Journal of Engineering Sciences

Vol 10, Issue 8, Aug /2019
ISSN NO:0377-9254

FSM Method For Reducing Rollback Faults In Memories

[#1]G VASANTHI, [#2]SHAIK MOHAMMAD SHAFI
[1,2]Assistant Professor
DEPT OF ECE
Dr.K V SUBBA REDDY INSTITUTE OF TECHNOLOGY, KURNOOL

ABSTRACT— In nanometer technologies, circuits are more and more sensitive to various kinds of perturbations. Alpha particles and atmospheric neutrons induce single-event upsets, affecting memory cells, latches, and flip-flops. They also induce single-event transients, initiated in the combinational logic and captured by the latches and flip-flops associated with the outputs of this logic. In the past, the major efforts were related on memories. However, as the whole situation is getting worse, solutions that protect the entire design are mandatory. Solutions for detecting the error in logic functions already exist, but there are only few solutions allowing the correction, leading to a lot of hardware overhead in nonprocessor design. In this paper, we present a novel technique that includes several hardware architectures and an algorithm for their implementations, which reduces the cost of rollback in any kinds of circuit.

The parity generating technique is one of the most widely used error detection techniques for the data transmission. In digital systems, when binary data is transmitted and processed , data may be subjected to noise so that such noise can alter 0s (of data bits) to 1s and 1s to 0s. Hence, **parity bit** is added to the word containing data in order to make number of 1s either even or odd.Thus it is used to detect errors , during the transmission of binary data .The message containing the data bits along with parity bit is transmitted from transmitter node to receiver node. At the receiving end, the number of 1s in the message is counted and if it doesn't match with the transmitted one, then it means there is an error in the data.

I. INTRODUCTION

"In 2008, a Quantas Airbus A330-303 pitched downward twice in rapid succession, diving first 650 feet and then 400 feet. ... The cause has been traced to errors in an on-board computer suspected to have been induced by cosmic rays." [7]

• "Canadian-based St. Jude Medical issued an advisory to doctors in 2005, warning that single bit-flips in the memory of its implantable cardiac defibrillators could cause excessive drain on the unit's battery." [8]

This list could be continued by other examples of drastic consequences of fault occurrences. Proper circuit functionality even under perturbations and faults has been always crucial in aerospace, defense, medical, and nuclear applications. Circuit tolerance towards transient faults (non-destructive, non-permanent) is an important research topic and an unavoidable characteristic of any circuit used in safety critical applications. Common sources of faults are natural radiation, such as neutrons of cosmic rays and alpha particles of packing or solder materials, capacitive coupling, electromagnetic interference, etc [7, 9]. Nowadays, technology shrinking and voltage scaling increase electronics susceptibility and the risk of fault occurrences. Circuit engineers use fault-tolerance techniques to mask or, at least, to detect faults. Regardless of the chosen technique, this step increases the level of complexity of the whole design. Commonly used simulation-based methodologies are not able to fully verify even the functional correctness due to the huge number of possible execution cases.

The verification of fault-tolerance properties by checking all fault injection scenarios raises the order of complexity. Non-exhaustive manual checks or simulation-based techniques are error-prone and may miss a circuit corruption scenario that leads to the loss of the circuit functionality or to degraded quality of service. Since engineers need their implementations to be simple and correct, they mostly use Triple-Modular Redundancy (TMR), a technique that triplicates the circuit and introduces majority voters. Modern EDA tools support TMR, as well as other basic techniques such as Finite State Machine (FSM) encoding [10–12], through automatic circuit transformations. While there are other more elegant and optimized fault-tolerance techniques [13, 14], their functional correctness and fault-tolerance properties are often not guaranteed.

Ensuring correctness of fault-tolerance techniques requires mathematically based techniques for the specification, development, and verification. Formalization of fault-models, circuit designs, and specifications gives a vast opportunity to create, to optimize, and to check the correctness of fault-tolerance techniques. Showing fault-tolerance properties w.r.t. the chosen fault-model eliminates all doubts about the circuit functionality under the faults whose occurrence and type are specified by the fault-

model. Thanks to this formal verification, the overall probability of the system failure is purely the probability of faults occurring outside of the fault-model. There are many different formal methods to verify properties of systems or circuits. In this dissertation, we mainly use static symbolic analysis and theorem proving.

1.1 objective

we presented a solution based on the principle that several isolate registers and embedded memories could have common state preserve mechanism. We showed that it leads to a great reduce of the hardware overhead compare to [23] and [24]. The counterpart is that the rollback is executed in more than one clock cycle. In this paper, we present a set of solutions based on the same idea. We propose two kinds of solutions for the registers: one with a fixed maximal number K for the rollback (K-cycle rollback [30]) that continuously update the last save state; the other solution is a checkpoint-based rollback that only save specific states of the system. We detail three architectures to protect the memories (including [23] and [24]). Using directed graph representation, we show how the implementation could be automated. We improve the concept presented in [30] with an algorithm to optimize the additional resources and reduce the hardware overhead. Instruction replay is already used in both single and multiprocessors and induces few additional clock cycles. Therefore, our targets are nonprocessor architectures. We will compare the hardware result of the proposed solutions depending of the number of K clock cycles required on several designs.

II. EXISTING METHOD

2.1 SPM FOR MEMORIES

During the retry phase, the logic parts will activate the same read and write operations over the memories of the system as during the regular phase. During the read operations of the retry phase, the memories of the system must provide the data that were supposed to be provided during the regular phase. However, during the last K cycles of the regular phase, the data of some memory locations can be modified by some write operations, and the memory may not be able to provide the expected data during the retry phase. Thus, the memories must have a specific SPM. The data in a memory can be stored since long time. Then, an error occurring in a memory and affecting the data stored in the memory for more than K-cycles cannot be corrected by a retry that repeats the last K cycles. Thus, long-term memories (i.e., that can store data for more than K-cycles) will have to be protected. This can be achieved, by using an ECC to protect the memory like in [8]. The solution discussed here must handle two main cases as follows.

Case a) First during the K clock cycles that must be repeated, a read can be performed over a memory address followed by a write at the same memory address. Then, during the retry phase, the value of the read data cannot be obtained by reading the memory address since the content was modified by the write operation. Thus, during the regular phase, the data obtained by the read operation must be saved in the SPM mechanism, and during the retry phase, this data must be provided to the system by the SPM mechanism.

Case b) Suppose that an error occurring in the system can contaminate data written in the memory r cycles after the occurrence of the error. Consider that the read of this data is performed within the $K - r$ cycles following the write. The data read during the regular phase will be erroneous. Thus, if during the regular phase the read data are saved in the SPM and then is used during the read operation of the retry phase, the incorrect data will be used during the retry phase. On the other hand, during the retry phase, the write data provided to the memory by the logic parts of the system will be correct. Those data must be used instead the ones saved in the SPM. To simplify the design, we use the following options.

1) During the regular phase, we do not save any write data in the SPM. We save the read data to the SPM.

2) During the retry phase, we perform each write over the memory. We do not perform any write over the SPM.

If during the last K-cycles of the regular phase a write operation is performed after a read operation in the same memory address, then, we provide to the system the data of the read operation that was stored in the SPM during the regular phase. If such a write was performed before the read operation, then, the data read from the memory are provided to the system. For doing so, a valid indicator is mandatory, to distinguish between the above two cases. This indicator can be updated during the regular phase. However, this updating is complicated, because when a read over a memory is performed, it cannot be predicted the instant at which an error detection may later occur. Thus, it cannot be known if a write performed previously on the same memory address will belong to the last K-cycles of the regular phase. Therefore, it is much easier to update the indicator after the occurrence of the error detection. In the implementation described below, it is done during the retry phase. To update the valid indicator, we will need to determine if an operation performed at a given cycle matches the address of a previous

operation. We can use for that a CAM. Each location of such a memory can be addressed by the contents of a subset of its bits.

2.1.1 First Implementation

In this implementation, we use a CAM composed of three fields that are detailed below [Fig. 6(a)] [30]. The address field stores the address of a read operation. The data field saves the read data. In this implementation, only the read operations of the regular phase are stored. The valid-flag field indicates if the data stored in a CAM location are valid, so they can be used during the retry phase. The size of this field will be one bit. We opted for a sequential addressing, which addresses cyclically all the CAM words one after another. In each memory cycle, a different word is selected. Thus, for a CAM of K words, the same word is selected every K-cycles. During the regular phase, if a read is performed a write operation is activated on the current CAM location.

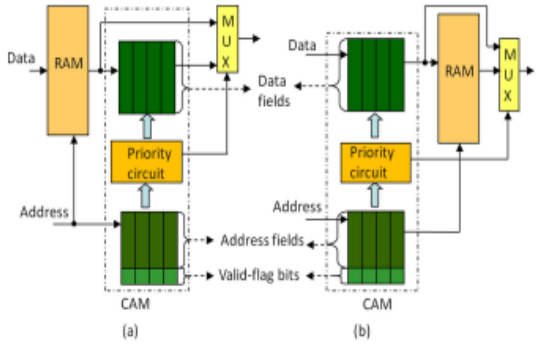


Fig. 6. Memory SPM. (a) Preserving read data (first and second implementation). (b) Delaying write data (third implementation).

This write consists on: storing the address of the read operation in the address field of the present CAM location; storing the data read from the memory in the data field of the present CAM location; setting to "1" the flag bit of the present CAM location. When no data are read, the flag bit is set to "0." During the retry phase, the flag bit of the present CAM location is examined for each read operation. If this bit is equal to "1," the data provided in the system are read in the data field of the current CAM location. If the flag bit is "0," then, the data provided in the system are read from the memory.

When a write operation is performed over the memory, at the same time the write address is compared in parallel with the contents of the address field of all the CAM locations. If the comparison matches with the content of some of these address fields, the corresponding flag bit(s) will be set to "0." Thus, when a write over a memory address is performed during the retry phase, the flag bit of each CAM location whose address field is identical to the current write address operation is set to "0." This way, the subsequent read operation(s) over this address will read the data from the memory.

2.1.2 Second Implementation

In this implementation, the data fields and the address fields become independent. Therefore, the data could be stored in a simple FIFO. The address fields and the flag fields are stored in a CAM. During the regular phase, all read data are stored in the FIFO. During the retry phase, the CAM is used as follows. At the beginning of the retry phase, all the flag fields are set to "0." When data are written in the memory, the address is stored inside the CAM, and the related flag is set to "1." When a read in a memory is requested, the address is compared with the addresses stored in the CAM. If the address was stored in the CAM, and the related flag was set to "1," then the data are read from the memory. Otherwise, the data are read from the FIFO. To be sure that the data read from the FIFO are the one that is mandatory, the FIFO is active during each cycle of the retry phase, even if its data are not used. Therefore, during the ith cycle of the re-execution phase the FIFO will provide to the system the data read from the memory during the $(K - i + 1)$th cycle before the error signal occur. It is worth to notice that this idea could also be used for the first implementation. Indeed, by updating the three fields like an FIFO, it can be considered that the data read from the CAM will always be the oldest one even if there are multiple addresses that match the desired one. Therefore, those two first implementations differ essentially by the way the CAM is used (storing the read addresses or the write addresses) and thus the hardware overhead are quite equivalent.

2.2 Global Memories' SPM Implementation

The previous descriptions detailed the SPM of the memories to re-execute the K clock cycles before the error detection signal becomes active. However, before the retry phase starts, the subsequent pipeline connected to the memory output must be brought back to its Kth previous state. So, additionally to the CAM associated with the memory, an SPM that preserves the related pipeline state must be inserted. This SPM depends of the CAM implementation, and if a checkpoint rollback implementation or a K-cycle rollback implementation is considered. Fig. 7 describes the global SPM implementation of a RAM (or a register file) with the first and the second implementation. The read data are saved during several clock cycles in the CAM and then it is saved in the SPM for the registers related to the subsequent

pipeline during a few more cycles. To perform a K-cycle rollback, the CAM must save the K last read operations. However, instead of using an FIFO of K + P − 1 stages to save the data of the subsequent pipeline of P-stages, like in Section II-A, an FIFO of P − 1 stages is sufficient.

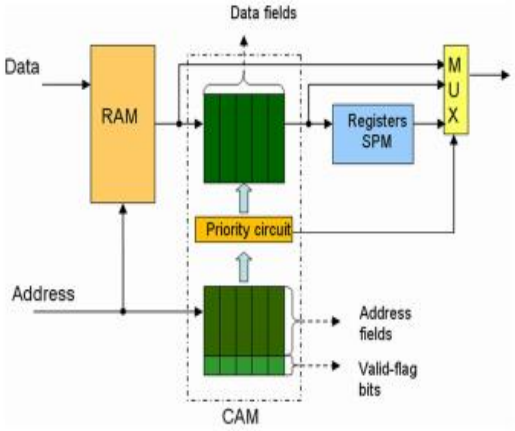


Fig. 7. Global memory SPM: first and second implementation.

Therefore, the read data are saved in the CAM during the K clock cycles after they are read, and then during P − 1 additional cycles inside the registers SPM. Therefore, the global implementation saves the read data during K +P−1 clock cycles as in Section II-A. The data of the FIFO are used to restore the context of the circuit K clock cycles before, and the data of the CAM to perform another time the K last operations of the circuit. Obviously, the CAM is not used during the rollback phase. On the other hand, as the third implementation save the data that are written in the memory instead of those that are read, the register SPM for the third implementation (Fig. 8) must be an FIFO of K + P −1 stages. In case a checkpoint rollback implementation is chosen, the results are quite different. As said in Section II-B, the re-execution process may involve a complete retry set.

Therefore, the CAM used in each implementation must meet that requirement. To evaluate that cost, let us consider the best case for the retry sets: it must avoid that the least recently saved context to be contaminated by an error. So, during a retry set the next save must start at least K clock cycles after the beginning of the retry set (see Fig. 9). If PM is the largest pipeline size of all regular pipeline, then each CAM must save the N + K last operations, with N ≥ K + PM − 1. For the third implementation, the register SPM must, as in Section II, be composed of two FIFOs of P-stages. On the other hand, for the two first implementations, the requested checkpoints will be always saved either inside the CAM or in a register SPM that save the least recent checkpoint (Fig. 9).

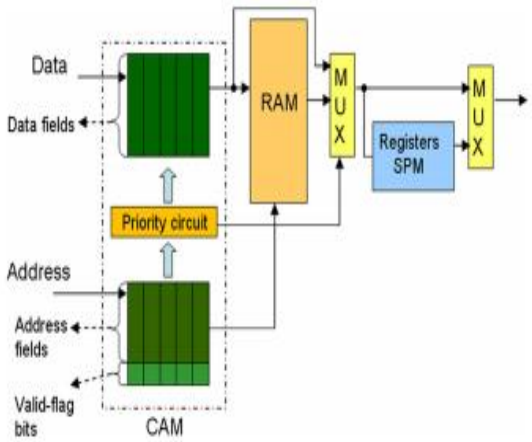


Fig. 8. Global memory SPM: third implementation.

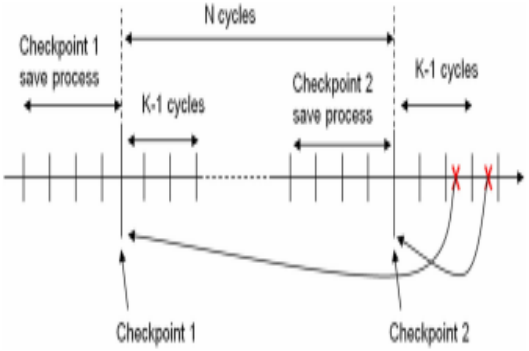


Fig. 9. Global memory SPM: third implementation.

Therefore, the register SPM in this implementation has only one FIFO of P-stages. So, the third CAM implementation always lead to a higher hardware overhead, due to the additional mechanism used to restore the context of the subsequent pipeline. Notice that those solutions differ from another solution described in [36], which used a read buffer and not a CAM. This solution is dedicated to instruction replay, and do not consider the global state restoration K clock cycles before of the subsequent pipeline. However, the interest of this solution is that they avoid having a priority circuit by performing a write back of the read buffer content inside the register file before the instruction replay starts [36]. Nevertheless, this idea is compatible with the first memory SPM we proposed that save both read data and read addresses. The two other implementations cannot use this principle.

2.3 ALGORITHM

The manual implementation of the proposed solution could be difficult, particularly concerning the

additional FIFOs, their positions and their sizes. Therefore, in this section, we detail an algorithm that handles this issue. Moreover, the algorithm reduces the hardware overhead of the straightforward implementation of the solution and provides the associated finitestate machine (FSM) used to restore the context. The circuit is modeled by a directed graph similar to the one used in [8]. The nodes of the graph represent the storage component and the arcs the combinational path between two storages components.

The algorithm is divided in four steps:

1) identification of the regular part of the circuit, and the source node;
2) computation of the size of each FIFO associated with the source node, depending of the decontamination process (with or without checkpoints);
3) reduction of the hardware overhead;
4) description of the FSM restoring the previous context to re-execute the clock cycle in which the error appeared.

A. Definitions

Vertices (V): The set V includes all storage components of a circuit (registers or isolated latches), plus the memory inputs, and the memory outputs which are considerate separately. Arcs (A): Consider (v1, v2) ∈ V2. If the output of v1 is connected through a combinational circuit to an input of v2, (v1, v2) is an arc. Then, A is the set of all arcs of the circuit. Path of Length n (Pn): Consider an integer n ∈ N∗ and n + 1 vertices (v0 ...vn) ∈ V n+1, such that ∀i, 0 ≤ i < n, (vi, vi+1) ∈ A. Therefore (v0,...vn) ∈ Pn. Minimal Distance (dmin): Consider (u,v) ∈ V 2, such that it exists n ∈ N∗ and (v0,...vn) ∈ Pn, v0 = u, and vn = v. Therefore dmin(u,v) = min{n ∈ N ∗ /∃(v0,...vn) ∈ Pn, v0 = u and vn = v}. The distance "0" is not defined, as this measure must take into account the loop paths that may exist, like in Fig. 3: R10-Log8-R10. It takes one clock cycle to go through that path. Thus, dmin(R10, R10) = 1. Input (Ip): Consider a vertex v ∈ V , the set of its input is defined as: Ip (v) = {u ∈ V/(u,v) ∈ A}.

B. Sources and Regular Pipeline Identification

A regular pipeline, as explained in Section II, must verify several properties. The set V of the circuit is split in several sets of nodes, which all verify those properties. Each source register of the circuit defined in Section II corresponds to a source node. Property 1: Every vertex of the graph must be at equal distance from all the source nodes they depend. Property 2: Consider a vertex that is not a source node; all its inputs must be at the same distance from all their sources. The property 2 is not automatically verified if the property 1 is verified. As the circuit is modeled by a directed graph we use three variables to characterize each node v of the circuit: f (v) = (SO (v), dS(v), SR(v)). SO (v) = "1" when v is identified as a source node, SO (v) = "0" otherwise. dS(v) is an integer that indicates the distance of the node v from its related sources SR(v) is the set of source nodes from which the node v is related to. The two properties enounced before are labeled as (1) and (2) and can be expressed as follows:

$$\forall v \in V,\ SO(v)=0$$
$$\exists! n \in N*,\ n=dS(v)/\forall s \in SR(v),\ dmin(s,v)=n \quad (1)$$
$$\forall v \in V,\ SO(v)=0,\ SO(I\,p(v))=0$$
$$\exists! n \in N\backslash\{0,1\},\ n=dS(v)/\forall u \in I\,p(v),\ dS(u)=n-1. \quad (2)$$

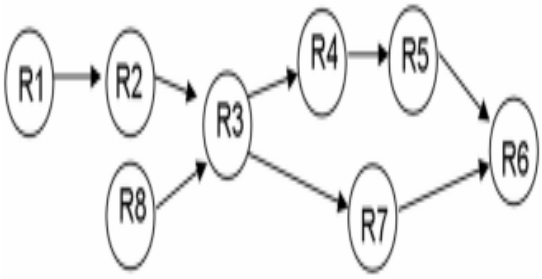


Fig. 10. Example of graph for algorithm illustration.

In addition, we defined for all source nodes v that dS(v) = 0. At the beginning of the algorithm, all variables are initialized to 0 except for all the nodes that correspond to memory outputs, or the nodes that have not any predecessors. They are identified as sources nodes, as they are the first node of their subsequent pipeline. Then, the algorithm will attribute values recursively to the three variables of each nodes of the circuit until all of them verify the two properties. The algorithm is illustrated with the graph of Fig. 10. At the beginning, only R1 and R8 are identified as source nodes. In a first step, the algorithm that performs a deep first search exploration starts by exploring the path (R1, R2, R3, R4, R5, R6). R2, R3, R4, R5, and R6 are marked as related to R1, and dS(R2) = 1, dS(R3) = 2 … dS(R6) = 5. The path (R1, R2, R3, R4, R7, R6) is considered next. Therefore, an issue is detected as the algorithm may now consider that dS(R6) = 4. Variables dS(R6) and SR(R6) are deleted, and R6 is now identified as a source node. The next step is related to the source node R8 (R6 was identified as a source node after R8, so its successors are considered after R8). As for R6, an issue is identified with the node R3. Therefore, R3 is considered as a source node and the associated variables of R4, R5, and R7 are reinitialized. At the end of the process four sources nodes are identified

(R1, R3, R6, R8). The other nodes verify the following properties: SR(R2) = {R1}, dS(R2) = 1, SR(R4) = {R3}, dS(R4) = 1, SR(R5) = {R3}, dS(R5) = 2, SR(R7) = {R3}, and dS(R7) = 1. However, there is a case when, after a source node identification, the reinitialization of the three variables of the successor nodes must be done carefully. In Fig. 11, after the source node R1 was considered, dS(R3) = dS(R7) = 2, and dS(R4) = 3. When the source node R8 is considered, an issue is detected first with the node R7. Therefore, R7 is identified as a source node, and the variables associated with R4 and R5 are reinitialized. However, if it is done without any consideration of the node R3, it leads to a mistake. During the next step, considering the source node R7, R4 is marked with

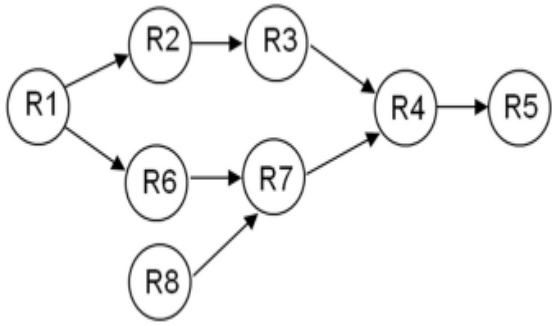


Fig. 11. Example of graph for algorithm illustration.

dS(R4) = 1, even if dS(R3) = 2. Therefore, the restoration of the context of R4 could not be done properly, because to be restored it needs the correct data coming from both R7 and R3 at the same time. In this case, the property 1 is verified without verifying the property 2. Indeed, R4 is not a source node and dS(R3) = 2 and dS(R7) = 0. Due to the reset operated over the successor of the node R7, R4 is still related to the node R7, but not to the node R1 anymore. Therefore, the FIFO that will be associated with R1 will not be adapted to restore the previous state of R4. To cope with that, R4 will be also identified as a source node after checking that the property 2 was not verified.

2.4 Hardware Cost Reduction

At the end of the precedent step, all the sources nodes are identified, as well as the depth of the associated regular pipeline $P(vS) = Max\{dS(v)/vS \in SR(v)\} + 1$. So, depending of the number K and the implementation chosen (with or without checkpoint), the size of each FIFO is known. Those FIFOs feet for the decontamination of the registers they are associated with. However, by moving backward several FIFOs to other associated vertex, the hardware overhead could be reduced. But there is a cost to pay for moving an FIFO from a vertex to one of its input vertices, i.e., the size of each associated FIFO increases by one to handle the added pipeline stage (as shown in Fig. 2). However, hardware reduction is possible, when moving backward a convergence point is reached in which a single FIFO (of higher size) replaces several FIFOs and MUXes. Note that, for a checkpoint rollback implementation, there are two FIFOs to be considered per source node. Fig. 12(a) and (b) corresponds to the same circuit, but with different options. For simplicity, it is assumed in this illustration that all register stages have the same number of bits. In case (a), REG1 is associated with a four-stage FIFO, and REG2 with a three-stage FIFO. On the other hand, in case (b), the FIFOs associated with REG1 and REG2 were moved to REG3, which is one pipeline stage further. Then REG3 is associated with a five-stage FIFO that reduces the hardware cost. If the three registers have not the same size, case (b) reduces the hardware overhead only if it verifies

$$(H[FIFO(REG1)] + H[MUX(REG1)] + H[FIFO(REG2)]+H[MUX(REG2)]) > (H[FIFO(REG3)]+H[MUX(REG3)]) \quad (3)$$

here H [A] is the hardware cost of the element A, FIFO (REG) is the FIFO associated with the register REG, and MUX (REG) is the MUX associated with the register REG. While this may be not very accurate, it is worth noting that this algorithm works with any cost function of the hardware components, and its efficiency in terms of execution time and solution space exploration is independent of the used cost function. The previous illustration shows that convergent vertices (CVs) having as successors several vertices of the original set of sources are more likely leading to hardware cost reduction. Thus, the cost reduction algorithm starts from each source node and goes backward to the graph description of the design, to determine CVs that have several successors belonging to the first set of sources identified. For each of these CVs the hardware cost of the initial solution is compared to the hardware cost of the solution that replaces by one CV all the successors of this CV.

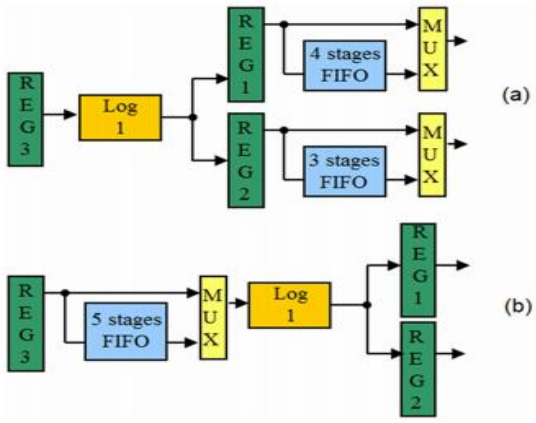


Fig. 12. Illustration of hardware cost reduction (a) without optimization and (b) with optimization.

Note, however, that, replacing by a CV only a subset of its successors may be more efficient. Indeed, if the cost of a successor is low (small FIFO depth and/or small number of bits) and the increase of the size of the FIFO associated with CV is high (i.e., when the distance between CV and this successor is high, or due to an increase of the number of bits stored), replacing this successor may increase cost. Thus, the replacement algorithm starts from the successors that are closer to CV, and then gradually move to the successors that are further apart. This approach guaranties determining the set of successors of CV that lead to the lowest cost when replaced by CV. At the end, the solution providing the lowest hardware cost is selected. However, this does not guarantee that this solution provides the lowest hardware overhead, as the algorithm only focuses on CVs and thus does not consider all possible solutions. Unfortunately, the global optimization problem is NP-complete. Considering only the convergence vertices improves the execution time, which depends of the circuit architecture and increases with both the number of vertices and the architecture irregularities.

2.5 Context Restoration Phase

During the restoration phase, the additional FIFOs and the subsequent pipeline must be unlocked in a specific order. The biggest FIFO has the largest number of states to restore. The last correct state of all components must be restored during the same clock cycle. Therefore, the largest FIFO is unlocked first, and so on in the reverse order of their size. Notice that for the rollback operation, only the FIFOs that belong to the registers or the memory are involved. The CAMs are only used for the re-execution phase and so are inactive during the rollback phase, except for a checkpoint rollback implementation if the desired data are inside a CAM (as explained in Section III-D). Each pipeline stage is unlocked depending of the state of the previous component. If all storages components connected via a logic function to the input of a particular one were unlocked during the previous cycle (or their associated FIFOs for the storage components that are a source node); therefore, this particular component could be unlocked during this current cycle.

Algorithm 1 generates the FSM for the decontamination operation. For a vertex c, c.lock returns the state lock (1) or unlock (0); c.valid determines if the data provide by c (or its associated FIFO) are valid (1) or nonvalid (0); if c is a source, then c.fifo-lock determines if the associated FIFO is active (0) or inactive (1). As stated in Section IV, V is the set of all nodes of the circuit, and S is the set of the source nodes after the optimization process. In case the implementation is a checkpoint rollback, the line 11 "if FIFO (c) = K + PM − clk then" is replaced by "if FIFO (c) = PM − clk + 1 then," and the FIFO is chosen depending of the used retry set. When a source node is unlocked during a given clock cycle, the associated FIFO will not be any more used starting from the next clock cycle. Notice that the PM value has been updated during the optimization phase, as the FIFOs after optimization may need to handle more stages. Let us consider Fig. 11 and let us suppose that K = 2.

Algorithm 1 FSM Algorithm Generation for Rollback

```
1: /*Initialization*/
2: for all vertices c ∈ V do
3:       c.lock = 1
4:       c.valid = 0
5:       c.fifo-lock = 1
6: end for
   /*Rollback Steps*/
8: for Clock clk from 1 to PM do
9:       /*Unlock a FIFO at a given clock cycle*/
10:      for all vertex c ∈ S do
11:         if FIFO(c) = K+PM − clk then
12:            c.valid = 1
13:            c.fifo-lock = 0
14:         end if
15:      end for
         /*mark as valid the registers unlocked during the
         previous clock cycle and stops the FIFO for sources
         which storage component where already unlocked*/
17:      for all vertices c ∈ V do
18:         if c.lock = 0 then
19:            c.valid = 1
20:            c.fifo-lock = 1
21:         end if
22:      end for
         /*unlock the registers for which each predecessor
         has a valid state*/
24:      for all vertices c ∈ V do
25:         if c.lock = 1 then
26:            c.lock = 0
27:            for all vertices c'∈ Ip(c) do
28:               if c'.valid = 0 then
29:                  c.lock = 1
30:               end if
31:            end for
32:         end if
33:      end for
34: end for
   /*Resume operation*/
36: for all vertices c ∈ V do
37:      c.lock = 0
38:      c.valid = 1
39:      c.fifo-lock = 0
40: end for
```

2.6 ADDING DELAY-FREE ECC TO ROLLBACK TECHNIQUE

In addition to those experimentations, we combined the solution developed in [8] and the solutions developed in this paper. The solution proposed in [8] deals with the delayfree ECCs implementation to protect embedded memories. Therefore, it can handle

data stored during a long time in memory affected by a soft error. A simple rollback may be not sufficient in that case if the data were stored a long time ago, and the technique we have presented yet is not adapted. Instead, it is better to check and correct the data integrity using ECC.

A. Delay-Free ECC Architecture It uses two memories, one for the data bits and one for the check bits, to avoid any speed penalty during write operations. The data bits are stored in memory without waiting for the check bits computation. During read operations, the data integrity is checked in parallel. Therefore, the read data are used by the computing elements without waiting for possible error detection. With the above solutions, data bits are written in and are read from the memory at full speed. However, if the read data are affected by an error, the downstream logic is then contaminated and may provide false results. To cope with this issue, in [8] was proposed to insert the decontamination circuitry in the error propagation path enabling restoring the correct circuit state. The decontamination process introduces few extra clock cycles, but as it is activated extremely rarely, it barely affects circuit performance. The decontamination circuitry is shown in Fig. 19, where a MUX is used to bypass the correction circuit during normal operation. However, when the detection circuit indicates an error, the MUX is controlled to provide to the downstream logic data coming from the correction circuit. Furthermore, the system flip-flops are hold, waiting the correction circuit to compute the correct data. The number of cycles that each flip-flop stage is hold after the activation of the error detection signal depends on the following:

1) the number m of cycles required for the correction block to provided corrected data;
2) the number k of clock cycles required for the generation of the detection signal;
3) the distance of the flip-flop stage from the read port of the memory

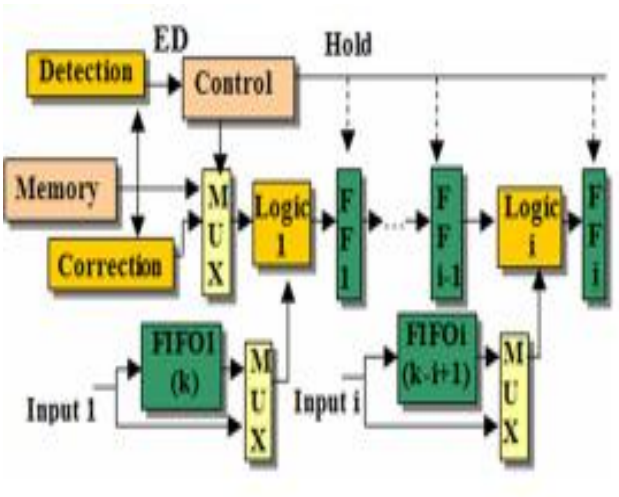


Fig. 19. Elimination of error detection and correction delays.

III.PROPOSED METHOD:

Parity generator and checker

A parity generator is a combinational logic circuit that generates the parity bit in the transmitter. On the other hand, a circuit that checks the parity in the receiver is called parity checker. A combined circuit or devices of parity generators and parity checkers are commonly used in digital systems to detect the single bit errors in the transmitted data word.

The sum of the data bits and parity bits can be even or odd . In even parity, the added parity bit will make the total number of 1s an even amount whereas in odd parity the added parity bit will make the total number of 1s odd amount.

The basic principle involved in the implementation of parity circuits is that sum of odd number of 1s is always 1 and sum of even number of 1s is always zero. Such error detecting and correction can be implemented by using Ex-OR gates (since Ex-OR gate produce zero output when there are even number of inputs).

Parity Generator

It is combinational circuit that accepts an n-1 bit stream data and generates the additional bit that is to be transmitted with the bit stream. This additional or extra bit is termed as a parity bit.

In **even parity** bit scheme, the parity bit is '**0**' if there are **even number of 1s** in the data stream and the parity bit is '**1**' if there are **odd number of 1s** in the data stream.

In **odd parity** bit scheme, the parity bit is '**1**' if there are **even number of 1**s in the data stream and the parity bit is '**0**' if there are **odd number of 1s** in the data stream. Let us discuss both even and odd parity generators.

Even Parity Generator

Let us assume that a 3-bit message is to be transmitted with an even parity bit. Let the three inputs A, B and C are applied to the circuits and output bit is the parity bit P. The total number of 1s must be even, to generate the even parity bit P.

The figure below shows the truth table of even parity generator in which 1 is placed as parity bit in order to make all 1s as even when the number of 1s in the truth table is odd.

Odd Parity Generator

Let us consider that the 3-bit data is to be transmitted with an odd parity bit. The three inputs are A, B and C and P is the output parity bit. The total number of bits must be odd in order to generate the odd parity bit.

In the given truth table below, 1 is placed in the parity bit in order to make the total number of bits odd when the total number of 1s in the truth table is even.

3-bit message			Even parity bit generator (P)
A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

3-bit message			Odd parity bit generator (P)
A	B	C	Y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Parity Check

It is a logic circuit that checks for possible errors in the transmission. This circuit can be an even parity checker or odd parity checker depending on the type of parity generated at the transmission end. When this circuit is used as even parity checker, the number of input bits must always be even.

When a parity error occurs, the 'sum even' output goes low and 'sum odd' output goes high. If this logic circuit is used as an odd parity checker, the number of input bits should be odd, but if an error occurs the 'sum odd' output goes low and 'sum even' output goes high.

Even Parity Checker

Consider that three input message along with even parity bit is generated at the transmitting end. These 4 bits are applied as input to the parity checker circuit which checks the possibility of error on the data. Since the data is transmitted with even parity, four bits received at circuit must have an even number of 1s.

If any error occurs, the received message consists of odd number of 1s. The output of the parity checker is denoted by PEC (parity error check).

The below table shows the truth table for the even parity checker in which PEC = 1 if the error occurs, i.e., the four bits received have odd number of 1s and PEC = 0 if no error occurs, i.e., if the 4-bit message has even number of 1s.

4-bit received message				Parity error check C_p
A	B	C	P	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Odd Parity Checker

Consider that a three bit message along with odd parity bit is transmitted at the transmitting end. Odd parity checker circuit receives these 4 bits and checks whether any error are present in the data.

If the total number of 1s in the data is odd, then it indicates no error, whereas if the total number of 1s is even then it indicates the error since the data is transmitted with odd parity at transmitting end.

The below figure shows the truth table for odd parity generator where **PEC =1** if the 4-bit message received consists of **even number of 1s** (hence the error occurred) and **PEC= 0** if the message contains **odd number of 1s** (that means no error).

3.1 MEMORIES with and Parity DATA

Journal of Engineering Sciences

Vol 10, Issue 8, Aug /2019
ISSN NO:0377-9254

In digital communications, a parity bit is a bit added to a binary stream to ensure that the total number of 1-valued bits is even or odd. This technique is a simple and widely used method for detecting errors. There are two types of parity bit methods, called even parity bit and odd parity bit.

The odd parity bit system consists of counting the occurrences of bits whose value is 1 in the data stream. If the number is even, the parity bit value is set to 1, so the total count of occurrences of high bits in the entire stream including the parity bit is odd. If the count of high bits is odd, the parity bit value is 0.

Figure 6. Serial data frame

Within several GreenPAK ICs the SPI block can be used to implement the serial-to-parallel conversion. The serial communication must have a 9600 baud rate.

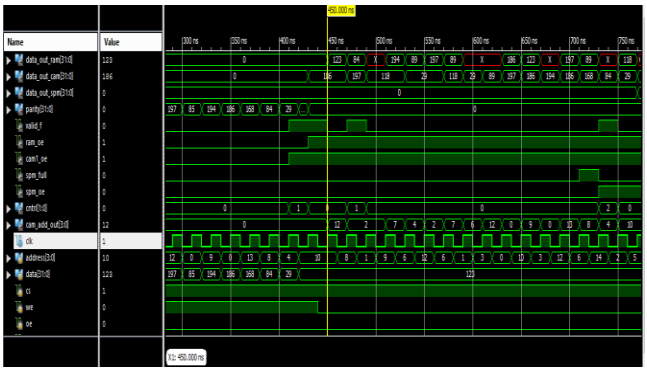
A falling edge detection is implemented to detect the start bit. When it is detected, a connection flag bit is set so two counters/delays are triggered. One of them, titled Bit Timer, is configured to have a period equal to the bit time duration (1/9600). The other counter, titled Frame delay, is configured to have a delay time equal to the 10-bit frame period (10/9600).

With these timers, the SPI block is connected so that the serial data input pin is connected to the MOSI input and the Bit Timer output is connected to CLK. The eight data bits are received by the SPI block.

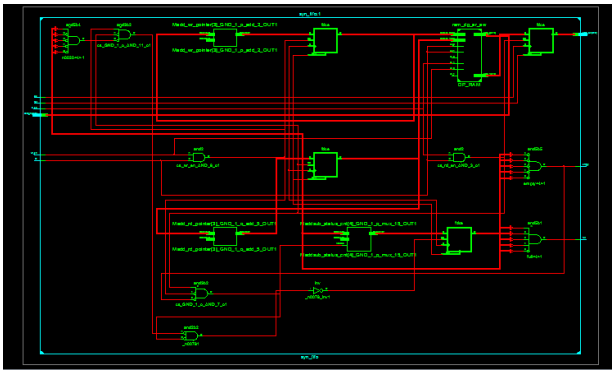
Additional logic is used for controlling the clock signal, so when the frame period has elapsed, the SPI clock stops and the data is held on the register.

IV.SIMULATION RESULTS

First and second implementation



Simulation



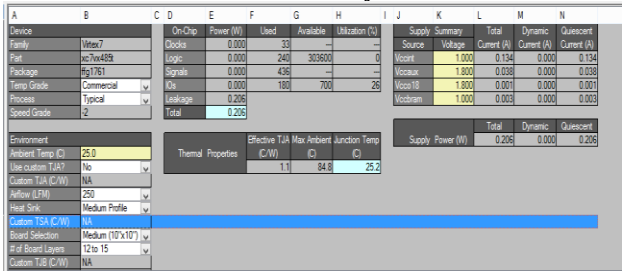
Rtl schematic

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	342	607200	0%
Number of Slice LUTs	268	303600	0%
Number of fully used LUT-FF pairs	190	420	45%
Number of bonded IOBs	180	700	25%
Number of BUFG/BUFGCTRL/BUFHCEs	1	200	0%

Design summary

```
Data Path: reset to s3/s2/DP_RAM/mem<1>_0
                                   Gate     Net
  Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
  ----------------------------------------  ------------
   IBUF:I->O           126   0.000   0.750  reset_IBUF (reset_IBUF)
   LUT5:I0->O           16   0.043   0.000  s3/s2/DP_RAM/Mmux_mem[0][7]_
   LD:D                    -0.034          s3/s2/DP_RAM/mem<1>_0
  ----------------------------------------
  Total                     0.793ns (0.043ns logic, 0.750ns route)
                                    (5.4% logic, 94.6% route)
```

Time summary



Power summary

V.CONCLUSION

We have presented several solutions to implement a rollback technique at moderate hardware cost: a fixed K-cycle rollback, or a checkpoint rollback. It was also proposed three different CAM implementations for the massive storage preservation. The K-cycle rollback with the two first CAM implementations provide very low hardware overhead compared to state-of-the-art technique by putting in common the data state preserves mechanism of several registers and the data state preserve mechanism of the

memory. However, the downside is the number of cycles to perform the rollback, as the state of-the-art technique performs the rollback in one clock cycle. In some cases, the checkpoint rollback implementation provides a lower hardware overhead in case the target architecture has not a lot of memories output but has many irregularities. To have a complete soft-error tolerant implementation, we add an advance ECC implementation technique for embedded memories. It allows using ECC without any speed penalty, except in the rare case an error is detected. Due to the large part dedicated to the embedded memories in many modern designs, the global implementation of both techniques produces a very low hardware overhead. To have a complete implementation of that technique, we should add a soft-error detection technique mentioned in the introduction. Then, we could test the fault resilient property of the whole design. In addition, an improvement of the proposed technique would concern very large designs for which the proposed rollback implementation is time-consuming even with an automated tool. Moreover, the rollback itself takes a lot of time in this kind of design. By splitting the whole design in several parts from which the rollback could be performed independently in each of them, it may improve both of those mentioned issue at a moderate hardware cost.

REFERENCES

[1] R. C. Baumann, "Soft errors in advanced computer systems," IEEE Design Test. Comput., vol. 22, no. 3, pp. 258–266, May/Jun. 2005.
[2] N. P. Rao and M. P. Desai, "A detailed characterization of errors in logic circuits due to single-event transients," in Proc. Euromicro Conf. Digit. Syst. Design, Funchal, Portugal, 2015, pp. 714–721.
[3] B. Narasimham et al., "Characterization of digital single event transient pulse-widths in 130-nm and 90-nm CMOS technologies," IEEE Trans. Nucl. Sci., vol. 54, no. 6, pp. 2506–2511, Dec. 2007.
[4] M. P. Baze and S. P. Buchner, "Attenuation of single event induced pulses in CMOS combinational logic," IEEE Trans. Nucl. Sci., vol. 44, no. 6, pp. 2217–2223, Dec. 1997.
[5] S. Buchner, M. Baze, D. Brown, D. McMorrow, and J. Melinger, "Comparison of error rates in combinational and sequential logic," IEEE Trans. Nucl. Sci., vol. 44, no. 6, pp. 2209–2216, Dec. 1997.
[6] J. Benedetto et al., "Heavy ion-induced digital single-event transients in deep submicron processes," IEEE Trans. Nucl. Sci., vol. 51, no. 6, pp. 3480–3485, Dec. 2004.
[7] M. A. Bajura et al., "Models and algorithmic limits for an ECC-based approach to hardening sub-100-nm SRAMs," IEEE Trans. Nucl. Sci., vol. 54, no. 4, pp. 935–945, Aug. 2007.
[8] T. Bonnoit, M. Nicolaidis, and N.-E. Zergainoh, "Using error correcting codes without speed penalty in embedded memories: Algorithm, implementation and case study," J. Electron. Test., vol. 29, no. 3, pp. 383–400, Jun. 2013.
[9] P. Papavramidou and M. Nicolaidis, "Test algorithms for ECC-based memory repair in ultimate CMOS and post-CMOS," IEEE Trans. Comput., vol. 65, no. 7, pp. 2284–2298, Jul. 2015.
[10] R. Vemu, A. Jas, J. A. Abraham, S. Patil, and R. Galivanche, "A lowcost concurrent error detection technique for processor control logic," in Proc. DATE, Munich, Germany, 2008, pp. 897–902.
[11] S. Ghosh, N. A. Touba, and S. Basu, "Synthesis of low power CED circuits based on parity codes," in Proc. 23rd IEEE VLSI Test Symp., May 2005, pp. 315–320.