IPSJ Transactions on System LSI Design Methodology Vol. 4 60–69 (Feb. 2011)

Regular Paper

A Fast Selector-Based Subtract-Multiplication Unit and Its Application to Butterfly Unit

Youhei Tsukamoto,[†1] Masao Yanagisawa,[†2]
Tatsuo Ohtsuki[†1] and Nozomu Togawa[†1]

Large-scale network and multimedia application LSIs include application specific arithmetic units. A multiply-accumulator unit or a MAC unit which is one of these optimized units arranges partial products and decreases carry propagations. However, there is no method similar to MAC to execute "subtract-multiplication". In this paper, we propose a high-speed subtract-multiplication unit that decreases latency of a subtract operation by bit-level transformation using selector logics. By using bit-level transformation, its partial products are calculated directly. The proposed subtract-multiplication units can be applied to any types of systems using subtract-multiplications and a butterfly operation in FFT is one of their suitable applications. We apply them effectively to Radix-2 butterfly units and Radix-4 butterfly units. Experimental results show that our proposed operation units using selector logics improves the performance by up to 13.92%, compared to a conventional approach.

1. Introductioan

Large-scale network and multimedia application LSIs include many application-specific arithmetic units. Fast and efficient arithmetic circuits included in such application LSIs are strongly required in scientific computations as well as consumer products.

A multiply-accumulate unit (MAC unit) is one of these optimized units [1),2)] which executes a multiply-accumulate operation. It is expressed by $a \times b + c$, where a, b, and c are input variables and very widely used in digital signal processing. A MAC unit effectively performs this operation, where its execution time is almost the same as the time to just multiply the two variables. This can be done by combining partial product generation and addition.

A subtract-multiplication operation is another application-specific operation expressed by $(a - b) \times c$, where a, b, and c are input variables. This operation is commonly used in a butterfly operation in the fast Fourier transform (FFT) but quite different from multiply-accumulation, where multiplication cannot be started until subtraction is finished. It can be a big problem since the total carry propagation delay of a subtract-multiplication unit must be the sum of subtraction and multiplication. As far as we know, there are no existing works for optimizing a subtract-multiplication operation.

To solve this problem, we propose a method that generates partial products for a subtract-multiplication operation very fast by utilizing "selector logics." A selector logic is expressed by $xz + y\overline{z}$, where x, y, and z is a 1-bit value. Since there is no carry-out in a selector logic, we can significantly decrease the carry propagation delay by using them. In order to apply selector logics to a subtract-multiplication operation, we first represent each n-bit input variable to be $-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$, where a_i is i-th bit of each input variable. Second, we perform the subtract-multiplication operation by using these bit-level representations. Since we find out the form of $a_i c_i + b_i \overline{c_i}$ in each bit, we can implement it by using a selector. Finally, we can design a subtract-multiplication unit effectively using selector logics.

Our proposed subtract-multiplication units can be applied to even any types of systems using subtract-multiplications. A butterfly operation which is used in FFT is a suitable application using them. Then we design a Radix-2 butterfly unit as well as a Radix-4 butterfly unit using our proposed subtract-multiplication units. Experimental results show that our proposed units improves the performance by a maximum of 13.92%, compared to a conventional one.

This paper is organized as follows: Section 2 proposes a subtract-multiplication unit utilizing selector logics; Section 3 proposes Radix-2 and Radix-4 butterfly units based on the proposed subtract-multiplication unit design; Section 4 demonstrates experimental results; Section 5 gives concluding remarks.

†1 Department of Computer Science and Engineering, Waseda University
†2 Department of Electronic and Photonic Systems, Waseda University

2. Subtract-multiplication Operation Based on Selector Logics

Bit-level transformation is one of the methods to optimize design of arithmetic unit [3]. In this section, we propose a bit-level transformation technique such that a selector logic can be applied to it.

2.1 Selector Logics

Let x, y, z and w be 1-bit variables. A selector logic is represented by an expression below:

$$w = xz + y\overline{z} \tag{1}$$

The output value w is set to be x or y by using the selecting signal z. This expression can be implemented by two logical ANDs and a logical OR. It can be also implemented by a "selector" unit where its area cost and delay are almost equal to those of two logical ANDs and a logical OR.

The *output range* of the selector logic is expressed by

$$\max|xz + y\overline{z}| - \min|xz + y\overline{z}| \le 1. \tag{2}$$

In other words, it generates no carry-out, since the output of the selector logic becomes 0 or 1. Generally speaking, any arithmetic operation which has two or three 1-bit inputs and whose output range is greater than one generates a carry-out. For example, a full adder and a half adder used very often in arithmetic units must generate a sum and a carry-out. This means that, if we can perform a bit-level transformation for a given arithmetic operation such that a selector logic can be applied, carry propagation can be much reduced and thus its resultant arithmetic unit will be much faster.

2.2 Bit-level Transformation of Subtract-multiplication Operation Using Selector Logics

Now let us pick up a subtract-multiplication $(a - b) \times c$, where a, b, and c are n-bit variables, and perform a bit-level transformation to it such that a selector logic can be applied to it. The input variables a, b, and c are represented by

$$a = [a_{n-1}a_{n-2}\cdots a_1a_0]_b = -a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i, \tag{3}$$

$$b = [b_{n-1}b_{n-2}\cdots b_1b_0]_b = -b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i, \tag{4}$$

$$c = [c_{n-1}c_{n-2}\cdots c_1c_0]_b = -c_{n-1}2^{n-1} + \sum_{i=0}^{n-2} c_i 2^i. \tag{5}$$

where a_k, b_k, and c_k $(k = 0, \cdots, n-1)$ are 1-bit variables which represent k-th digit in a, b, and c, respectively. Since we use here a 2's complementary form, $-c$ is expressed by

$$-c = -\overline{c_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{c_i} 2^i + 1. \tag{6}$$

Using Eqs. (3)–(6), we can compute $(a - b) \times c$ as:

$$\begin{aligned}
&(a-b)\times c = a\times c + b\times(-c)\\
&= \left(-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i\right) \times \left(-c_{n-1}2^{n-1} + \sum_{i=0}^{n-2} c_i 2^i\right)\\
&\quad + \left(-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i\right) \times \left(-\overline{c_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{c_i} 2^i + 1\right)\\
&= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2} (a_i c_j + b_i \overline{c_j}) 2^{i+j} + a_{n-1}2^{n-1}\left(-\sum_{i=0}^{n-2} c_i 2^i\right)\\
&\quad + c_{n-1}2^{n-1}\left(-\sum_{i=0}^{n-2} a_i 2^i\right) + b_{n-1}2^{n-1}\left(-\sum_{i=0}^{n-2} \overline{c_i} 2^i\right)\\
&\quad + \overline{c_{n-1}}2^{n-1}\left(-\sum_{i=0}^{n-2} b_i 2^i\right) - b_{n-1}2^{n-1} + \sum_{i=1}^{n-2} b_i 2^i\\
&\quad + (a_{n-1}c_{n-1} + b_{n-1}\overline{c_{n-1}})2^{2n-2}.
\end{aligned} \tag{7}$$

Using Eq. (6), we have:

$$-\sum_{i=0}^{n-2} c_i 2^i = -1\cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{c_i} 2^i + 1$$

$$-\sum_{i=0}^{n-2} a_i 2^i = -1\cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{a_i} 2^i + 1$$

$$-\sum_{i=0}^{n-2} b_i 2^i = -1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{b_i} 2^i + 1$$
$$-\sum_{i=0}^{n-2} \overline{c}_i 2^i = -1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} c_i 2^i + 1 \quad (8)$$

Based on Eqs. (7) and (8), we finally have:

$$\begin{aligned}\text{Eq. (7)} = &\sum_{i=0}^{n-2}\sum_{j=0}^{n-2}(a_i c_j + b_i \overline{c_j})2^{i+j} + \sum_{i=0}^{n-2} b_i 2^i \\ &+ \sum_{i=0}^{n-2}\left\{(b_{n-1}c_i + a_{n-1}\overline{c_i}) + (\overline{a_i}c_{n-1} + \overline{b_i}\overline{c_{n-1}})\right\} 2^{i+n-1} \\ &+ (a_{n-1}c_{n-1} + b_{n-1}\overline{c_{n-1}} - a_{n-1} - c_{n-1} - b_{n-1} - \overline{c_{n-1}})2^{2n-2} \\ &+ a_{n-1}2^{n-1} + (c_{n-1} + \overline{c_{n-1}})2^{n-1}. \quad (9)\end{aligned}$$

Assume that we sum up the generated partial products above using the Wallace tree [4] and fast carry-propagate adder (CPA) [5]. If the number of inputs to the Wallace tree is increased, the area cost and delay of its circuit will also be increased. The key point here is *how to reduce the number of inputs to the Wallace tree.*

How to Reduce Negative Terms:

Let us focus on the terms in Eq. (9) whose coefficient is 2^{2n-2}. The two positive terms, $a_{n-1}c_{n-1}$ and $b_{n-1}\overline{c_{n-1}}$, are assigned to the two 1-bit inputs to the 2^{2n-2}-th digit of the Wallace tree. But the remaining four negative terms are assigned to the four 1-bit inputs to the 2^{2n-2}-th digit and also to the 2^{2n-1}-th digit of the Wallace tree, because we have to consider the "sign bits." Its overhead may be large. Thus we perform a bit-level transformation to the term in Eq. (9) whose coefficient is 2^{2n-2} as follows:

$$\begin{aligned}&(a_{n-1}c_{n-1} + b_{n-1}\overline{c_{n-1}} - a_{n-1} - c_{n-1} - b_{n-1} - \overline{c_{n-1}}) \\ =\ &(a_{n-1}c_{n-1} - c_{n-1} - a_{n-1} + 1) - 1 + (b_{n-1}\overline{c_{n-1}} - b_{n-1} - \overline{c_{n-1}} + 1) - 1 \\ =\ &(\overline{a_{n-1}}\ \overline{c_{n-1}}) + (\overline{b_{n-1}}c_{n-1}) - 2. \quad (10)\end{aligned}$$

Assigning Eq. (10) to Eq. (9) leads to:

$$\begin{aligned}\text{Eq. (7)} = &\sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\underline{(a_i c_j + b_i \overline{c_j})}2^{i+j} \quad (11)\\ &+\{\underline{(\overline{a_{n-1}}\ \overline{c_{n-1}}) + (\overline{b_{n-1}}c_{n-1})}\}2^{2n-2} \quad (12)\\ &+\sum_{i=0}^{n-2}\left\{\underline{(b_{n-1}c_i + a_{n-1}\overline{c_i})} + \underline{(\overline{a_i}c_{n-1} + \overline{b_i}\overline{c_{n-1}})}\right\} 2^{i+n-1} \quad (13)\\ &+\sum_{i=0}^{n-2} b_i 2^i \quad (14)\\ &+a_{n-1}2^{n-1} + 2^{n-1} \quad (15)\\ &-2^{2n-1}. \quad (16)\end{aligned}$$

Since Eqs. (11)–(16) have only one negative terms, the input bits of the Wallace tree become $(2n^2 - 4n + 2)$ bits for Eq. (11), 2 bits for Eq. (12), $4n - 4$ bits for Eq. (13), $n - 1$ bits for Eq. (14), 2 bits for Eq. (15), and 1 bits for Eq. (16). The total number of input bits becomes $(2n^2 + n + 2)$.

How to Reduce Partial Products Using Selector Logics:

Next let us focus on the underlined terms in Eqs. (11), (12), and (13). Since they have a form which is completely the same as the selector logic in Eq. (1), they can be implemented by selectors. A selector logic directly computes each of the underlined terms in Eqs. (11), (12) and (13), and then the total number of inputs to the Wallace tree can be decreased to $(n^2 + n + 2)$ if we use selector logics to pre-compute them. Since selector logics generate no carry-outs, this pre-computation can be done very fast.

Using these bit-level transformation and selector logics, carry propagations in both subtraction and multiplication are combined and then the subtract-multiplication operation can be designed by a very fast circuit compared to conventional ones, where subtraction and multiplication are separately implemented and thus carry propagation is required for each of subtraction and multiplication.

Figure 1 shows an example of partial products generated from bit-level transformation with $n = 8$. "Digit" is shown in the first row and partial products is shown in the subsequent rows. For example, five partial products: a_1c_0, $b_1\overline{c_0}$, a_0c_1, $b_0\overline{c_1}$, and b_1 are generated at 2^1-th digit. But $(a_1c_0, b_1\overline{c_0})$ and $(a_0c_1, b_0\overline{c_1})$ are compressed from 4 bits to 2 bits by pre-computing them us-

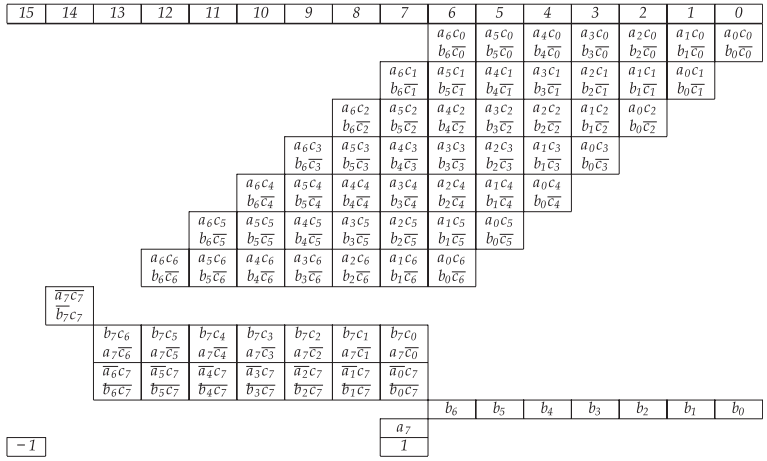


Fig. 1 Partial products generated for an 8-bit subtract-multiplication operation using selector logics. Each box shows "a partial product" to be added up.

ing two selectors. Consequently total number of partial products at 2^1-th digit is decreased from five to just three.

3. Selector-Based Butterfly Unit

A Radix-2 or Radix-4 butterfly operation is widely used in fast fourier transform (FFT) processing, whose input is two complex numbers or four complex numbers [6)–9)]. The speed of FFT processing is much limited by subtract-multiplication operations in butterfly operation. How to implement a Radix-2 and Radix-4 operations is an important key in speeding-up FFT processing.

In this section, we propose a novel Radix-2 or Radix-4 butterfly unit using the subtract-multiplication unit proposed in Section 2. In this design, the butterfly unit will be faster than the one using a naive subtract-multiplication design.

3.1 Radix-2 Butterfly Operation

The discrete fourier transform (DFT) of length N is expressed as follows:

$$X(k) = \sum_{s=0}^{N-1} x(s) W_N^{ks} \quad \left(W_N = e^{-j\frac{2\pi}{N}} \right), \tag{17}$$

where $x(s)$ is an input signal, $X(k)$ is an output signal, and $k = 0, 1, \cdots, (N-1)$. Using the Radix-2 based decimation-in-frequency algorithm [10)], Eq. (17) is decomposed into:

$$X(2m) = \sum_{s=0}^{N/2-1} \left\{ x(s) + x(s+\frac{N}{2}) \right\} W_{N/2}^{ms} \tag{18}$$

$$X(2m+1) = \sum_{s=0}^{N/2-1} \left\{ x(s) - x(s+\frac{N}{2}) \right\} W_N^s W_{N/2}^{ms} \tag{19}$$

where $m = 0, \cdots, \frac{N}{2} - 1$. Let us define $x_e(s)$ and $x_o(s)$ to be:

$$x_e(s) = x(s) + x(s+\frac{N}{2}) \tag{20}$$

$$x_o(s) = \left\{ x(s) - x(s+\frac{N}{2}) \right\} W_N^s. \tag{21}$$

Then, Eqs. (18)–(19) can be considered to be DFT of length $N/2$, whose inputs are $x_e(s)$ and $x_o(s)$. This means that Eq. (17) can be decomposed into two DFT operations. By recursively applying this decomposition, we can finally have DFT of length 2 where multiplication is executed $N \log N$ times. That is more effective than Eq. (17) where multiplication is executed N^2 times. This is a basic concept of FFT.

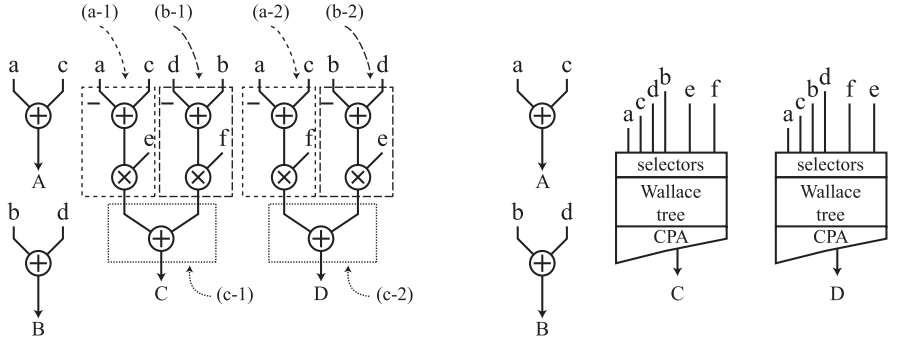
Equations (20)–(21) are calculated in parallel by most FFT hardware architectures. They are called *the Radix-2 butterfly operation* described as shown in **Fig. 2** (i). In this figure, $X(2m)$, $X(2m+1)$, $x(s)$ and $x(s+\frac{N}{2})$ in Eqs. (20)–(21) are represented by:

$$\begin{aligned} &x_e(s) = A + Bj, \qquad x(s) = a + bj, \\ &x(s+\tfrac{N}{2}) = c + dj, \\ &x_o(s) = C + Dj, \qquad W_N^s = e + fj. \end{aligned} \tag{22}$$

where A, B, C, D, a, b, c, d, e, and f show real variables and j shows "an imaginary unit." a, b, c, d, e, and f can also be represented by the form of bit-level variables as in Eqs. (3)–(5).

3.2 Radix-2 Butterfly Unit Based on Selector Logics

Let us focus on the operations surrounded by the dashed lines (a-1) and (b-1)



(i)Radix-2 butterfly operation. (ii)Selector-based Radix-2 butterfly operation.

Fig. 2 Radix-2 butterfly operation.

in Fig. 2 (i). Since these nodes show subtract-multiplication operations, we can implement them using the selector-based subtract-multiplication unit shown in Section 2, which can lead to low latency compared with conventional implementations.

However, if we apply the selector-based subtract-multiplication units to (a-1) and (b-1) in Fig. 2 (i) separately, we require carry propagations for (a-1), (b-1), and (c-1). In order to avoid this situation, we will perform bit-level transformation for both (a-1) and (b-1) simultaneously so that all the partial products from (a-1) and (b-1) can be inputted into a single Wallace tree.

Based on this discussion, the Radix-2 butterfly operation can be transformed into:

$$
\begin{aligned}
C &= (a-c)e + (d-b)f \\
&= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}(a_i e_j + c_i\overline{e_j})2^{i+j} + \sum_{i=0}^{n-2} c_i 2^i + \sum_{i=0}^{n-2}\{(c_{n-1}e_i + a_{n-1}\overline{e_i}) \\
&\quad + (\overline{a_i}e_{n-1} + \overline{c_i e_{n-1}})\}2^{i+n-1} \\
&\quad + \{(\overline{c_{n-1}}e_{n-1}) + (\overline{a_{n-1}e_{n-1}})\}2^{n-2} - 2^{2n-1} + a_{n-1}2^{n-1} + 2^{n-1} \\
&\quad + \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}(d_i f_j + b_i\overline{f_j})2^{i+j} \\
&\quad + \sum_{i=0}^{n-2} b_i 2^i + \sum_{i=0}^{n-2}\left\{(b_{n-1}f_i + d_{n-1}\overline{f_i}) + (\overline{d_i}f_{n-1} + \overline{b_i f_{n-1}})\right\}2^{i+n-1} \\
&\quad + \{(\overline{b_{n-1}}f_{n-1}) + (\overline{d_{n-1}f_{n-1}})\}2^{n-2} - 2^{2n-1} + d_{n-1}2^{n-1} + 2^{n-1} \\
&= \sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\{(a_i e_j + c_i\overline{e_j}) + (d_i f_j + b_i\overline{f_j})\}2^{i+j} + \sum_{i=0}^{n-2}(c_i + b_i)2^i \\
&\quad + \sum_{i=0}^{n-2}\{(c_{n-1}e_i + a_{n-1}\overline{e_i}) \\
&\quad + (\overline{a_i}e_{n-1} + \overline{c_i}\ \overline{e_{n-1}}) + (b_{n-1}f_i + d_{n-1}\overline{f_i}) + (\overline{d_i}f_{n-1} + \overline{b_i}\ \overline{f_{n-1}})\}2^{i+n-1} \\
&\quad + \{(\overline{c_{n-1}}e_{n-1}) + (\overline{a_{n-1}}\ \overline{e_{n-1}}) + (\overline{b_{n-1}}f_{n-1}) + (\overline{d_{n-1}}\ \overline{f_{n-1}})\}2^{2n-2} \\
&\quad - 2^{2n} + (a_{n-1} + d_{n-1})2^{n-1} + 2^n. \qquad (23)
\end{aligned}
$$

Equation (23) generates many partial products but they are added up by using a single Wallace tree. The last stage of the Wallace tree can use a fast CPA. Similarly, the output D which consists of operations surrounded by the dashed lines (a-2), (b-2), and (c-2) can be computed by using selector logics. The block diagram of this Radix-2 butterfly unit design is shown in Fig. 2 (ii).

3.3 Radix-4 Butterfly Operation

Equation (17) can also be expressed as follows:

$$
\begin{aligned}
X(4m) &= \sum_{s=0}^{N/4-1}\left\{x(s) + x(s+\frac{N}{4}) + x(s+\frac{2N}{4}) + x(s+\frac{3N}{4})\right\}W_{N/4}^{ms} \\
X(4m+1) &= \sum_{s=0}^{N/4-1}\left\{x(s) - jx(s+\frac{N}{4}) - x(s+\frac{2N}{4}) + jx(s+\frac{3N}{4})\right\}W_{N/4}^{ms}W_N^s \\
X(4m+2) &= \sum_{s=0}^{N/4-1}\left\{x(s) - x(s+\frac{N}{4}) + x(s+\frac{2N}{4}) - x(s+\frac{3N}{4})\right\}W_{N/4}^{ms}W_N^{2s} \\
X(4m+3) &= \sum_{s=0}^{N/4-1}\left\{x(s) + jx(s+\frac{N}{4}) - x(s+\frac{2N}{4}) - jx(s+\frac{3N}{4})\right\}W_{N/4}^{ms}W_N^{3s}
\end{aligned} \qquad (24)
$$

Each of $X(4m)$, $X(4m+1)$, $X(4m+2)$, and $X(4m+3)$ $(m = 0, \cdots, (N/4)-1)$ is DFT of length $N/4$. As in the discussion in Section 3.1, we can finally have

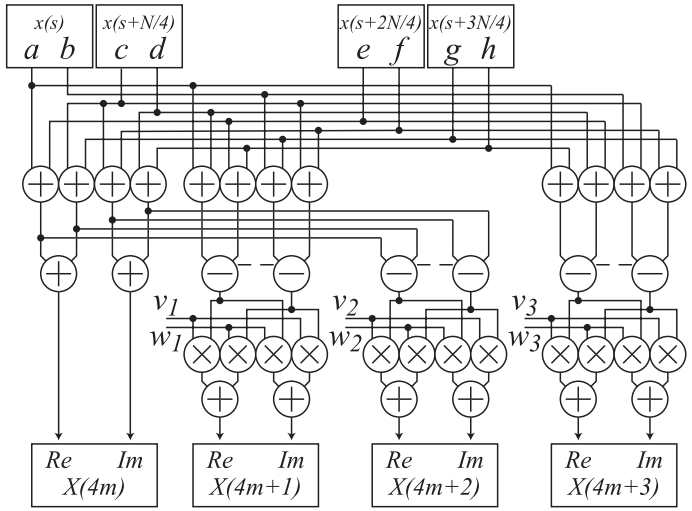


Fig. 3 Radix-4 butterfly operation.

DFT of length 4 if we apply this decomposition to Eq. (17) recursively.

Equation (24) are calculated in parallel by most FFT hardware architectures. They are called *the Radix-4 butterfly operation* described as shown in **Fig. 3**. In this figure, $X(s)$, $X(s+\frac{N}{4})$, $x(s+\frac{2N}{4})$, $x(s+\frac{3N}{4})$, W_N^s, W_N^{2s}, and W_N^{3s} are represented by

$$\begin{array}{ll} x(s) = a + bj, & x(s+\frac{N}{4}) = c + dj, \\ x(s+\frac{2N}{4}) = e + fj, & x(s+\frac{3N}{4}) = g + hj, \\ W_N^s = v_1 + w_1 j, & W_N^{2s} = v_2 + w_2 j, \\ W_N^{3s} = v_3 + w_3 j. & \end{array} \tag{25}$$

where $a, b, \cdots, h$, v_1, v_2, v_3 w_1, w_2, and w_3 shows real variables and j shows "an imaginary unit." $a, b, \cdots, h$, v_1, v_2, v_3 w_1, w_2, and w_3 can also be represented by the form of bit-level variables as in Eqs. (3)–(5).

3.4 Radix-4 Butterfly Unit Based on Selector Logic

Let us focus on the operations surrounded by the dashed line in **Fig. 4**. Since they have the same structure with the Radix-2 butterfly operation shown in Section 3.2, they can be implemented by using the selector-based operation unit.

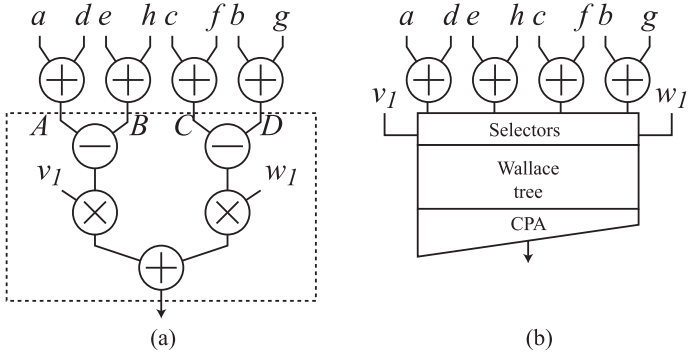


Fig. 4 Selector-based Radix-4 butterfly operation. (a) Original basic data flow. (b) Selector-based operation.

First, let A be $(a+d)$, B be $(e+h)$, C be $(c+f)$, D be $(b+g)$, E be v_1, and F be w_1 (see Fig. 4). Then we have the form of $Re\ X(4m+1) = (A-B) \times E + (C-D) \times F$, which is completely the same as Eq. (23). Then $Re\ X(4m+1)$ is transformed into:

$$\begin{aligned} &Re\ X(4m+1) = (A-B) \times E + (C-D) \times F \\ &= \sum_{i=0}^{n-1} \sum_{j=0}^{n-2} \{(A_i E_j + C_i \overline{E_j}) + (D_i F_j + B_i \overline{F_j})\} 2^{i+j} + \sum_{i=0}^{n-1} (C_i + B_i) 2^i \\ &+ \sum_{i=0}^{n-2} \{(C_n E_i + A_n \overline{E_i}) + (B_n F_i + D_n \overline{F_i})\} 2^{i+n} \\ &+ \sum_{i=0}^{n-1} \{(\overline{A_i} E_{n-1} + \overline{C_i}\ \overline{E_{n-1}}) + (\overline{D_i} F_{n-1} + \overline{B_i}\ \overline{F_{n-1}})\} 2^{i+n-1} \\ &+ \{(\overline{C_{n-1}} E_{n-1}) + (\overline{A_{n-1}}\ \overline{E_{n-1}}) + (\overline{B_{n-1}} F_{n-1}) + (\overline{D_{n-1}}\ \overline{F_{n-1}})\} 2^{2n-1} \\ &- 2^{2n+1} + (A_{n-1} + D_{n-1}) 2^n + 2^n. \end{aligned} \tag{26}$$

Equation (26) is implemented by selector logics to generate partial products and Wallace tree to add up them in the same way as the case of Eq. (23). Figure 4 (a) describes the block diagram of this computation. Eq. (26) refers only the nodes surrounded by the dashed line in Fig. 4 (a). Then, we call it *Radix-4 partial transform.*

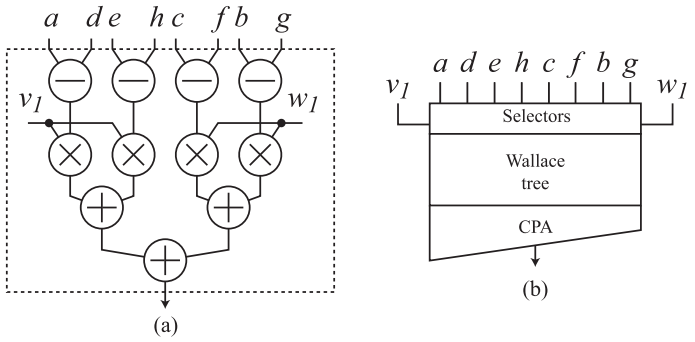


Fig. 5 Selector-based radix-4 butterfly operation by complete transform. (a) Modified data flow. (b) Selector-based operation

Equation (26) may be effective but its total delay time becomes the sum of addition and subtract-multiplication as shown in Fig. 4 (b). This is because the four additions in Fig. 4 (b) at the first stage still remain. Thus we modify the basic data flow in Fig. 4 (a) so that the subtractions are processed at the first stage. **Figure 5** (a) shows the modified data flow. As in Fig. 5 (a), four subtract operations are moved upward and then we can see four subtract-multiplication operations there. This means that four subtract-multiplication operations can be done simultaneously and, after that, their partial products can be summed up.

Based on the above discussion, $Re\ X(4m+1)$ can be transformed according to the data flow in Fig. 5 (a) as follows:

$$
\begin{aligned}
&Re\ X(4m+1) = \\
&\sum_{i=0}^{n-2}\sum_{j=0}^{n-2}\{(a_iV_j + e_i\overline{V_j}) + (d_iV_j + h_i\overline{V_j}) + (b_iW_j + c_i\overline{W_j}) + (g_iW_j + f_i\overline{W_j})\}2^{i+j} \\
&+ \sum_{i=0}^{n-2}\{(e_{n-1}V_i + a_{n-1}\overline{V_i}) + (\overline{a_i}V_{n-1} + \overline{e_i}\overline{V_{n-1}}) \\
&+ (h_{n-1}V_i + d_{n-1}\overline{V_i}) + (\overline{d_i}V_{n-1} + \overline{h_i}\overline{V_{n-1}}) + (c_{n-1}W_i + b_{n-1}\overline{W_i}) \\
&+ (\overline{b_i}W_{n-1} + \overline{c_i}\overline{W_{n-1}}) \\
&+ (f_{n-1}W_i + g_{n-1}\overline{W_i}) + (\overline{g_i}W_{n-1} + \overline{f_i}\ \overline{W_{n-1}})\}2^{i+n-1} \\
&+ \sum_{i=0}^{n-2}(e_i + h_i + c_i + f_i)2^i - 2^{2n+1} \\
&+ \{(\overline{e_{n-1}}V_{n-1} + \overline{a_{n-1}}\overline{V_{n-1}}) + (\overline{h_{n-1}}V_{n-1} + \overline{d_{n-1}}\ \overline{V_{n-1}}) \\
&+ (\overline{c_{n-1}}W_{n-1} + \overline{b_{n-1}}\ \overline{W_{n-1}}) \\
&+ (\overline{f_{n-1}}W_{n-1} + \overline{g_{n-1}}\overline{W_{n-1}})\}2^{2n-2} + 2^{n+1} \\
&+ (a_{n-1} + d_{n-1} + b_{n-1} + g_{n-1})2^{n-1}. \qquad (27)
\end{aligned}
$$

Equation (27) is also implemented by selector logics to generate partial products and Wallace tree to add up them. Since Eq. (27) transforms all the nodes in the basic data flow in a Radix-4 operation, we call it *Radix-4 complete transorm*. Figure 5 (b) describes the block diagram of this computation.

In Fig. 5 (b), we do not need additions at the first stage unlike the one in Fig. 4 (b), since we generate all the partial products directly. Although the number of partial products increases to the double in this computation, it can run faster than configuration as shown in Fig. 4.

As in Fig. 3, $Re\ X(4m+1), Im\ X(4m+1), Re\ X(4m+2), Im\ X(4m+2), Re\ X(4m+3)$ and $Im\ X(4m+3)$ have the same operation structure and can be calculated by the same data flow as shown in Fig. 4 (a), which shows the case of $Re\ X(4m+1)$.

4. Experimental Results

We have compared our selector-based subtract-multiplication units with the ones using three conventional methods:

Method 1 (arithmetic operators): In Method 1, we use conventional *arithmetic operators* such as plus (+), minus (-) and multiply (*). In our experiences, Design Compiler using arithmetic operators synthesizes very much fast arithmetic circuits based on many optimizing techniques.

Method 2 (without selector logics): Without using selector logics, all the partial products generated by $(a-b)\times c$ are added up by the Wallace tree.

Method 3: In the computation of $(a-b)\times c$, each digit of $(a-b)$ becomes 0, 1, or (-1). This means that its partial product becomes 0, c, or $(-c)$, according to the result of $(a-b)$. Adding up these partial products leads to the result of $(a-b)\times c$. This approach must look very natural to many LSI designers.

Table 1 Experimental results on subtract-multiplication units.

Function	Method	Word length [bits]	Delay time [ns]	Area [μm^2]
Subtract-multiplication	Method 1	8	0.79 (1.00)	723.95 (1.00)
	Method 2		0.81 (1.03)	1399.91 (1.93)
	Method 3		0.78 (0.99)	1030.00 (1.42)
	Selector logics		0.68 (0.86)	812.32 (1.12)
	Method 1	12	0.90 (1.00)	1471.53 (1.00)
	Method 2		0.93 (1.03)	3048.72 (2.07)
	Method 3		0.96 (1.07)	1918.70 (1.30)
	Selector logics		0.82 (0.91)	1667.16 (1.13)
	Method 1	16	0.98 (1.00)	2302.73 (1.00)
	Method 2		1.01 (1.03)	5870.24 (2.55)
	Method 3		1.05 (1.07)	3220.89 (1.40)
	Selector logics		0.90 (0.92)	2909.01 (1.26)

We used Design Compiler Version B-2008.09-SP4 with the cell libraries in STARC CMOS 90 nm to synthesize them where its objective function is to minimize their delays with no area constraints.

Experimental results for synthesizing a subtract-multiplication unit are shown in **Table 1**. All the synthesized units using our proposed selector logics have smaller delays than those using other designing methods. Particularly, the 8-bit subtract-multiplication unit using our selector logics speeds up by 13.92% in comparison with that using Method 1. Furthermore, Method 1 and Method 2 generally have better performance thatn Method 3 when the the word lengh is larger.

Then we have compared our selector-based Radix-2 and Radix-4 butterfly units with Method 1 and Method 2. Synthesizing conditions are the same as the previous case. **Tables 2** and **3** show the comparison results. In these tables, all the synthesized units using our proposed selector logics also have smaller delays than Method 1 and Method 2. The 8-bit Radix-4 butterfly unit using our selector logics speeds up by 0.16 nsec.

Overall, our speedup ratio is approximately 10% on average compared with Method 1. Area overhead is approximately 50% on average compared with Method 1.

Table 2 Experimental results on Radix-2 butterfly units.

Function	Method	Word length [bits]	Delay time [ns]	Area [μm^2]
Radix-2 butterfly operation	Method 1	8	0.97 (1.00)	2704.92 (1.00)
	Method 2		0.99 (1.02)	4626.97 (1.71)
	Selector logics		0.86 (0.89)	3004.62 (1.11)
	Method 1	12	1.07 (1.00)	5255.49 (1.00)
	Method 2		1.12 (1.05)	10337.04 (1.97)
	Selector logics		0.99 (0.93)	5985.25 (1.14)
	Method 1	16	1.17 (1.00)	8168.73 (1.00)
	Method 2		1.22 (1.04)	17759.78 (2.17)
	Selector logics		1.05 (0.90)	12266.68 (1.50)

Table 3 Experimental results on Radix-4 butterfly units.

Function	Method	Word length [bits]	Delay time [ns]	Area [μm^2]
Radix-4 butterfly operation	Method 1	8	1.17 (1.00)	8594.38 (1.00)
	Method 2		1.15 (0.98)	27973.69 (3.25)
	Selector logics (partial transform)		1.12 (0.96)	10557.36 (1.23)
	Selector logics (complete transform)		1.01 (0.86)	15848.48 (1.84)
	Method 1	12	1.28 (1.00)	16239.74 (1.00)
	Method 2		1.29 (1.01)	58986.04 (3.63)
	Selector logics (partial transform)		1.24 (0.97)	20973.78 (1.29)
	Selector logics (complete transform)		1.15 (0.90)	34214.72 (2.11)
	Method 1	16	1.38 (1.00)	26144.95 (1.00)
	Method 2		1.40 (1.01)	102214.63 (3.91)
	Selector logics (partial transform)		1.37 (0.99)	35995.48 (1.38)
	Selector logics (complete transform)		1.24 (0.90)	59454.03 (2.27)

5. Conclusions

In this paper, we have proposed a fast subtract-multiplication unit by using selector logics. The proposed subtract-multiplication unit reduces the number of partial products from $(2n^2+n+2)$ to (n^2+n+2), where n shows the input bit width. We can apply the proposed subtract-multiplication unit to any applications using subtract-multiplication operations, such as butterfly units and FFT units.

Experimental results show that our proposed subtract-multiplication unit, Radix-2 butterfly unit, and Radix-4 butterfly unit outperforms the conventional implementations using arithmetic operators by an average of 10.33%, and a max-

imum of 13.92%.

In the future, we will develop an automatic synthesis algorithm targeting bit-level transformation and selector logics.

Acknowledgments This research was supported in part by KAKENHI (22300019) and JST A-STEP (AS221Z00015A). This research was also supported by Dai Nippon Printing Co., Ltd., and the authors are deeply grateful to Yuuji Oosumi and Motonobu Tonomura for insightful comments and suggestions.

References

1) Pang, K.F., Soong, H.-W., Sexton, R. and Ang, P.-H.: Generation of high speed CMOS multiplier-accumulators, *Proc. ICCD*, pp.217–220 (1988).
2) Iwamura, J., Taguchi, S., Kazuo, S., Minoru, K., Hiroyuki, T., Kazuaki, I. and Tai, S.: A high speed and low power CMOS/SOS multiplier-accumulator, *Microelectronics Journal*, Vol.14, No.6, pp.49–57 (1983).
3) Krithivasan, S., Schulte, M.J. and Glossner, J.: A subword-parallel multiplication and sum-of-squares unit, *Proc. ISVLSI*, pp.273–274 (2004).
4) Bewick, G.W.: Fast multiplication algorithms and implementation, Ph.D. dissertation, Stanford University (1994).
5) Weste, N. and Harris, D.: *CMOS VLSI Design*, Addison Wesley (2004).
6) Hung, C.-P., Chen, S.-G. and Chen, K.-L.: Design of an efficient variable-length FFT processor, *Proc. ISCAS*, Vol.2, pp.833–836 (2004).
7) Wey, C.-L., Tang, W.-C. and Lin, S.-Y.: Efficient VLSI implementation of memory-based FFT processors for DVB-T applications, *Proc. ISVLSI '07*, pp.98–106 (2007).
8) Wey, C.-L., Tang, W.-C. and Lin, S.-Y.: Efficient memory-based FFT architectures for digital video broadcasting (DVB-T/H), *Proc. VLSI-DAT 2007*, pp.1–4 (2007).
9) Lin, Y.-T., Tsai, P.-Y. and Chiueh, T.-D.: Low-power variable-length fast Fourier transform processor, *Proc. Comput. Digit. Tech.*, Vol.152, No.4, pp.499–506 (2005).
10) Saidi, A.: Generalized FFT algorithm, *Proc. ICC*, Vol.1, pp.227–231 (1993).

(Received May 26, 2010)
(Revised September 3, 2010)
(Accepted October 22, 2010)
(Released February 8, 2011)

(Recommended by Associate Editor: *Yutaka Tamiya*)

Youhei Tsukamoto received his B. Eng. from Waseda University in 2008. He is presently working toward M. Eng. degree there. His research interests include VLSI design and selector-logic applications. He is a student member of Information Processor Society of Japan.

Masao Yanagisawa received his B. Eng., M. Eng., and Dr. Eng. degrees from Waseda University in 1981, 1983, and 1986, respectively, all in electrical engineering. He was with University of California, Berkeley from 1986 through 1987. In 1987, he joined Takushoku University. In 1991, he left Takushoku University and joined Waseda University, where he is presently a Professor in the Department of Computer Science and Engineering. His research interests are combinatorics and graph theory, computational geometry, VLSI design and verification, and network analysis and design. He is a member of IEEE, ACM, and IEICE.

Tatsuo Ohtsuki received his B. Eng., M. Eng., Dr. Eng. degrees from Waseda University in 1963, 1965 and 1970, respectively, all in electrical engineering. In 1965, he joined the NEC Corporation Ltd., Tokyo, Japan. From 1978 to 1980, he served as Research Manager, Application System Research Laboratory, at Central Research Laboratories. In 1980, he left NEC and Joined Waseda University, where he is presently a Professor in the Department of Computer Science and Engineering. His research interests are algorithm and hardware engines for VLSI design and verification, computer algorithms for combinatorial problems, and network analysis/design. He is a fellow of IEEE, and a fellow of IEICE.

Nozomu Togawa received his B. Eng., M. Eng., and Dr. Eng. degrees from Waseda University in 1992, 1994, and 1997, respectively, all in electrical engineering. He is presently a Professor in the Department of Computer Science and Engineering, Waseda University. His research interests are VLSI design, graph theory, and computational geometry. He is a member of IEEE and IEICE.