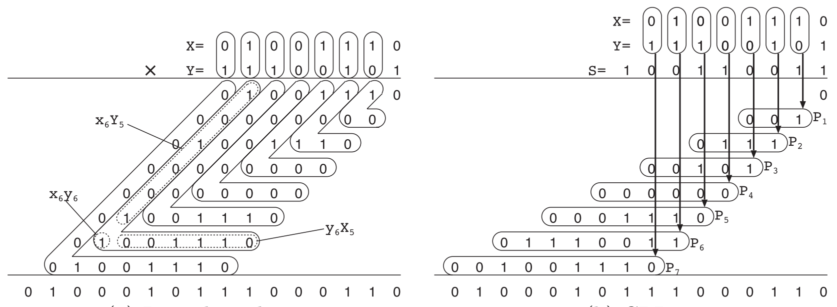
$$P_i = \begin{cases} 0 & \texttt{if } (x_i, y_i) = (0,0) \\ X_{i-1} & \texttt{if } (x_i, y_i) = (0,1) \\ Y_{i-1} & \texttt{if } (x_i, y_i) = (1,0) \\ 2^i + X_{i-1} + Y_{i-1} & \texttt{if } (x_i, y_i) = (1,1) \end{cases}$$

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 $[s_n s_{n-1} s_{n-2} \cdots s_1 s_0]$. We define S_i as $[s_i s_{i-1} \cdots s_1 s_0]$. We can express P_i in an $(i+2)$-bit unsigned binary representation as follows:

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

$\overline{s_i}$ 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

$$X \times Y = x_0 y_0 + \sum_{i=1}^{n-1} 2^i P_i.$$



(a) Partial products (b) CPPs

Fig. 1 An example of 8-bit unsigned multiplication.

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 `[100110011]`. We show CPPs in Fig. 1 (b) and basic partial products in Fig. 1 (a). In Fig. 1 (a), each "*L-shaped*" area shows a set of partial product bits which correspond to a CPP. In Fig. 1 (b), each CPP is shown in a circle. For example, since $(x_3, y_3) = (1,0)$, P_3 is represented as $\texttt{[00}y_2 y_1 y_0\texttt{]=[00101]}$. P_4, P_5 and P_6 are represented as `[000000]`, $\texttt{[00}x_4 x_3 x_2 x_1 x_0\texttt{]=[0001110]}$ and $[s_6 \overline{s_6} s_5 s_4 s_3 s_2 s_1 s_0]\texttt{=[01110011]}$, 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 $i \le n-2$, X_i and Y_i are expressed in unsigned binary representation as $[x_i x_{i-1} \cdots x_1 x_0]$ and $[y_i y_{i-1} \cdots y_1 y_0]$, respectively. The values of them are $\sum_{j=0}^{i} 2^j x_j$ and $\sum_{j=0}^{i} 2^j y_j$, respectively. We define S as the sum of the operands $X+Y$. S is represented in $(n+1)$-bit two's complement binary representation.

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:

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

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 $-S_{n-1} = -2^n + \overline{S_{n-1}} + 1$, where $\overline{S_{n-1}}$ indicates the bitwise inversion of S_{n-1}. We treat $\overline{S_{n-1}}$ as an n-bit unsigned integer. Then $P_{n-1} = -2^n + 2^{n-1} + \overline{S_{n-1}} + 1$. 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:

$$P^*_{n-1} = \begin{cases} [\ 1 \quad 1 \quad 1 \quad 1 \quad \cdots \quad 1 \quad 1\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (0,0) \\ [\ 1 \quad 1 \quad \overline{x_{n-2}} \ \overline{x_{n-3}} \quad \cdots \quad \overline{x_1} \quad \overline{x_0}\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (0,1) \\ [\ 1 \quad 1 \quad \overline{y_{n-2}} \ \overline{y_{n-3}} \quad \cdots \quad \overline{y_1} \quad \overline{y_0}\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (1,0) \\ [s_{n-1} s_{n-1} \overline{s_{n-2}} \ \overline{s_{n-3}} \quad \cdots \quad \overline{s_1} \quad \overline{s_0}\ ] & \texttt{if } (x_{n-1}, y_{n-1}) = (1,1) \end{cases}$$

Finally, signed multiplication is performed as

$$X \times Y = x_0y_0 + \left(\sum_{i=1}^{n-2} 2^i P_i\right) + 2^{n-1}P_{n-1}$$

$$= x_0y_0 + \left(\sum_{i=1}^{n-2} 2^i P_i\right) + 2^{n-1}P^*_{n-1} + 2^{n-1}.$$

The total number of CPP bits is $\frac{1}{2}n^2 + \frac{3}{2}n - 1$. CPPs and x_0y_0 are represented in $\frac{1}{2}n^2 + \frac{3}{2}n$ bits in total.

We show an example of 8-bit signed multiplication in **Fig. 2**. We assume that X is `[01001110]` and Y is `[11100101]`. Both X and Y are represented in two's complement representation. x_0y_0 and CPPs from P_1 to P_6 are the same as the unsigned multiplication shown in Fig. 1. In this example $(x_7, y_7) = (0, 1)$, therefore P^*_7 is represented as $[11\overline{x_6}\ \overline{x_5} \cdots \overline{x_1}\ \overline{x_0}] = [110110001]$.

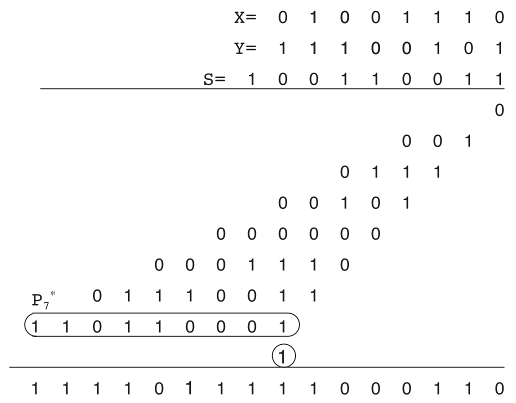


Fig. 2 An example of singed multiplication.

The proposed method performs multiplication and addition simultaneously, i.e., $X \times Y$ and $X + Y$ are obtained simultaneously. Therefore the proposed method can be used more effectively on applications that require both a product and a sum of the same pair of operands.

3. Reduced Area Multipliers Employing the CPPs

In this section, we show a design of a multiplier employing the CPPs. A block diagram of an 8-bit unsigned parallel multiplier employing the CPPs is shown in **Fig. 3**. The multiplier consists of an operand adder, an operand recoder, a CPP generator, a CPP compressor and a final adder. The operand adder is a carry

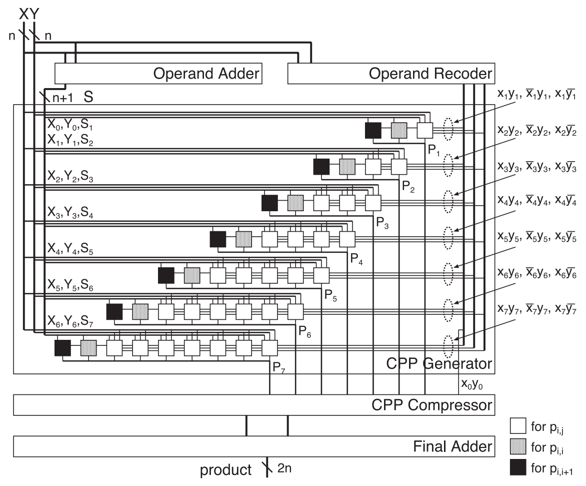


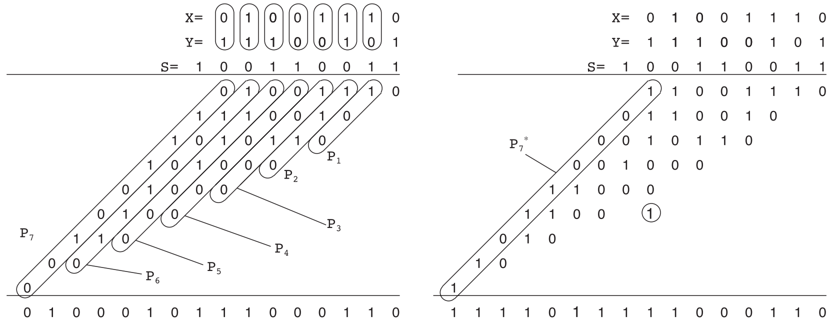
Fig. 3 A block diagram of an 8-bit unsigned multiplier employing the CPPs.

propagate adder for calculating the sum of the operands. The operand recoder generates $x_i \wedge y_i$, $\overline{x_i} \wedge y_i$ and $x_i \wedge \overline{y_i}$ for $0 \leq i \leq n-1$. The CPP generator consists of $\frac{1}{2}n^2 + \frac{3}{2}n - 1$ selector cells. Each selector cell generates a CPP bit from (x_j, y_j, s_j) and $(x_i \wedge y_i, \overline{x_i} \wedge y_i, x_i \wedge \overline{y_i})$. 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:

$$\begin{aligned} p_{i,j} &= ((\overline{x_i} \wedge y_i) \wedge x_j) \vee ((x_i \wedge \overline{y_i}) \wedge y_j) \vee ((x_i \wedge y_i) \wedge s_j) \\ p_{i,i} &= (x_i \wedge y_i) \wedge \overline{s_i} \\ p_{i,i+1} &= (x_i \wedge y_i) \wedge s_i \end{aligned}$$

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 $0 \leq j \leq n-2$:



(a) CPPs for unsigned multiplication (b) CPPs for signed multiplication

Fig. 4 Rearrangement of the CPPs.

$$\begin{aligned} p^*_{n-1,j} &= \overline{((\overline{x_{n-1}} \wedge y_{n-1}) \wedge x_j) \vee ((x_{n-1} \wedge \overline{y_{n-1}}) \wedge y_j) \vee ((x_{n-1} \wedge y_{n-1}) \wedge s_j)} \\ p^*_{n-1,n-1} &= s_{n-1} \vee \overline{(x_{n-1} \wedge y_{n-1})} \\ p^*_{n-1,n} &= p_{n-1,n-1} \end{aligned}$$

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

Table 1 Comparison between horizontal and diagonal placement of CPPs for unsigned multipliers.

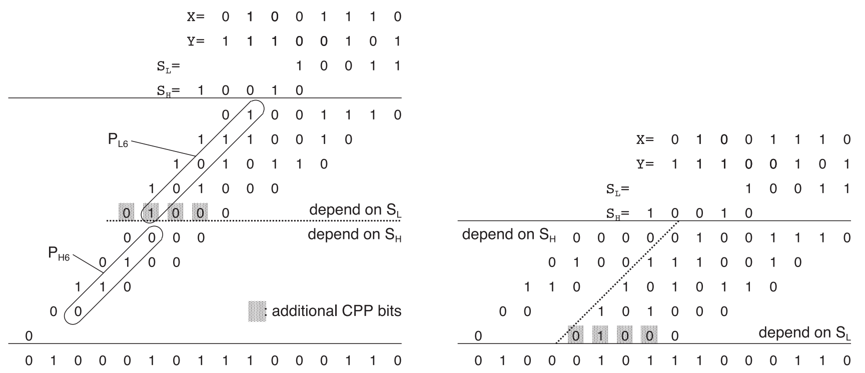
		Rohm 0.18 μm		STARC 90 nm	
		area	delay	area	delay
16-bit	horizontal	20,070.1	13.08	4,592.3	7.04
	diagonal	19,459.4	11.35	4,039.6	5.63
32-bit	horizontal	73,484.7	25.76	16,895.9	14.17
	diagonal	72,165.1	22.20	15,351.0	10.21
64-bit	horizontal	280,449.5	49.99	64,765.3	28.46
	diagonal	279,052.3	43.07	59,750.9	20.89

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 μm 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



(a) CPPs in the case of dividing the addition into two sections (b) CPPs filled up into the $\lfloor \frac{n}{2} \rfloor + 1$ rows

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 : $[x_{\frac{n}{2}-1}x_{\frac{n}{2}-2}\cdots x_1x_0]$

X_H : $[x_{n-1}x_{n-2}\cdots x_{\frac{n}{2}+1}x_{\frac{n}{2}}]$

Y_L : $[y_{\frac{n}{2}-1}y_{\frac{n}{2}-2}\cdots y_1y_0]$

Y_H : $[y_{n-1}y_{n-2}\cdots y_{\frac{n}{2}+1}y_{\frac{n}{2}}]$.

X_L, X_H, Y_L and Y_H have values $\sum_{j=0}^{\frac{1}{2}n-1} 2^j x_j$, $\sum_{j=\frac{1}{2}n}^{n-1} 2^j x_j$, $\sum_{j=0}^{\frac{1}{2}n-1} 2^j y_j$ and $\sum_{j=\frac{1}{2}n}^{n-1} 2^j y_j$, 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 $[l_{\frac{n}{2}}l_{\frac{n}{2}-1}\cdots l_1l_0]$ and $[h_{\frac{n}{2}}h_{\frac{n}{2}-1}\cdots h_1h_0]$, respectively. Note that $S_L + 2^{\frac{n}{2}}S_H = S$. 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 $P_{Li} + 2^{\frac{n}{2}}P_{Hi} = P_i$. 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_{Li}'s, and parallelize CPP compression. The total number of CPP bits increases by $\frac{1}{2}n$ bits due to the acceleration method. Since the acceleration method divides the operand addition, an additional bit $l_{\frac{1}{2}n}$, 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 $l_{\frac{1}{2}n}$. The number of additional CPP bits is $\frac{1}{2}n$. 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 $\lfloor \frac{n}{2} \rfloor + 1$ rows for unsigned multiplication. We show the alignment of CPP bits in Fig. 5 (b). For signed multiplication, the number of rows is $\lfloor \frac{n}{2} \rfloor + 2$. CPP bits are generated in order from the first row to the $\lfloor \frac{n}{2} \rfloor + 1$-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 with that of the array multipliers using radix-4 Booth's method whose number of partial product is $\lfloor \frac{n}{2} \rfloor + 1$.

Note that we can divide the addition into any number of sections at any points. If operand addition is divided into more sections, the calculation becomes faster and the number of the additional CPP bits increases.

5. Evaluation

We designed the multipliers using the cell libraries for Rohm 0.18 μm 5-metal CMOS technology and Semiconductor Technology Academic Research Center (STARC) 90 nm 6-metal CMOS technology. Both libraries are provided by VLSI Design and Education Center (VDEC), the University of Tokyo. We optimized the multipliers with Synopsys Design Compiler. We used Cadence SoC Encounter and Synopsys Astro for the physical design with the 0.18 μm cell library and the 90 nm cell library, respectively. All cells and wires of the multipliers are placed and routed with over 95% core utilization.

We evaluate the multipliers employing the CPPs by comparing their circuit area and delay with those of conventional multipliers. Input widths are 16-bit, 32-bit and 64-bit. We evaluate two constructions of the multipliers with the proposed method.

CPP+array : Multipliers using an array-type CPP compressor. The operand addition is not divided.

CPP+div2+array : Multipliers using an array-type CPP compressor. The operand addition is divided into two sections.

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 μm 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 μm and STARC 90 nm, respectively. These features show the useful-

Table 2 Area (μm^2) and delay (ns) of multipliers under area constraint.

			Rohm 0.18 μm		STARC 90 nm	
			area	delay	area	delay
unsigned	16-bit	array	24,946.8	11.47	5,790.6	6.70
		array+Booth	22,902.3	13.37	4,522.5	5.98
		CPP+array	19,459.4	11.35	4,039.6	5.63
		CPP+div2+array	20,470.9	11.42	4,311.2	6.44
	32-bit	array	103,775.1	23.67	24,127.1	14.09
		array+Booth	84,552.3	22.37	17,362.1	10.84
		CPP+array	72,165.1	22.20	15,351.0	10.21
		CPP+div2+array	74,397.7	22.58	15,911.3	10.70
	64-bit	array	423,104.6	48.00	98,448.4	28.95
		array+Booth	322,622.9	41.93	66,727.6	20.84
		CPP+array	279,052.3	43.07	59,750.9	20.89
		CPP+div2+array	283,645.9	43.41	60,298.5	20.79
signed	16-bit	array	24,753.1	11.61	5,748.2	6.73
		array+Booth	21,618.5	13.08	4,251.6	5.59
		CPP+array	19,615.9	11.40	4,057.2	5.62
		CPP+div2+array	21,652.1	11.42	4,384.6	5.92
	32-bit	array	103,362.3	23.83	24,037.2	14.21
		array+Booth	81,821.4	23.36	16,651.1	10.77
		CPP+array	72,483.0	22.39	15,385.6	10.65
		CPP+div2+array	76,746.4	22.24	16,052.4	10.86
	64-bit	array	422,256.8	48.18	98,264.0	28.90
		array+Booth	317,325.6	41.25	65,437.1	20.25
		CPP+array	279,697.0	43.20	59,750.9	20.94
		CPP+div2+array	288,414.8	43.61	61,196.7	22.24

Table 3 Area (μm^2) and delay (ns) of multipliers under both area and delay constraint.

			Rohm 0.18 μm		STARC 90 nm	
			area	delay	area	delay
unsigned	16-bit	array	27,478.9	8.15	12,954.8	2.71
		array+Booth	35,103.0	5.72	11,379.9	1.85
		CPP+array	24,423.4	7.42	10,245.3	2.18
		CPP+div2+array	27,122.5	5.88	10,358.2	1.95
	32-bit	array	108,445.4	15.73	51,755.8	5.45
		array+Booth	105,590.0	9.80	35,140.3	3.55
		CPP+array	83,170.4	13.49	28,034.9	4.48
		CPP+div2+array	89,573.8	10.30	33,225.3	3.63
	64-bit	array	432,733.4	31.22	192,510.3	11.43
		array+Booth	383,888.0	18.14	115,831.3	6.73
		CPP+array	301,362.8	26.54	102,784.1	8.99
		CPP+div2+array	319,258.6	18.80	112,801.5	7.13
signed	16-bit	array	27,262.8	8.29	12,859.6	2.61
		array+Booth	35,041.5	6.42	12,294.4	1.90
		CPP+array	24,865.6	7.14	9,796.6	2.26
		CPP+div2+array	27,214.4	6.37	10,790.7	2.08
	32-bit	array	108,288.2	15.88	51,882.8	5.44
		array+Booth	107,904.9	10.73	33,841.3	3.53
		CPP+array	86,848.3	13.13	30,380.3	4.35
		CPP+div2+array	91,805.6	10.48	32,262.9	3.70
	64-bit	array	432,169.9	30.53	186,657.3	11.55
		array+Booth	384,976.5	19.35	124,665.4	6.63
		CPP+array	309,527.1	26.01	116,498.1	9.34
		CPP+div2+array	320,856.1	19.91	115,926.6	7.48

ness of the proposed method as the small area multipliers and that the multipliers using CPPs can be an alternative to the array multipliers using radix-4 Booth's method.

In Section 3 and Section 4, we pointed out that the delay of **CPP+array** and **CPP+div2+array** are expected to be comparable with that of **array** and **array+Booth**, respectively. Here, we can see it from the experimental results. The delays of **array** and **CPP+array** indicate very close values. Differences of the delay between **CPP+div2+array** and **array+Booth** are almost within 10%.

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 multpliers

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**. **Wallce** 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

Table 4 Area (μm^2) and delay (ns) of multipliers using Wallace tree.

(a) Under area constraint and no delay constraint

			Rohm 0.18 μm		STARC 90 nm	
			area	delay	area	delay
unsigned	16-bit	Wallace	25,085.9	10.18	5,823.8	4.69
		Wallace+Booth	22,814.1	13.05	4,743.7	5.94
		CPP+Wallace	19,650.0	11.44	4,292.9	5.50
	32-bit	Wallace	103,911.6	20.73	24,160.9	9.27
		Wallace+Booth	84,359.8	24.59	17,320.2	10.37
		CPP+Wallace	72,357.5	22.66	15,981.8	10.38
	64-bit	Wallace	423,241.2	42.59	98,482.8	18.85
		Wallace+Booth	322,250.3	46.03	67,516.7	19.52
		CPP+Wallace	279,246.2	44.34	61,542.4	21.06
signed	16-bit	Wallace	24,892.2	10.30	5,780.0	4.68
		Wallace+Booth	20,806.7	12.69	4,366.3	5.18
		CPP+Wallace	19,765.6	11.40	4,110.1	5.27
	32-bit	Wallace	103,502.0	20.62	24,069.5	9.37
		Wallace+Booth	80,054.6	23.80	17,244.9	10.09
		CPP+Wallace	72,632.6	22.82	15,946.6	10.36
	64-bit	Wallace	422,398.8	42.28	98,295.3	18.79
		Wallace+Booth	313,588.5	46.51	65,279.3	19.73
		CPP+Wallace	279,849.0	44.70	61,542.4	20.84

(b) Under both area and delay constraint

			Rohm 0.18 μm		STARC 90 nm	
			area	delay	area	delay
unsigned	16-bit	Wallace	32,432.9	4.30	12,356.5	1.45
		Wallace+Booth	38,168.0	4.18	9,824.1	1.46
		CPP+Wallace	29,893.8	4.96	7,413.0	1.83
	32-bit	Wallace	121,921.2	5.45	41,037.7	2.41
		Wallace+Booth	119,321.4	5.50	34,931.4	2.26
		CPP+Wallace	98,790.6	7.04	27,151.5	2.76
	64-bit	Wallace	468,251.3	7.35	150,714.8	4.02
		Wallace+Booth	411,641.9	7.48	112,519.9	3.58
		CPP+Wallace	345,247.5	10.32	105,950.1	4.06
signed	16-bit	Wallace	32,674.6	4.07	10,386.4	1.43
		Wallace+Booth	41,975.1	4.04	10,877.5	1.65
		CPP+Wallace	31,506.8	4.72	7,878.0	1.89
	32-bit	Wallace	123,510.4	5.64	38,745.9	2.27
		Wallace+Booth	120,589.5	5.40	32,112.6	2.19
		CPP+Wallace	101,488.9	6.81	25,293.6	2.71
	64-bit	Wallace	470,246.1	7.73	151,475.4	3.59
		Wallace+Booth	413,382.9	7.65	107,321.8	3.46
		CPP+Wallace	341,909.2	9.72	99,666.0	4.20

performs addition and multiplication simultaneously and therefore can be used more efficiently for applications where the sum and the product are required simultaneously.

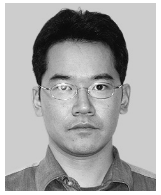
Acknowledgments This work is supported by Core Research for Evolutional Science and Technology (CREST) from Japan Science and Technology Agency and VLSI Design and Education Center (VDEC), the University of Tokyo in collaboration with Synopsys, Inc., Cadence Design Systems, Inc., and Rohm Corporation.

References

1) Ercegovac, M.D. and Lang, T.: *Digital Arithmetic*, Morgan Kaufmann Publishers (2003).
2) MacSorley, O.: High-Speed Arithmetic in Binary Computers, *Proc. IRE*, Vol.49, pp.67–91 (1961).
3) Chen, I.-N. and Willoner, R.: An 0() Parallel Multiplier with Bit-Sequential Input and Output, *IEEE Trans. Comput.*, Vol.28, No.10, pp.721–727 (1979).
4) Strader, N.R. and Rhyne, V.T.: A Canonical Bit-Sequential Multiplier, *IEEE Trans. Comput.*, Vol.31, No.8, pp.791–795 (1982).
5) Gnanasekaran, R.: On a Bit-Serial Input and Bit-Serial Output Multiplier, *IEEE Trans. Comput.*, Vol.32, No.9, pp.878–880 (1983).
6) Wallace, C.: A Suggestion for a Fast Multiplier, *IEEE Trans. Electronic Computers*, Vol.EC-13, pp.14–17 (1964).

(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.