Sneak Path Free Reconfiguration of Via-switch Crossbars Based FPGA

Ryutaro Doi [†, ‡] Jaehoon Yu [†] Masanori Hashimoto [†]

[†] Department of Information Systems Engineering,
Graduate School of Information Science and Technology, Osaka University

[‡] Research Fellow of Japan Society for the Promotion of Science
[doi.ryutaro,yu.jaehoon,hasimoto]@ist.osaka-u.ac.jp

ABSTRACT

FPGA that utilizes via-switches, which are a kind of nonvolatile resistive RAMs, for crossbar implementation is attracting attention due to higher integration density and performance. However, programming via-switches arbitrarily in a crossbar is not trivial since a programming current must be provided through signal wires that are shared by multiple via-switches. Consequently, depending on the previous programming status in sequential programming, unintentional switch programming may occur due to signal detour, which is called sneak path problem. This problem interferes the reconfiguration of via-switch FPGA, and hence countermeasures for sneak path problem are indispensable. This paper identifies the circuit status that causes sneak path problem and proposes a sneak path avoidance method that gives sneak path free programming order of via-switches in a crossbar. We prove that sneak path free programming order necessarily exists for arbitrary on-off patterns in a crossbar as long as no loops exist, and also validate the proof and the proposed method with simulation-based evaluation. Thanks to the proposed method, any practical configurations of via-switch FPGA can be successfully programmed without sneak path problem.

KEYWORDS

Nonvolatile Via-switch FPGA, Sneak Path Avoidance, Crossbar Programming, Switch Programming Order

1 INTRODUCTION

Field programmable gate arrays (FPGAs) are gaining their popularity since the development cost of application specific integrated circuits (ASICs) is elevating due to the device miniaturization and larger scale integration. However, conventional FPGAs are still inferior to ASICs regarding operating speed, power consumption, and implementation area [5]. These drawbacks originate from a large number of programmable switches that are included in FPGAs to acquire reconfigurability. In static random access memory (SRAM)-based FPGAs, which are the most widely used FPGAs, a programmable switch is composed of a transmission gate for switching and an SRAM cell to hold the on/off-state of the switch. These components consist of transistors, and hence the transmission gate has high resistance and large capacitance and the SRAM cell having six transistors consumes large area. Therefore, SRAM-based programmable switches lead to the degradation of interconnect performance and area efficiency [7].

To overcome the drawbacks of conventional FPGAs, FPGAs that exploit resistive random access memories (RRAMs) as programmable switches instead of SRAM-based ones are widely studied [3, 4, 6, 8, 10–12]. In these RRAM-based FPGAs, however, one or two access transistors per a programmable switch are required for switch programming. The access transistor is relatively large despite the small footprint of an RRAM-based switch, and hence it prevents further area reduction. To eliminate access transistors, nonvolatile via-switch is actively developed [1, 2]. The via-switch consists of atom switches, which are a kind of nonvolatile RRAMs developed for application to FPGAs, and varistors in place of access transistors.

In the via-switch FPGA, the crossbar, which has a via-switch at each intersection of horizontal signal wire and vertical signal wire, is responsible for the routing of interconnects. However, programming those via-switches arbitrarily in a crossbar is not trivial since a programming voltage must be given through signal wires that are shared by multiple via-switches. In this case, unintentional switch programming may occur depending on the previous programming status due to signal detour, which is called sneak path problem. This problem interferes the reconfiguration of FPGA, and hence the verification of occurrence conditions and countermeasures is crucially important.

This paper identifies the crossbar programming status that causes sneak path problem and proposes a method that provides a programming sequence of via-switches for sneak path free programming. We prove that such an order for sneak path free programming must exist for arbitrary on-off patterns in a crossbar as long as no loops exist, and devise an algorithm to find the programming order representing the connection status of signal wires in a crossbar as a tree structure. This paper also validate the proof and the proposed method with simulation-based evaluation. Thanks to the proposed method, any practical configurations of via-switch FPGA can be successfully programmed without sneak path problem.

The remainder of this paper is organized as follows. Section 2 explains the structure of via-switch FPGA and sneak path problem. In Section 3, we investigate occurrence conditions of sneak path

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240849

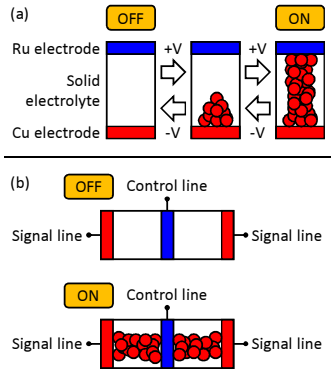

Figure 1: Structure and operation of (a) atom switch and (b) CAS.

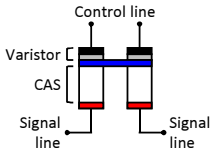

Figure 2: Via-switch structure.

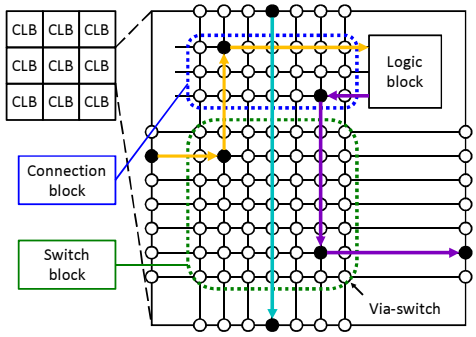

Figure 3: Structure of via-switch FPGA.

problem and identify the circuit status that causes sneak path. Section 4 proposes a sneak path avoidance method that determines the sneak path free programming order of via-switches in a crossbar followed by the simulation-based validation in Section 5. Concluding remarks are given in Section 6.

2 VIA-SWITCH FPGA

2.1 Via-switch

The via-switch is a nonvolatile, rewritable, and compact switch that is developed to implement a crossbar switch by Banno et al. [1], and it is composed of atom switches and varistors. Here, we explain the device structure, functionality, and characteristics in the following. The programming of via-switch crossbar will be explained later in Section 2.2.

The atom switch consists of a solid electrolyte sandwiched between copper (Cu) and ruthenium (Ru) electrodes as shown in Figure 1-(a). By applying a positive voltage to the Cu electrode, a Cu bridge is formed in the solid electrolyte, and the switch turns on. On the other hand, when a negative voltage is applied, Cu atoms in the bridge are reverted to the Cu electrode, and then the switch turns off. The switching between on-state and off-state is repeatable, and each state is nonvolatile. For improving the device reliability, the complementary atom switch (CAS) is devised, where it consists of two atom switches connected in series with opposite direction as shown in Figure 1-(b). In the programming of CAS, a pair of signal line and control line supply a programming voltage to each atom switch, and two atom switches are programmed sequentially. During normal operation, on the other hand, only signal lines are used for routing [8].

To accurately provide the programming voltage only to the target atom switch in a switch array, the varistor is introduced into the via-switch. Figure 2 shows the structure of via-switch, where the varistor is connected to the control terminal of CAS. When a voltage higher than the threshold value (programming voltage) is applied between the signal and control lines, the varistor supplies programming current to an atom switch. On the other hand, the varistor isolates the control lines from the signal lines during normal operation [1].

Here, we summarize the main of via-switch to FPGAs. The footprint, on-resistance, and capacitance are 18 F^2, 200 Ω, and 0.14 fF respectively [1, 9]. Thanks to these characteristics, the area efficiency and performance of via-switch FPGA are dramatically improved compared to SRAM-based one. Ochi et al. report that the crossbar density is improved by 26x, and the delay and energy in the interconnection are reduced by 90% or more at 0.5 V operation [9].

2.2 Sneak Path Problem in Via-switch FPGA

The structure of via-switch FPGA is an array of configurable logic blocks (CLBs), and each CLB is composed of a logic block and a crossbar where a via-switch is placed at each intersection of signal lines as shown in Figure 3 [9]. The via-switch in the crossbar is responsible for connection and disconnection between the horizontal and vertical signal lines. Besides, the top half of the crossbar serves as input and output multiplexers to the logic block and corresponds to the connection block in conventional FPGAs. On the other hand, the bottom half of the crossbar, which corresponds to the switch block, routes global interconnections. The logic block organizes combinational and sequential circuits.

The following explains the programming of a via-switch crossbar and sneak path problem. Figure 4 illustrates the via-switch based crossbar structure. Both signal and control lines are aligned horizontally and vertically. Figure 4 also exemplifies programming steps in 2x2 crossbar where an atom switch is turned on at each step. A pair of the perpendicular signal and control lines crossing at the via-switch of interest are used for switch programming. Two programming drivers are activated at each step, and a positive voltage is given to one of the signal lines, and a ground voltage is given to one of the control lines. Other lines are floated. We can see that the via-switch at the bottom left is successfully turned on at steps (1) and (2). Succeeding steps (3) and (4) also turn on the top left via-switch normally. However, the next programming of the bottom right via-switch at step (5) cannot be performed correctly. The atom switch that composes the top right via-switch is under programming unintentionally at step (5) since the positive voltage is provided through the on-state via-switches at the bottom left and top left. Such an unintentional switch programming due to signal detouring through on-state via-switches is sneak path problem. The sneak path problem interferes the reconfiguration of FPGA, and hence it is essential to identify the occurrence conditions and find countermeasures of this problem.

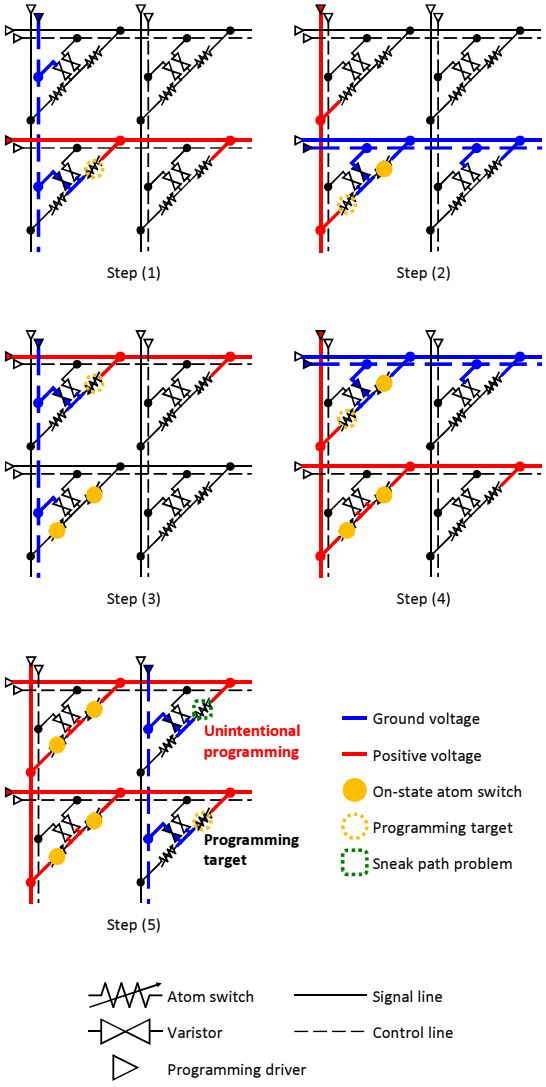

Figure 4: Via-switch based crossbar structure and switch programming steps.

2.3 Conventional Countermeasure for Sneak Path Problem

Here, let us introduce a conventional countermeasure for the sneak path problem and its drawback. Ochi et al. claim that the sneak path problem can be avoided by imposing a programming constraint [9]. This constraint allows multiple on-state via-switches on the same signal line only in one direction. In other words, this constraint prohibits the configurations in which multiple on-state via-switches exist in both the same horizontal line and the same vertical line such as step (5) in Figure 4. The authors also prove that there is no sneak path problem in the programming of any-sized crossbar under the programming constraint with mathematical induction. However, their countermeasure involves a clear disadvantage. The programming constraint prohibits some configurations of via-switch FPGA, and hence imposing the constraint leads to a decrease in the number of available configurations. Consequently, the routing flexibility is limited.

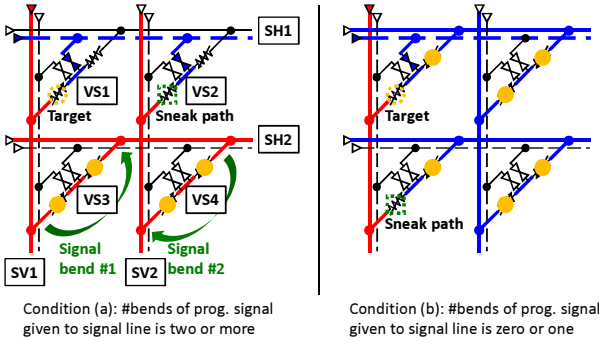

Figure 5: Two occurrence conditions of sneak path problem.

3 OCCURRENCE CONDITIONS OF SNEAK PATH PROBLEM

This section clarifies the programming status of a crossbar that leads to the sneak path problem for developing a more efficient countermeasure. As explained in Section 2.2, the atom switch at the intersection is turned on when a positive voltage is provided to the signal line and a ground voltage is provided to the control line. When the number of such intersections is two or more, the sneak path problem occurs. Focusing on the number of bends of the programming signal given to the signal line, we can classify the circuit status that causes the sneak path problem into two situations, namely conditions (a) and (b) as shown in Figure 5. In condition (a), the programming signal provided to the signal line bends twice or more, whereas the number of signal bends is one or zero in condition (b). The followings discuss each condition in detail. It should be noted that we only consider the programming operations to turn on the atom switch in the following because programming operations to turn on and off an atom switch are a symmetrical operation and the same discussion can be done by swapping the voltage given to the signal line and control line.

In condition (a) of Figure 5, two vertical signal lines SV1 and SV2 are connected by multiple on-state via-switches VS3 and VS4 in the same line SH2. When a programming signal is given to one of signal lines SV1 and SV2, the same signal is provided to the other signal line, which means we cannot distinguish SV1 and SV2 anymore in the programming. Therefore, when we try to turn on the lower atom switch of via-switch VS1 in the line SV1, the atom switch in the same position of signal line SV2, i.e. lower atom switch of via-switch VS2 is programmed simultaneously. In summary, the sneak path problem arises when programming an atom switch in already undistinguishable vertical lines or undistinguishable horizontal signal lines. From the opposite point of view, we could avoid the sneak path problem if we would program such an atom switch before multiple vertical/horizontal signal lines become undistinguishable. For example, in condition (a) in Figure 5, we need to turn on the lower atom switch of via-switch VS1 before programming VS3 and VS4. It should be noted that, for programming a via-switch, we must turn on two atom switches, which are the lower atom switch connected to the vertical signal line and the upper atom switch connected to the horizontal signal

line. The vertical signal line is used when programming the lower atom switch (e.g. step (2) in Figure 4), and hence we need to pay attention to multiple on-state via-switches in the same horizontal signal line that connect multiple vertical signal lines. On the other hand, we should care about multiple on-state via-switches in the same vertical signal line when programming the upper atom switch.

Let us move to condition (b) in Figure 5, where the number of bends of programming signal given to the signal line is one or zero. In this case, a sneak path problem can occur in configurations where the control signal is detoured through two bents as shown in condition (b) of Figure 5. In this case, the two horizontal control lines are indistinguishable. On the other hand, such a condition is satisfied only when a loop is intentionally programmed in a crossbar. For example, in Figure 5-(b), all the four via-switches are intended to be turned on, otherwise the top two atom switches of VS1 and VS3 never be on. This condition arises only when programming the last two atom switches that compose a loop, and the ground voltage applied to the control line is propagated to the non-target switch because of a loop structure.

From the above discussion, the sneak path problem cannot be avoided in the configurations that include a loop. Fortunately, such configurations with a loop are not used in practical applications since the looped signal routing increases the wire capacitance and degrades delay and power compared to non-loop routing. Therefore, we do not need to take care of condition (b). Consequently, what we have to consider for sneak path avoidance is only condition (a).

4 PROPOSED SNEAK PATH AVOIDANCE METHOD

In this section, we propose a sneak path avoidance method based on the discussion in Section 3. The proposed method gives sneak path free programming order of via-switches in a crossbar. Target configurations are non-looped configurations, which are used for signal routing. Some key properties necessary for proving that a sneak path free programming order necessarily exists are in *italic*. With those properties, Section 4.3 gives a proof.

4.1 Overview of Proposed Method

The proposed method consists of two steps: step (1) turning on all the upper atom switches of interest and step (2) turning on all the lower atom switches of interest.

In step (1), all the upper atom switches of the via-switches to be turned on in a target configuration are programmed. The via-switch connects the vertical and horizontal signal lines only when both the upper and lower atom switches that compose the via-switch are on-state. Therefore, any signal lines are not connected each other in step (1), and hence no sneak path problem arises in this step. Then, *arbitrary programming order works fine in step (1).*

Step (2) turns on all the lower atom switches to be programmed after step (1) completion. In step (2), a vertical line and a horizontal line are connected by a via-switch each time a lower atom switch is turned on since the corresponding upper atom switch is already turned on in step (1). Therefore, we have to determine the programming order of the lower switches paying attention to the occurrence condition of sneak path problem as discussed in Section 3. Please remind that we provide a programming signal

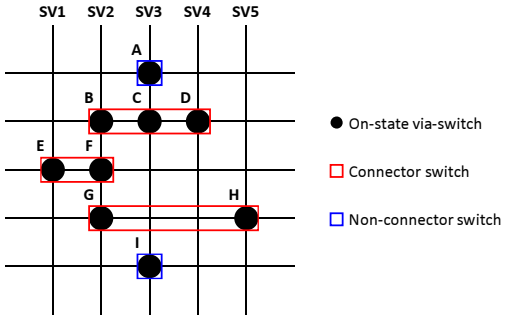

Figure 6: Example of non-looped configuration and definition of connector/non-connector switches.

to a vertical signal line when programming a lower atom switch and hence we care about only multiple on-state via-switches in the same horizontal signal line. Multiple on-state via-switches in the same vertical signal line do not matter. The details on how to determine the programming order are explained in the next subsection.

It should be noted that the swapped sequence of step (2) followed by step (1), i.e. programming all the upper atom switches after turning on all the lower atom switches can also avoid the sneak path problem since the crossbar has a symmetrical structure. In this case, we need to determine the programming order of the upper switches. In the following, we consider only the sequence of step (1) and step (2).

4.2 Programming Order Determination with Connection Tree

Here, we explain how to derive a sneak path free programming order of via-switches in a crossbar. In the following, we use a configuration of a 5x5 crossbar shown in Figure 6 as an example.

As mentioned in the previous subsection, we have to pay attention to multiple on-state via-switches in the same horizontal signal line that connect multiple vertical signal lines and make them undisistinguishable. Therefore, we define two categories of the via-switch, namely connector switch and non-connector switch. When multiple on-state via-switches exist in the same horizontal signal line, these switches are classified as the connector switch. On the other hand, when there is only one via-switch in the horizontal signal line, we categorize such a switch as the non-connector switch. For example, via-switches B, C, D, E, F, G, and H are connector switches and via-switches A and I are non-connector switches in Figure 6. A pair of connector switches connects two vertical signal lines, e.g. via-switches E and F connect vertical signal lines SV1 and SV2 in Figure 6. The non-connector switch, on the other hand, does not connect any vertical signal lines.

Please remind that the sneak path problem occurs when turning on the lower atom switch included in the already connected vertical signal lines as explained in Section 3. Therefore, *all the non-connector switches should be programmed before the connector switches* so that we can avoid the sneak path problem in programming the non-connector switches since the connector switches which may connect vertical lines are still off-state. In this case, *the programming order of the non-connector switches is arbitrary.*

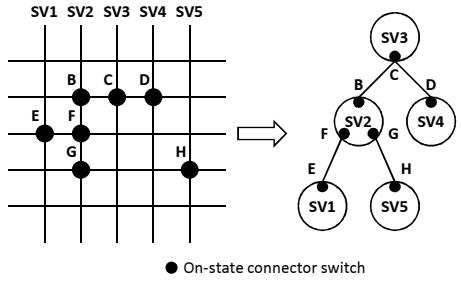

Figure 7: Example of connection tree for connector switches in Figure 6.

Next, we determine the programming order of connector switches. In this step, the programming order is not arbitrary since the sneak path problem may arise depending on the programming order. For example in Figure 6, when we are turning on the connector switches B or G after programming the connector switches of E and F, the sneak path problem occurs since switches E and F connect vertical lines SV1 and SV2.

To determine the programming order, we construct a connection tree that represents the connection status of vertical signal lines in a crossbar. Figure 7 exemplifies a connection tree for the connector switches in Figure 6, where each node corresponds to a vertical signal line. The root node can be arbitrarily selected. Vertical line SV3 is selected as the root node in Figure 7. When two vertical signal lines are supposed to be connected in the configuration of interest, we give an edge between the two nodes corresponding to these two vertical signal lines. Two black dots depicted at both ends of an edge represent connector switches, and when both the two connector switches are turned on, we suppose the edge is activated and two vertical lines are connected. From the definition, *any non-looped configurations can be necessarily expressed by the tree structure*, which means loops are not included in the graph.

The connection tree tells us *the connector switch that can be turned on at the end of programming as the leaf node of the connection tree.* Let us explain what happens when we program the connector switch in the leaf node and non-leaf node of the connection tree at the last programming step, where the last programming step means that only one switch remains off and the others are already turned on in the target configuration. In Figure 8-(a), the connector switch in the leaf node SV1 is under programing, and the other connector switches are already on-state. In this case, node SV1 and node SV2 are not connected yet, and hence the programming signal never propagate to any other vertical signal lines. Consequently, the target connector switch can be turned on without the sneak path problem. We can also confirm that there is no sneak path problem when programming the connector switch in the leaf node from Figure 8-(b), which is the circuit diagram corresponding to Figure 8-(a). Next, let us turn on the connector switch in non-leaf node SV2 in Figure 8-(c) at the last programming step. In this case, the programming signal reaches the other vertical signal lines through the connector switches that are already on-state, and consequently the sneak path problem arises. The circuit diagram of Figure 8-(d) also indicates that atom switches placed at the same vertical position as the target on the connected indistinguishable vertical signal lines are unintentionally programmed.

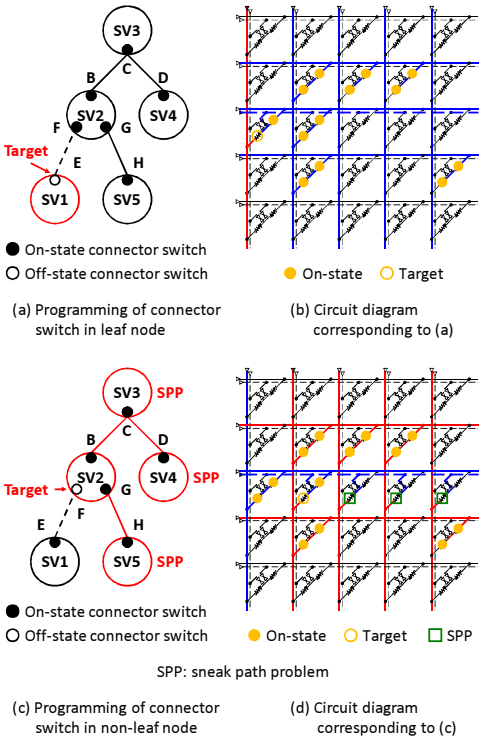

Figure 8: Programming of connector switch in leaf/non-leaf node at the last programming step.

We propose to *recursively search a connector switch that can be turned on at the final programming step* for obtaining a sneak path free programming order of connector switches. Figure 9 illustrates the recursive process. Here, there are two types of connector switches in each node, namely the connector switch connected to the parent node (e.g. switch B in node SV2) and the connector switch connected to the child node (e.g. switches F and G in node SV2). *In each node, all the switches connected to the child node must be turned on before the switch connected to the parent node.* Otherwise, the sneak path problem arises when programming the switch connected to the child node since the programming signal is propagated to the parent node through the on-state switch to the parent node as pointed out in Figure 8-(c).

Let us roll back the recursive programing step one by one with Figure 9. As we discussed, we can program only the connector switch in the leaf node at the final programming step. Then, switch E in SV1 is selected as the last switch to be programmed and node SV1 and the edge between SV2 and SV1 are deleted. This modified graph is again analyzed to find the next last switch to be programmed. In this case, switch H is selected. After SV1 and SV5 are removed from the graph, SV2 has no child nodes, and hence switch B in leaf node SV2 connecting to parent node SV3 can be programmed. In this way, one recursive process chooses one leaf node, identifies the switch in the leaf node connecting to the parent node as the last switch to be programmed in the current graph, and remove the leaf node and its edge to the parent node. Eventually, *all the nodes except the root node are removed, and the recursive process finishes.* It should be noted that there remain some on-state

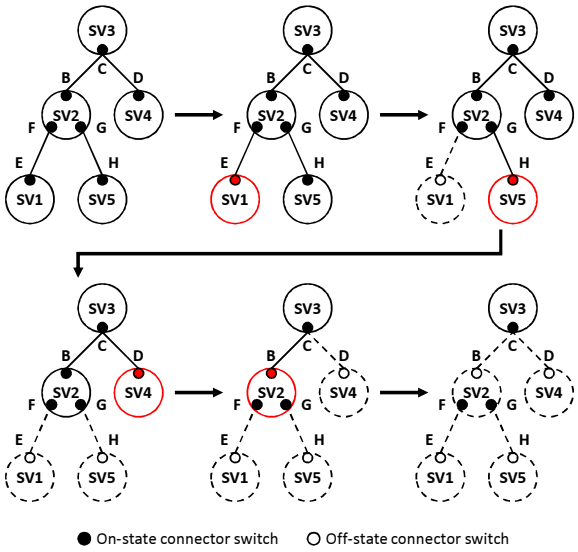

Figure 9: Recursively searching switch which can be programmed lastly for each shrinking graph.

connector switches, for example switches C, F, and G in Figure 9. These switches can be programmed in an arbitrary order as long as they are programmed before the switches selected in the recursive processes.

Depending on the target configuration, multiple connection trees may be constructed for a non-looped configuration. Each tree has no connection to other trees, and hence the programming signal never propagates to other trees when programming the switch in the tree of interest. Consequnetly, we can handle each connection tree independently with the proposed method.

4.3 Proof of Existence of Sneak Path Free Programming Order

This subsection proves that a sneak path free programming order always exists for arbitrary non-looped configurations. Firstly, all the upper atom switches can be programmed in an arbitrary order as clarified in the second paragraph of Section 4.1. After this, non-connector switches can be programmed in an arbitrary order, which is verified in the third paragraph of Section 4.2. In the following, the programming of remaining connector switches is addressed in the proof. The proof consists of three lemmas, and hence we prove them.

Lemma 4.1. *Given a non-looped configuration, it can be expressed by a connection tree or multiple trees.*

Proof. Any non-looped configuration can necessarily expressed by a undirectional graph or multiple graphs without loops where each node corresponds to each vertical signal line. When a node is selected as the root node, the graph is expressed by a tree structure, which is called connection tree in the previous section. □

Lemma 4.2. *At the last programming step for a connection tree, only a switch in a leaf node connecting to its parent node can be programmed without the sneak path problem.*

Proof. When programming a switch in a leaf node at the final programmming step, there is no connection between the leaf node and its parent node, and hence the programming signal never propagates to any other nodes. On the other hand, a non-leaf node has at least two connections, i.e. connection to its parent node and child node. Therefore, programming a switch in a non-leaf node at the end always causes the sneak path problem by propagating the programming signal to other node through the connection to the parent or child node. □

Lemma 4.3. *Recursively searching a switch which can be programmed at the last programming step always finds the sneak path free programming order.*

Proof. By recursive searching a switch that can be turned on at the final programming step in the current graph and removing the leaf node and its edge to the parent node from the graph, all the nodes except the root node must be eventually eliminated and the recursive search necessarily finishes since the tree must have at least one leaf node when the number of nodes is two or larger. The remaining switches can be programmed in an arbitrary order before the switches chosen in the recursive search since each node has no connection to other nodes at this moment, i.e. all the vertical signal lines are distinguishable. □

4.4 Pseudo Code and Execution Example

Algorithm 1 summarizes the overall determination procedure of a programming order. This algorithm determines a sneak path free programming order of non-connector and connector switches, and stores it to queue *ProgOrder*. Line 1 defines a set $\mathbb{S}$ of switches to be turned on. Lines 2-4 search non-connector switches by checking the number of on-state switches in each horizontal signal line. Specifically, line 2 creates a set $\mathbb{H}_j$ of on-state switches in j-th horizontal signal line, line 3 enumerates non-connector switches in j-th horizontal line where the number of elements of $\mathbb{H}_j$ is one, and line 4 enqueues all the non-connector switches to *ProgOrder*. Then, line 5 removes all the non-connector switches from the set $\mathbb{S}$, and subsequent lines 6-9 determine the programming order of connector switches. Line 6 selects i^*-th vertical signal line, in which the connector switch to be turned on exists, as the root node of a connection tree, and the programming order of connector switches in this connection tree is determined by the function SEARCH in line 7. SEARCH is a recursive function and traverses a connection tree from the root node to leaf nodes. Please remind that all the switches connected to child nodes need to be programmed before programming any switch connected to its parent node. SEARCH classifies the switches connected to child nodes of the parent node i^* (line 11) and the switches connected to the parent node i^* (line 15). The former is enqueued to *ProgOrder* in line 16 since switches connected to the child have to turn on before the switches connected to the parent. On the other hand, the latter is enqueued in $Buffer$ in line 17 until all the switches connected to the child node are enqueued to *ProgOrder* by the recursive function SEARCH. This function is recursively executed for all the child nodes of the node of interest (lines 19-21), and returns when the node of interest is a leaf node of the connection tree (lines 12-14). Finally, all the switches connected to the parent node in all nodes are enqueued to *ProgOrder* in line 9.

Let us explain an example when the proposed algorithm is applied to the configurations as shown in Figure 6. Lines 2-4 find

Algorithm 1 Programming order determination of non-connector and connector switches.

1: $\mathbb{S} = \{S_{i,j} \mid S_{i,j}$ is on-state, $0 \le i < W, 0 \le j < H\}$
2: $\mathbb{H}_j = \{S_{i,j} \mid S_{i,j} \in \mathbb{S}\}$
3: $\mathbb{S}_N = \{S_{i,j} \mid |\mathbb{H}_j| = 1, S_{i,j} \in \mathbb{S}\}$
4: ENQUEUE($\mathbb{S}_N$) to $ProgOrder$
5: $\mathbb{S} \leftarrow \mathbb{S} - \mathbb{S}_N$
6: **for** $i^* \in \{i \mid \exists S_{i,j} \in \mathbb{S}\}$ **do**
7: SEARCH($i^*, \mathbb{S}, ProgOrder, Buffer$)
8: **end for**
9: ENQUEUE($Buffer$) to $ProgOrder$

10: **function** SEARCH($i^*, \mathbb{S}, ProgOrder, Buffer$)
11: $\mathbb{S}_P = \{S_{i^*,j} \mid S_{i^*,j} \in \mathbb{S}\}$
12: **if** $\mathbb{S}_P = \emptyset$ **then**
13: **return**
14: **end if**
15: $\mathbb{S}_C = \{S_{i,j} \mid i \neq i^*, \exists S_{i^*,j} \in \mathbb{S}_P\}$
16: ENQUEUE($\mathbb{S}_p$) to $ProgOrder$
17: ENQUEUE($\mathbb{S}_c$) to $Buffer$
18: $\mathbb{S} \leftarrow \mathbb{S} - (\mathbb{S}_P \cup \mathbb{S}_C)$
19: **for** $i' \in \{i \mid \exists S_{i,j} \in \mathbb{S}_C\}$ **do**
20: SEARCH($i', \mathbb{S}, ProgOrder, Buffer$)
21: **end for**
22: **end function**

W: crossbar width; H: crossbar height

non-connector switches A and I, and enqueue them to $ProgOrder$. Line 7 determines a programming order of the remaining connector switches B-H. Assuming the vertical line SV3 is selected as a root node i^*, at the first execution of the function SEARCH, the set $\mathbb{S}_P$ contains the switch C connected to the child node as shown in Figure 7, and the set $\mathbb{S}_C$ contains switches B and D connected to the parent node (root node). Line 16 enqueues the switch C to $ProgOrder$ and line 17 stores switches B and D to $Buffer$. After that, function SEARCH is executed again for vertical lines SV2 and SV4 where switches B and D exist. When SEARCH is executed for line SV2, set $\mathbb{S}_P$ contains switches F and G, and set $\mathbb{S}_C$ contains switches E and H. On the other hand, when SEARCH is executed for line SV4, set $\mathbb{S}_P$ is empty and SEARCH returns. Eventually, we successfully obtain a sneak path free programming order and it is switches A, I, C, F, G, B, D, E, and H. Figure 10 shows the programming order determined by the proposed method in two example configurations. In each configuration, we can see that the proposed method finds a sneak path free programming order.

Also, in the case that there exist multiple connection trees, "for statement" in line 6 of Algorithm 1 is executed as many times as the number of connection trees.

Thus far, we discussed the programming order for operations of turning on the switch. Programming operations to turn on and off an atom switch are symmetric except that applied voltages to the signal line and control line are reversed. Therefore, we can turn off all the switches without the sneak path problem in the reverse order.

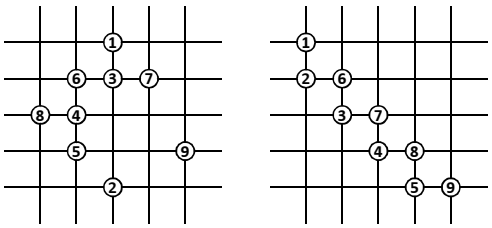

Figure 10: Results of programming order determination with the proposed method in two non-looped configurations.

Table 1: Evaluation results of comprehensive simulation for small crossbars.

Crossbar size	# of all configs.	# of looped configs.	# of non-looped configs.	
			No SPP	SPP occurs
2x2	16	1	15	0
3x3	512	184	328	0
4x4	65,536	49,391	16,145	0
5x5	33,554,432	32,078,576	1,475,856	0

SPP: sneak path problem

5 EXPERIMENTAL RESULTS

5.1 Simulation based Validation

The previous section proved that the proposed method could find a sneak path free programming order of via-switches for arbitrary configurations without a loop in an arbitrarily-sized crossbar. Here, we validate the discussion in the previous section with simulations just in case. We implement the proposed method and verify the sneak path problem in each programming step of the determined programming order. We perform two types of evaluation, which are comprehensive evaluation for small crossbars and Monte Carlo evaluation for large crossbars.

In small crossbars, the total number of configurations is relatively small, and hence we can simulate all the configurations. We evaluated four crossbars including 2x2, 3x3, 4x4, and 5x5 crossbars with the comprehensive simulation. Table 1 shows evaluation results. The column of "No SPP" in the table represents the number of non-looped configurations where the sneak path problem can be avoided by the determined programming order. The column of "SPP occurs" lists the number of non-looped configurations where the proposed method cannot eliminate the sneak path problem. As discussed in Section 3, the sneak path problem is unavoidable in looped configurations, but those configurations are practically meaningless for signal routing. In any non-looped configurations, on the other hand, we can see that the sneak path problem can be avoided by the proposed method in Table 1. This simulation results definitely support the theoretical proof in the previous section.

When the number of via-switches in a crossbar is n, the number of possible configurations is 2^n since each via-switch has two states, namely on- and off-state. Consequently, the total number of configurations exponentially increases as the crossbar size becomes larger, and the comprehensive simulation of larger crossbars is infeasible. We generated random configurations for

Table 2: Evaluation results of Monte Carlo simulation.

% of on-state via-switches	# of samples	# of looped configs.	# of non-looped configs.	
			No SPP	SPP occurs
2%	10,000	10,000	0	0
1%	10,000	3,690	6,310	0
0.5%	10,000	154	9,846	0
0.1%	10,000	0	10,000	0

SPP: sneak path problem

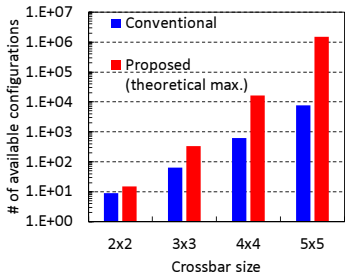

Figure 11: Number of available configurations with conventional countermeasure and proposed method.

a large crossbar in Monte Carlo manner and checked the sneak path avoidance. In this evaluation, the crossbar size was set to 100x100 and the number of trials was 10,000. We also varied the percentage of on-state via-switches in the 100x100 crossbar from 0.1% to 2%. Evaluation results are shown in Table 2. The proposed method successfully solved the sneak path problem in non-looped configurations as we expected with the proof in Section 4.

5.2 Advantage of Proposed Method

Next, we discuss the advantage of the proposed method. As explained in Section 2.3, the conventional countermeasure for the sneak path problem imposes a programming constraint that prohibits a class of configurations of the via-switch FPGA. Therefore, the number of usable configurations decreases in the conventional countermeasure, and consequently the routing flexibility is diminished. On the other hand, the proposed method can give a sneak path free programming order for any non-looped configurations. Figure 11 compares the number of programmable configurations with the conventional countermeasure and the proposed method in 2x2, 3x3, 4x4, and 5x5 crossbars. In this evaluation, the conventional countermeasure prohibits configurations in which multiple on-state via-switches exist in both the same horizontal line and the same vertical line. We can see that the proposed method increases the number of usable configurations compared to the conventional countermeasure. Even in the small 5x5 crossbar, the number of available configurations increases by over two orders of magnitude. The figure also shows that the increasing ratio of the number of usable configurations becomes larger as the crossbar size increments, which suggests that the increase in the number of configurations that are newly enabled by the proposed is significant in practically-sized crossbars.

There are almost no disadvantages of the proposed countermeasure since the crossbar structure is unchanged and then no hardware overhead is required. A small disadvantage is the computation of the sneak path free programming order, but it can be derived efficiently by the proposed algorithm.

6 CONCLUSION

This paper identified programming status of the via-switch crossbars based FPGA that cause the sneak path problem, and clarified that the sneak path problem cannot be avoided in looped configurations. On the other hand, we have proved that a via-switch programming order which can avoid the sneak path problem always exists for all the non-looped configurations, and we proposed a sneak path avoidance method that gave sneak path free programming order of via-switches in a crossbar. We also validated the proof and the proposed method with supportive simulation-based evaluation. The proposed method successfully solves the sneak path problem in any practical configurations of via-switch FPGA. Future works include evaluating the computational complexity of the proposed algorithm.

ACKNOWLEDGMENTS

This work was supported by JSPS KAKENHI Grant Number JP17J10008 and JST CREST Grant Number JPMJCR1432, Japan.

REFERENCES

[1] N. Banno, M. Tada, K. Okamoto, N. Iguchi, T. Sakamoto, M. Miyamura, Y. Tsuji, H. Hada, H. Ochi, H. Onodera, M. Hashimoto, and T. Sugibayashi. 2015. A novel two-varistors (a-Si/SiN/a-Si) selected complementary atom switch (2V-1CAS) for nonvolatile crossbar switch with multiple fan-outs. In *2015 IEEE International Electron Devices Meeting (IEDM)*. 2.5.1–2.5.4.

[2] N. Banno, M. Tilda, K. Okamoto, N. Iguchi, T. Sakamoto, H. Hada, H. Ochi, H. Onodera, M. Hashimoto, and T. Sugibayashi. 2016. 50x20 crossbar switch block (CSB) with two-varistors (a-Si/SiN/a-Si) selected complementary atom switch for a highly-dense reconfigurable logic. In *2016 IEEE International Electron Devices Meeting (IEDM)*. 16.4.1–16.4.4.

[3] J. Cong and B. Xiao. 2014. FPGA-RPI: A Novel FPGA Architecture With RRAM-Based Programmable Interconnects. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 22, 4 (April 2014), 864–877.

[4] P. E. Gaillardon, D. Sacchetto, G. B. Beneventi, M. H. Ben Jamaa, L. Perniola, F. Clermidy, I. O'Connor, and G. De Micheli. 2013. Design and Architectural Assessment of 3-D Resistive Memory Technologies in FPGAs. *IEEE Transactions on Nanotechnology* 12, 1 (Jan 2013), 40–50.

[5] I. Kuon and J. Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (Feb 2007), 203–215.

[6] Y. Y. Liauw, Z. Zhang, W. Kim, A. E. Gamal, and S. S. Wong. 2012. Nonvolatile 3D-FPGA with monolithically stacked RRAM-based configuration memory. In *2012 IEEE International Solid-State Circuits Conference*. 406–408.

[7] M. Lin, A. El Gamal, Y. C. Lu, and S. Wong. 2007. Performance Benefits of Monolithically Stacked 3-D FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (Feb 2007), 216–229.

[8] M. Miyamura, T. Sakamoto, M. Tada, N. Banno, K. Okamoto, N. Iguchi, and H. Hada. 2014. Low-power programmable-logic cell arrays using nonvolatile complementary atom switch. In *Fifteenth International Symposium on Quality Electronic Design*. 330–334.

[9] H. Ochi, K. Yamaguchi, T. Fujimoto, J. Hotate, T. Kishimoto, T. Higashi, T. Imagawa, R. Doi, M. Tada, T. Sugibayashi, W. Takahashi, K. Wakabayashi, H. Onodera, Y. Mitsuyama, J. Yu, and M. Hashimoto. 2018. Via-Switch FPGA: Highly Dense Mixed-Grained Reconfigurable Architecture With Overlay Via-Switch Crossbars. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26 (2018), 1–14.

[10] K. Okamoto, M. Tada, T. Sakamoto, M. Miyamura, N. Banno, N. Iguchi, and H. Hada. 2011. Conducting mechanism of atom switch with polymer solid-electrolyte. In *2011 International Electron Devices Meeting*. 12.2.1–12.2.4.

[11] S. Tanachutiwat, M. Liu, and W. Wang. 2011. FPGA Based on Integration of CMOS and RRAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19, 11 (Nov 2011), 2023–2032.

[12] X. Tang, P. E. Gaillardon, and G. De Micheli. 2014. A high-performance low-power near-Vt RRAM-based FPGA. In *2014 International Conference on Field-Programmable Technology (FPT)*. 207–214.