A C-Testable Multiple-Block Carry Select Adder

Nobutaka KITO†(a), Member, Shinichi FUJII††, Nonmember, and Naofumi TAKAGI†, Senior Member

SUMMARY We propose a C-testable multiple-block carry select adder with respect to the cell fault model. Full adders and 2:1 multiplexers are considered as cells. By an additional external input, we obtain a C-testable carry select adder. We only modify the least significant position of each block. The adder is testable with a test set consisting of 16 patterns regardless of the size of each block and the number of blocks. This is the minimum test set for the adder. We show two gate-level implementations of the adder which are testable with a test set of 9 patterns and 7 patterns respectively, with respect to the single stuck-at fault model.

key words: carry select adder, design for testability, C-testability

1. Introduction

The growth in the number of logic gates integrated in a VLSI chip has made testing chips more difficult. It is important to make chip’s component circuits, such as arithmetic circuits, easily testable. An adder is one of the key component circuits in VLSI systems. Many structures of adders have been studied for various applications. In this paper, we propose an easily testable multiple-block carry select adder. A multiple-block carry select adder consists of several blocks connected in cascade [1]. It is small and suitable for VLSI layout owing to its regular structure. It is fast relative to its small circuit area.

Easily testable adders and test generation methods of them have been proposed in the literature, e.g. [2]–[6]. It is well-known that a ripple carry adder is C-testable with respect to the cell fault model. An arithmetic circuit is said to be C-testable, if it can be tested with a constant number of input patterns independent of its operand size. In [2], a C-testable carry look-ahead adder with respect to the cell fault model is shown. In [3] and [4], C-testable carry look-ahead adders consisting of blocks of fixed size are shown. In [5], a level-testable parallel prefix adder and a level-testable conditional sum adder are shown. An arithmetic circuit is said to be level-testable, if it can be tested with the number of input patterns growing linearly with the number of its circuit levels. In [6], an easily testable carry skip adder and a test pattern generation algorithm for it are shown. The adder in [6] is testable with the number of patterns growing linearly with the number of internal blocks.

Although many easily testable adders and test generation methods for them have been proposed, the testability of a carry select adder has not been determined and no carry select adder testable with a small number of test patterns has been demonstrated, while carry select adders are widely used. Recently, many researches on build-in self test of adders has been proposed such as [7]–[10]. Easily testable adders and knowledge of testability of adder are useful for build-in self test.

In this paper, we propose a C-testable multiple-block carry select adder with respect to the cell fault model and show its test set. Full adders and 2:1 multiplexers are considered as cells. The test set of the adder with respect to the cell fault model is independent of gate-level implementations of cells. We introduce an additional external input and modify a small part of each block in the adder for test. The adder is testable with 16 patterns regardless of the size of each block, the number of blocks, and the operand size of the adder. The number of patterns is minimum for the adder with respect to the cell fault model. Furthermore, we show two gate-level implementations which are testable with 9 and 7 patterns respectively, with respect to the single stuck-at fault model.

This paper is organized as follows. In the next section, we briefly review a multiple-block carry select adder and the cell fault model. In Sect. 3, we propose a C-testable multiple-block carry select adder and a test set for it. In Sect. 4, we show gate-level implementations of the adder and their test set with respect to the single stuck-at fault model. In Sect. 5, we estimate hardware overhead and delay overhead of a gate-level implementation.

2. Preliminaries

2.1 Multiple-Block Carry Select Adder

We consider an n-bit adder. We let the augend X be \([x_{n-1} \cdots x_0]\), the addend Y be \([y_{n-1} \cdots y_0]\), the carry input to the least significant position be \(c_0\), and the sum S be \([s_n s_{n-1} \cdots s_0]\).

A multiple-block carry select adder consists of several blocks as shown in Fig. 1. The size of blocks may differ one block to another (Of course, all blocks may be of the same size). We number the blocks starting from the least significant one, letting the least significant one be block\(_0\). We let the number of blocks be \(b\), and the size of block\(_k\) be \(n_k\)-bit. Block\(_k\) has the carry input \(cin_k\) and the carry output \(cout_k\). Each block except block\(_0\) consists of two ripple carry
adder blocks (RCAs) and a row of 2:1 multiplexors (MUXs). Block0 consists of an RCA. In Fig. 1, RCAs are denoted by rectangles with broken lines.

In each block, except block0, two candidate sums, one assuming a carry input of 0 from the next lower block and the other assuming a carry input of 1, are calculated by two RCAs, and the correct sum is selected by a row of MUXs based on the actual carry output of the next lower block. We name the RCA assuming a carry input of 0 in block_k RCA0_k, and the other one RCA1_k.

In this paper, when a signal name has two subscripts, the first one denotes the block number and the second denotes the position in the block. When a signal name has a superscript, it denotes the RCA that it belongs to. For example, \( s^0_{k,j} (0 \leq j < n_k) \) is the output of the RCA0_k at position j in block_k, and \( x_{k,j} \) and \( y_{k,j} (0 \leq j < n_k) \) are the inputs at position j in block_k. Note that \( x_{k,j} = x_l \) where \( l = j + \sum_{m=0}^{k-1} n_m \).

2.2 The Cell Fault Model

We adopt the 'cell fault model' [11] as the fault model. In the cell fault model, it is assumed that the considered circuit consists of cells. A test set with respect to the cell fault model is independent of gate-level implementation of cells. We treat basic circuit blocks, such as full adders and MUXs, as cells.

In the model, the followings hold.

- At most one cell can be faulty in the circuit.
- The faulty cell is memoryless. Namely, the faulty cell works as a combinational circuit, and its output is determined by only its present input.
- There is at least one input pattern of the faulty cell that makes the output of the cell incorrect.

A test set with respect to this model must satisfy the following two conditions.

- All cells in the circuit must receive exhaustive input patterns when all test patterns in the test set are applied to the circuit.
- The effect of a faulty cell must propagate to at least one of the primary outputs of the circuit.

3. A C-Testable Multiple-Block Carry Select Adder and Its Test Patterns

3.1 A C-Testable Multiple-Block Carry Select Adder

We propose a C-testable multiple-block carry select adder, which is shown in Fig. 2. The adder consists of MUX cells,
FA cells, MAJ cells, INV cells and EXOR cells. A MUX, a MAJ, an EXOR and an INV cells realize 2:1 multiplexer function, 3-input majority function, exclusive or, and inversion, respectively.

We introduce an external input \( z_{in} \) for test. \( z_{in} \) is fed to the blocks except block0. In the normal operation, \( z_{in} \) is 0. We modify only the least significant position of blocks except block0. We show modified parts by bold broken lines in Fig. 2. At the least significant position in block_k (k > 0), the MUX cell can not receive exhaustive patterns because \( s_{k,0}^j \) always takes the inverse value of \( s_{k,0}^0 \). We use an EXOR cell instead of the MUX cell to calculate the output. We calculate only the carry in RCA1_k by using a MAJ cell instead of an FA cell. Note that, in block_k (k > 0) of the original adder (Fig. 1), RCA1_k always outputs a larger value than RCA0_k. Therefore, the MUX cell at the most significant position of block_k can not receive exhaustive patterns because \( c_{k,0}^j \) is always larger than or equal to \( c_{k,0}^0 \). We introduce \( z_{in} \) and the INV cell in order to feed exhaustive patterns to the MUX cell.

3.2 Test Set

We show a test set for the proposed adder. Table 1 shows a set of patterns for test of a block in Fig. 2. Block0 is an RCA, and its function differs from the other blocks when \( z_{in} = 1 \). In parentheses of the 5th column, we show the carry input of block0, i.e., the carry input of the whole adder.

We discuss test of block_k (k > 0). Table 2 shows inputs of the FAs, the MUXs, the MAJ and the EXOR in the block when we apply the patterns in Table 1. Note that the MUX in the most significant position receives carries \( c_{k,n}^0 \) and \( c_{k,n}^1 \) of two RCAs while the other MUXs receive \( s_{k,j}^0 \) and \( s_{k,j}^1 \).

Table 1: Set of input patterns for test of block_k.

| \( j \): even | \( j \): odd | \( z_{in} \) | \( c_{in0}(c_{in1}) \) |
|---|---|---|---|
| 0 | 00 | 11 | 1 (0(1)) |
| 1 | 01 | 01 | 1 (0(1)) |
| 2 | 10 | 10 | 1 (0(1)) |
| 3 | 11 | 11 | 1 (0(1)) |
| 4 | 00 | 00 | 0 (0(0)) |
| 5 | 01 | 01 | 0 (0(0)) |
| 6 | 10 | 10 | 0 (0(0)) |
| 7 | 11 | 00 | 0 (0(0)) |

| 8 | 00 | 11 | 0 (1(1)) |
| 9 | 01 | 01 | 1 (1(1)) |
| 10 | 10 | 10 | 1 (1(1)) |
| 11 | 11 | 11 | 1 (1(1)) |
| 12 | 00 | 00 | 1 (0(0)) |
| 13 | 01 | 01 | 1 (0(0)) |
| 14 | 10 | 10 | 1 (0(0)) |
| 15 | 11 | 00 | 1 (0(0)) |

Table 2: Input patterns of cells in block_k (k > 0).

| FA in RCA0_k | FA in RCA1_k | MAJ | \( c_{k,0}^k \) | \( c_{k,1}^k \) | \( c_{k,0}^k \) c_{k,1}^k c_{in0} \) | \( s_{k,j}^0 \) s_{k,j}^1 c_{in1} \) | \( s_{k,j}^0 \) s_{k,j}^1 c_{in1} \) |
|---|---|---|---|---|---|---|---|
| \( j \): even | \( j \): odd | \( j \): even | \( j \): odd | \( j \): even | \( j \): odd | \( j \): even | \( j \): odd |
| j0 | 00 | 11 | 100 | 001 | 100 | 100 | 011 | 010 |
| j1 | 11 | 00 | 010 | 010 | 010 | 010 | 010 | 010 |
| j2 | 10 | 10 | 100 | 100 | 100 | 100 | 100 | 100 |
| j3 | 00 | 00 | 000 | 000 | 000 | 000 | 000 | 000 |
| j4 | 10 | 01 | 011 | 101 | 010 | 100 | 100 | 100 |
| j5 | 00 | 10 | 010 | 101 | 011 | 100 | 100 | 100 |
| j6 | 01 | 10 | 011 | 101 | 010 | 100 | 100 | 100 |
| j7 | 00 | 01 | 100 | 100 | 100 | 100 | 100 | 100 |
| j8 | 00 | 10 | 101 | 101 | 011 | 100 | 100 | 100 |
| j9 | 01 | 10 | 110 | 110 | 110 | 110 | 110 | 110 |
| j10 | 00 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j11 | 10 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j12 | 01 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j13 | 00 | 01 | 000 | 000 | 000 | 000 | 000 | 000 |
| j14 | 01 | 01 | 001 | 001 | 001 | 001 | 001 | 001 |
| j15 | 10 | 01 | 100 | 100 | 100 | 100 | 100 | 100 |
| j16 | 01 | 01 | 101 | 101 | 101 | 101 | 101 | 101 |
| j17 | 10 | 11 | 110 | 110 | 110 | 110 | 110 | 110 |
| j18 | 00 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j19 | 11 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j20 | 01 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j21 | 10 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j22 | 01 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
| j23 | 10 | 11 | 111 | 111 | 111 | 111 | 111 | 111 |
respectively, for each position \( j \) (\( n_k > j > 0 \)), and we can not generate patterns that produce 1 and 0 for \( c_k \) and \( c_{k-1} \), respectively. These properties are obstacle for achieving C-testability and are eliminated by the external input \( z_{in} \).

We show a test set for the whole adder based on the test set for blocks discussed above. At test generation, we have to mind that the carry input \( cin_k \) of block \( k \) is the carry output \( cout_{k-1} \) of block \( k-1 \). Table 3 shows the test set generation rule for the whole adder. In the table, the first column shows input patterns for block \( k-1 \), the second column shows the values of \( cout_{k-1} \) (i.e., the values of \( cin_k \)), and the third column shows input patterns for block \( k \) when \( n_{k-1} \) is even or odd. Using the table, we can obtain a test set of 16 patterns by constructing patterns that feed patterns from \( t_0 \) to \( t_{15} \) to block \( n_0 \). By this test set, all patterns from \( t_0 \) to \( t_{15} \) are fed to all blocks.

As an example, we show a test set for a 4-block 16-bit adder in Table 4. We assume \( n_3, n_2, n_1, \) and \( n_0 \) are 5, 4, 4, and 3, respectively. In this table, when we apply \( t_0 \) to block \( k \) shown in the first column, we use \( t_0 \) to block \( k \), because \( n_0 \) is odd. In the same way, we use \( t_2 \) and \( t_0 \) to block \( k \) and block \( 3 \), respectively. As shown in the table, each block receives all patterns from \( t_0 \) to \( t_{15} \).

Note that, although we can not observe \( cout_k \) directly, if an faulty cell exists and the effect of it reached \( cout_k \), the effect propagates to the least significant bit position of the next higher block and can be observed at some primary outputs.

### 4. Gate-Level Implementations

We show two gate-level implementations of the proposed C-testable carry select adder and show their test sets with respect to the single stuck-at fault model. We can regard a stuck-at fault at a gate input or a gate output as a fault inside a cell and can detect it by the 16 patterns shown in the previous section. We can also detect a stuck-at fault at a primary input and a primary output by the 16 patterns. Therefore, gate-level adder implementation is testable with at most 16 test patterns with respect to the single stuck-at fault model. In the following, we show modified implementations and derive minimum or near minimum test sets for them.

#### 4.1 Implementation 1

We can obtain a straightforward gate-level implementation of the adder by using the implementation of cells shown in Fig.3 (a), i.e., MUX1, MAJ1 and FA1. We implement an INV cell and an EXOR cell by an inverter and an EXOR gate, respectively. In [4], the minimum test set for FA1 with respect to the single stuck-at fault model is shown. We show it in Table 5 (a), where we can choose either 0 or 1 for \( \alpha \) and \( \beta \) in the table. \( \bar{\beta} \) means the inverse of \( \beta \). We show the minimum test set for MAJ1 in Table 5 (b), where we can choose either 0 or 1 for \( \gamma \).

Table 6 shows patterns for test of a block in the straightforward implementation. The FA1s in RCA1 adder receive all patterns in Table 5 (a) by patterns with \( cin_k = 0 \), i.e., \( t_0 \),...
and $t_1$, $t_5$, $t_6$, and $t_7$, and the FA1s in RCA1 receive them by patterns with $c_{in_k} = 1$, i.e., $t_8$, $t_9$, $t_{10}$, $t_{13}$, $t_{14}$, and $t_{15}$. The MAJ1 receives all patterns in Table 5 (b) by patterns with $c_{in_k} = 1$. Each MUX1 receives exhaustive patterns from 000 to 111. Of course, the inverter implementing INV and the EXOR gate implementing EXOR receive exhaustive patterns. Therefore, if there is a single stuck-at fault, the effect of the fault appears at the outputs of the block.

We can generate a test set for the whole adder using the 11 patterns in Table 6. We show the test generation rule for the straightforward implementation in Table 7. As in Sect. 3.2, with this rule, we can generate a test set for the implementation. While there are 6 patterns $t_8$, $t_9$, $t_{10}$, $t_{13}$, $t_{14}$ and $t_{15}$ with $c_{in_k} = 1$, there are only 5 patterns with $cout_k = 1$, i.e., $t_1$, $t_5$, $t_9$, $t_{10}$ when $n_k$ is even, and $t_1$, $t_7$, $t_9$, $t_{10}$ and $t_{15}$ when $n_k$ is odd. Therefore, we use $t_1$ which is with $c_{in_k} = 0$ and $cout_k = 1$ twice. By the test set generated with this rule, each block receives all the 11 patterns in Table 6. The number of the test patterns is 12.

We can merge some gates in the straightforward implementation. Both the FA1 in RCA0 and the FA1 in RCA1 at the same position calculate $x_k y_k j$ and $x_k j \oplus y_k j$. Therefore, we can merge the parts of the two FA1s. All MUX1 have an inverter generating $c_{in_k}$. We can merge the inverters into one. We show a block of a resultant implementation (implementation 1) in Fig. 4. In the figure, the FA1 in

**Table 7** Test pattern generation rule for the straightforward implementation using FA1.

| Input to block$_{k-1}$ | $cout_{k-1}$ ($= c_{in_k}$) | Input to block$_k$ |
|-------------------------|-----------------------------|-------------------|
| $t_0$                   | 1 / 0                       | $t_5$ / $t_0$     |
| $t_1$                   | 1 / 1                       | $t_{14}$ / $t_{14}$|
| $t_5$                   | 1 / 1                       | $t_{13}$ / $t_{13}$|
| $t_6$                   | 0 / 0                       | $t_6$ / $t_6$     |
| $t_7$                   | 0 / 1                       | $t_7$ / $t_7$     |
| $t_8$                   | 1 / 0                       | $t_8$ / $t_7$     |
| $t_9$                   | 1 / 1                       | $t_9$ / $t_9$     |
| $t_{10}$                | 1 / 1                       | $t_{10}$ / $t_{10}$|
| $t_{13}$                | 0 / 0                       | $t_{13}$ / $t_1$  |
| $t_{14}$                | 0 / 0                       | $t_{14}$ / $t_1$  |
| $t_{15}$                | 0 / 1                       | $t_{15}$ / $t_{15}$|

**Table 8** Test pattern generation rule for implementation 1.

| Input to block$_{k-1}$ | $cout_{k-1}$ ($= c_{in_k}$) | Input to block$_k$ |
|-------------------------|-----------------------------|-------------------|
| $t_0$                   | 1 / 0                       | $t_5$ / $t_0$     |
| $t_1$                   | 1 / 1                       | $t_{14}$ / $t_{14}$|
| $t_5$                   | 1 / 1                       | $t_{13}$ / $t_{13}$|
| $t_6$                   | 0 / 0                       | $t_6$ / $t_6$     |
| $t_7$                   | 0 / 1                       | $t_7$ / $t_7$     |
| $t_8$                   | 1 / 0                       | $t_8$ / $t_7$     |
| $t_9$                   | 1 / 1                       | $t_9$ / $t_9$     |
| $t_{10}$                | 1 / 1                       | $t_{10}$ / $t_{10}$|
| $t_{13}$                | 0 / 0                       | $t_{13}$ / $t_1$  |
| $t_{14}$                | 0 / 0                       | $t_{14}$ / $t_1$  |
| $t_{15}$                | 0 / 1                       | $t_{15}$ / $t_{15}$|

**Table 6** Input patterns of circuits in a block in the straightforward implementation using FA1.

| FA1 in RCA0 | FA1 in RCA1 | MAJ1 | MUX1 | EXOR |
|------------|------------|------|------|------|
| $x_k j$, $y_k j$, $z_k j$ | $x_k j$, $y_k j$, $z_k j$ | $x_k j$, $y_k j$, $z_{in}$ | $c_{in_k}$ | $c_{in_k}$ |
| $j > 0$ | $j > 0$ | $j > 0$ | $j > 0$ | $j > 0$ |
| $j$ | $j$ | $j$ | $j$ | $j$ |
| $j$ : even | $j$ : odd | $j$ : even | $j$ : odd | $j$ : even | $j$ : odd |
| $t_0$ | 001 | 110 | 001 | 110 | 000 | 110/0000 | 110 | 000 | 10 |
| $t_1$ | 011 | 011 | 011 | 011 | 001 | 010/0100 | 010 | 001 | 00 |
| $t_5$ | 010 | 010 | 011 | 011 | 011 | 010/0100 | 100 | 100 | 10 |
| $t_6$ | 100 | 100 | 100 | 100 | 100 | 010/0100 | 100 | 100 | 10 |
| $t_7$ | 110 | 001 | 110 | 000 | 111 | 000/1100 | 000 | 110 | 00 |
| $t_8$ | 001 | 110 | 000 | 111 | 000 | 111/0011 | 111 | 001 | 01 |
| $t_9$ | 010 | 010 | 011 | 011 | 011 | 011/0111 | 101 | 101 | 11 |
| $t_{10}$ | 100 | 100 | 100 | 100 | 100 | 010/1001 | 011 | 011 | 01 |
| $t_{13}$ | 011 | 011 | 011 | 011 | 011 | 011/0111 | 101 | 101 | 11 |
| $t_{14}$ | 101 | 101 | 100 | 100 | 100 | 010/1011 | 011 | 011 | 01 |
| $t_{15}$ | 110 | 001 | 111 | 110 | 001 | 001/1111 | 001 | 111 | 11 |
RCA₀, and the FA₁ in RCA₁ₖ at the same position share a half adder (HA₁) consisting of an AND gate and an EXOR gate. We show the gate-level implementation for the lowest position and one for the other positions in Fig. 4 (a) and (b), respectively.

The implementation of a block shown in Fig. 4 is testable with the patterns shown in Table 6. For the implementation, when \( t₀ \) is applied, the FA₁’s, the MAJ₁’, and the MUX₁’s receive the same pattern as those that they receive when \( t₁₀ \) is applied. The remaining circuits, i.e. the HA₁’s, are testable by \( t₀ \), \( t₁ \), \( t₅ \), \( t₆ \), and \( t₇ \). Therefore, one of \( t₀ \) and \( t₁₀ \) is unnecessary. Similarly, we can omit either of \( t₁₃ \) or \( t₁₄ \). Therefore, the implementation of the block in Fig. 4 is testable with 9 patterns. We need at least 4 patterns for test of the MAJ₁’ and the FA₁’ in Fig. 4 obviously. It is because we need three patterns for test of the AND gate, and by those patterns, we can not detect a fault of the side input of the OR gate. We need at least 5 patterns for test of a pair of FA₁’ and HA₁ (i.e. the original FA₁) and a pair of MAJ₁’ and HA₁ (i.e. the original MAJ₁) as described above. Thus, we need at least 9 (= 4 + 5) patterns for test of the two RCAs.

The set of 9 patterns is the minimum for the implementation of a block.

We show the test pattern generation rule for implementation 1 in Table 8. We can test implementation 1 with 9 test patterns independent of their operand size. This test set is the minimum for implementation 1 because we need at least 9 patterns for block₁ as described above. As an example, we show a test set for the implementation of a 4-block 16-bit adder with the block size \( n₃ \), \( n₂ \), \( n₁ \) and \( n₀ \) being 5, 4, 4, and 3, respectively, in Table 9.

### 4.2 Implementation 2

We can obtain another straightforward gate-level implementation using the implementations of cells shown in Fig. 3 (b), i.e., MUX₂, MAJ₂, and FA₂. Again we implement an INV cell and an EXOR cell by an inverter and an EXOR gate, respectively. In [4], the minimum test set for FA₂ is shown. We show it in Table 10 (a). We also show test sets for MAJ₂, and MUX₂ in Tables 10 (b) and (c), respectively, where we can choose either 0 or 1 for \( \delta \). This implementation of a block is testable with patterns \( t₁, t₃, t₅, t₆, t₀, t₁₀, t₁₁, t₁₃, \) and \( t₁₄ \) in Table 11 because all FA₂s, the MAJ₂, and all MUX₂s receive the patterns in Tables 10 (a), (b) and (c), respectively, and, the inverter and the EXOR gate in the least significant position receive exhaustive patterns.

As in implementation 1, we can merge some gates in the two FA₂s at the same position and inverters in the row...
of the MUXs. We can omit patterns $t_{10}$ and $t_{13}$ for test of this modified implementation. We can derive a further simple implementation. As shown in Table 10 (c), the set of patterns for MUX2 contains patterns with $i_0 = \delta$ and $i_1 = \delta$, and does not contain patterns with $i_0 = \delta$ and $i_1 = \bar{\delta}$. Namely, if we choose $1(0)$ for $\delta$, we need patterns with $i_0 = 0(1)$ and $i_1 = 1(0)$, but do not need patterns with $i_0 = 1(0)$ and $i_1 = 0(1)$ for test of MUX2 unlike MUX1. By this property, we can obtain an C-testable implementation without the additional input $z_{2n}$.

In the resultant implementation (implementation 2), we use the circuits in Fig. 5 (a) and (b) in place of those in Fig. 4 (a) and (b). In this gate-level implementation, as shown in Fig. 5 (a), we use the HA1 at the least significant position of RCA0k in place of FA2. We can use patterns $t_1$, $t_3$, $t_4$ and $t_6$ for test of RCA0k, and patterns $t_9$, $t_{11}$ and $t_{14}$ for test of RCA1k. We modify the least significant part of these patterns to generate the same logic values for $c_{\text{ink}}$ as those in the straightforward implementation. We show the modified patterns in Table 12 and the corresponding input patterns of circuits in the block in Table 13. In Table 13, the columns of EXOR1, EXOR2, EXOR3 correspond to the EXOR gates in the least significant position in Fig. 5 (a). As shown in Table 13, we can test all MUX2 and EXOR gates. Thus, each block is testable with the patterns in Table 12.

Note that $t'_{4}$ is unnecessary for test because all required patterns are fed to all MUX2s, all FA2s (HA1-FA2’s), the HA1 in position 0, and the EXOR gates by the other 6 patterns. As a result, we can test the block with the 6 patterns $t'_{1}$, $t'_{3}$, $t'_{6}$, $t'_{9}$, $t'_{11}$ and $t'_{14}$. It is the minimum set of patterns for the block because we need at least three patterns for test of each AND gate in RCA0k and RCA1k.

We can generate a test set for the whole adder using the above 6 patterns. While there are four patterns $t'_{1}$, $t'_{3}$, $t'_{6}$ and $t'_{9}$ with $c_{\text{out}}k = 1$, there are only three patterns $t'_{9}$, $t'_{11}$ and $t'_{14}$ with $c_{\text{in}}k = 1$. Therefore, we use $t'_{14}$ which is with $c_{\text{in}}k = 0$ and $c_{\text{out}}k = 1$ twice as shown in Table 14. The number of patterns of a test set is 7.

### Table 11

| FA2 in RCA0k | FA2 in RCA1k | MAJ2 | MUX2 | EXOR |
|-------------|-------------|------|------|------|
| $x_{k}j y_{k}c_{\text{ink}}$ | $x_{k}j y_{k}c_{\text{ink}}$ | $x_{k}j y_{k}c_{\text{ink}}$ | $x_{k}j y_{k}c_{\text{ink}}$ | $x_{k}j y_{k}c_{\text{ink}}$ |
| $j > 0$ | $j > 0$ | $j > 0$ | $j > 0$ | $j > 0$ |
| $t_1$ | 011 | 011 | 010 | 010 | 00 |
| $t_3$ | 111 | 111 | 110 | 110 | 10 |
| $t_4$ | 000 | 000 | 001 | 000 | 00 |
| $t_6$ | 100 | 100 | 101 | 101 | 10 |
| $t_9$ | 001 | 011 | 011 | 011 | 01 |
| $t_{10}$ | 100 | 101 | 101 | 101 | 01 |
| $t_{11}$ | 111 | 111 | 111 | 111 | 11 |
| $t_{13}$ | 011 | 001 | 010 | 010 | 01 |
| $t_{14}$ | 101 | 101 | 100 | 100 | 01 |

### Table 12

| $x_{k}j y_{k}$ | $c_{\text{ink}}$ |
|-------------|-------------|
| $j > 0$ | $j > 0$ |
| $t'_{1}$ | 01 | 11 |
| $t'_{3}$ | 11 | 11 |
| $t'_{6}$ | 00 | 00 |
| $t'_{9}$ | 10 | 10 |
| $t'_{9}$ | 01 | 01 |
| $t'_{11}$ | 11 | 11 |
| $t'_{14}$ | 10 | 00 |

5. Hardware Overhead and Delay Overhead

We estimate hardware overhead and delay overhead of implementation 1. Implementation 2 has no additional gate. Hardware overhead of implementation 1 comes from the additional gates at the least significant position of blocks except block0. In other word, blocks in implementation 1 has the same amount of additional hardware. In a block, we added the FA1’, an AND gate in the MAJ1’, and the inverter at the least significant position. We estimate additional hardware by equivalent number of 2-input NAND gates as shown in Table 15. We derive values of Table 15 by the number of transistors in CMOS realization. For example, we can realize a 2-input EXOR gate with 10 transistors. Thus, we consider an EXOR gate as 2.5 gate equivalents because a 2-input NAND gate is realized with 4 transistors. We show hardware overhead(ratio of additional hardware) of a block from 4 bits to 32 bits in Table 16. We also show hardware overhead in a 64-bit adder consisting of blocks of the same size from 4 bits to 32 bits in Table 17. In Table 16, hardware overhead of larger blocks is smaller than that of smaller blocks. Hardware overhead of an adder in Table 17 is smaller than that of a block of corresponding size in Table 16. It is because block0 in an adder has no additional
In normal mode, value for $z_{ik}$ is fixed to 0. Delay overhead of block $k$ is at most one gate delay (OR, AND, and EXOR) at $c_{ik0}, c_{ik1}$, and $s_{ik0}$, respectively. Thus, delay overhead of a block is one gate delay. Delay overhead of a whole adder is gate delay of the number of blocks.

6. Concluding Remarks

We have proposed a C-testable multiple-block carry select adder with respect to the cell fault model. We introduce an additional external input, and modify only the least significant position of blocks. The adder is testable with 16 patterns. The number of the test patterns is the minimum for the proposed adder.

We have shown two gate-level implementations testable with 9 and 7 patterns, respectively with respect to the single stuck-at fault model.

References

[1] O.J. Bedrij, “Carry-select adder,” IRE Trans. Electronic Computers, vol.11, pp.340–344, June 1962.
[2] R.D. Blanton and J.P. Hayes, “Testability of convergent tree circuits,” IEEE Trans. Comput., vol.45, no.8, pp.950–963, Aug. 1996.
[3] W.R. Moore, “Minimal C-testable tests for block-CLA adders,” Int. J. Electron., vol.85, pp.611–628, 1998.
[4] S. Kajihara and T. Sasao, “On the adders with minimum tests,” IPSJ Journal (in Japanese), vol.42, no.4, pp.1045–1053, April 2001.
[5] B. Becker, “Efficient testing of optimal time adders,” IEEE Trans. Comput., vol.37, no.9, pp.1113–1121, Sept. 1988.
[6] D.E. Popescu and C. Popescu, “Efficient algorithms for testing the two-level carry skip adder.” Proc. 49th IEEE International Midwest Symposium on Circuits and Systems, vol.2, pp.319–322, Aug. 2006.
[7] H. Al-Asaad, J.P. Hayes, and B.T. Murray, “Scalable test generators for high-speed datapath circuits,” J. Electronic Testing, vol.12, pp.111–125, 1998.
[8] H.T. Vergos, D. Nikolos, M. Bellos, and C. Efstathiou, “Deterministic BIST for RNS adders,” IEEE Trans. Comput., vol.52, no.7, pp.896–906, July 2003.
[9] D.G. Nikolos, D. Nikolos, H.T. Vergos, and C. Efstathiou, “An efficient BIST scheme for high-speed adders,” Proc. 9th IEEE On-Line Testing Symposium, pp.89–93, July 2003.
[10] M. Pulukuri and C. Stroud, “On built-in self-test for adders,” J. Electronic Testing, vol.25, pp.343–346, 2009.
[11] W.H. Kautz, “Testing for faults in combinational cellular logic arrays,” Proc. 8th Annual Symposium on Switching and Automata Theory, pp.161–174, 1967.
Shinichi Fujii received the B.E. and M.I.S. degrees in information engineering from Nagoya University, Nagoya, Japan, in 2009 and 2011, respectively.

Naofumi Takagi received the 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.