FT-EALU: fault-tolerant arithmetic and logic unit for critical embedded and real-time systems

Athena Abdi1 · Sina Shahoveisi1

Accepted: 28 June 2022 / Published online: 14 July 2022
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2022

Abstract
This paper presents a fault-tolerant ALU (“FT-EALU”) based on time redundancy and reward/punishment-based learning approaches for real-time embedded systems that face limitations in hardware and power consumption budgets. In this method, operations are diversified to three versions in order to correct permanent faults along with the transient ones. The diversities of versions considered in FT-EALU are provided by lightweight modifications to differentiate them and clear the effect of permanent faults. Selecting lightweight modifications such as shift and swap would avoid high timing overhead in computation while providing significant differences which are necessary for fault detection. Next, the replicated versions are executed serially in time, and their corresponding results are voted based on the derived learned weights. The proposed weighted voting module generates the final output based on the results and their weights. In the proposed weighted voting module, a reward/punishment strategy is employed to provide the weight of each version of execution indicating its effectiveness in the final output. To this aim, in the method defined for each version of execution, a weight is defined according to its correction capability confronting several faulty scenarios. Thus, this weight defines the reliability of the temporal results as well as their effect on the final result. The final result is generated bit by bit based on the weight of each version of execution and its computed result. Based on the proposed learning scheme, positive or negative weights are assigned to execution versions. These weights are derived in bit level based on the capability of execution versions in mitigating permanent faults in several fault injection scenarios. Thus, our proposed method is low cost and more efficient compared to related research which are mainly based on information and hardware redundancy due to employing time redundancy and static learning approach in correcting permanent faults. Several experiments are performed to reveal the efficiency of our proposed approach based on which FT-EALU is capable of correcting about 84.93% and 69.71% of permanent injected faults on single and double bits of input data.
Keywords  Fault tolerance system · Permanent and transient faults · Embedded systems · Time redundancy · Arithmetic and logic operations

1 Introduction

Embedded and real-time systems play an important role in modern applications. These systems are utilized in many critical applications such as health care, automation, avionics, and communication. Embedded processors as the main part of these systems should function correctly and autonomously [1, 2]. In many applications, the target system is employed in critical or hardly available situations where their correct and self-determined operation is very effective on their final performance. Arithmetic and logic operations are the main part of each processing unit and play a critical role in the system’s functionality as well as reliability. Consideration of fault tolerance property in arithmetic and logic unit of embedded processors makes their operations more precise and autonomous [3–5].

Along with the technology advances and shrinkage in size of transistors, the vulnerability of systems against transient and permanent faults increases. These faults affect various aspects of systems and make their results erroneous. Computation and arithmetic units are very sensitive to the faults since their results propagate to the system output, directly. Applying fault-tolerance approaches in the computation unit is necessary to meet its correctness and autonomous requirements [5–7]. Redundancy is the main solution in designing fault-tolerant systems and could be applied in various aspects of the system. Depending on the employed level of redundancy, fault detection or correction is possible in the cost of various performance and area overheads. In the case of transient faults with a very short lifetime, detection could be solely enough leading to system restarting and recovery. Conversely, permanent faults are more complicated and remain at systems for a longer time. Thus, consideration of more complicated approaches involving correction and recovery along with detection methods is essential for their mitigation [8–11].

Time redundancy is an efficient fault-tolerance approach that replicates the system execution in time. In this approach, multiple replicated versions of the system are considered and repeat the same operation, serially. By replicating the execution of operands and comparing the results, transient faults are simply tolerated. To consider permanent faults in these methods, making variations in each replicated version of execution is beneficial and facilitates fault detection by comparing the differentiated results of computation. Detection capability is not enough for making a system fault-tolerant, where locating the faulty place and correcting it accordingly are also required. These capabilities are achieved through considering extra controlling schemes such as reconfiguration or voting. Time redundancy-based methods are efficient in terms of hardware cost compared to parallel execution and are considered as a proper choice for designing fault-tolerant embedded as well as real-time systems owing to their time predictability [1, 10, 12, 13].

Since in embedded and real-time systems, the execution time analysis is performed based on the worst-case estimation, applying some time overhead during system execution could be acceptable. Normal executions are generally shorter than
the worst-case estimation, so the saved time could be employed for the replicated executions and making the system fault-tolerant. The main mission of embedded and real-time systems could be summarized in control applications where the exact and correct computations are crucial. Computations are mainly performed in the arithmetic and logic unit (ALU) of the processors so designing them fault-tolerant is very important. In this context, applying redundancy to various aspects of the system is practical, but the resulting overhead should be studied precisely based on the requirements of embedded systems [14–17].

In this paper, we propose our fault-tolerant time redundancy-based ALU architecture for embedded and real-time systems (FT-EALU). The main purpose of this method is to mitigate the transient and permanent faults occurring in arithmetic and logic unit of embedded processors. Since most embedded and real-time systems are autonomous, their self-recovery from potential faults is required. In this context, the passive fault-tolerance approaches that perform voting among the replicated versions of the operation and do not require additional management are convenient. Transient faults are detected by comparing the results of replicated versions of computation and generally corrected in response to performing a simple voting mechanism among them due to their short remaining time in the system. Reversely, to correct permanent faults, repeating the execution is not enough due to their long lifetime, and mitigation approaches including fault locating and recovery are also required. Moreover, by replicating the same version of the execution, permanent faults will not be detected, and applying variations to each version is necessary to make a diversion in the final results. In our proposed approach, simple and low overhead modifications of input data such as shift and swap are considered to avoid hardware costs and establish sufficient diversity in input data to mitigate permanent faults. Due to the effect of permanent faults, by diversifying the various versions of computation, different results are generated which could be mitigated during the voting process. Traditional majority voting schemes are generally not efficient for mitigating permanent and inclusive faults; so employing an appropriate weighted voter which identifies the trusted results and magnifies their effect on the final output would improve the efficiency. Experimental results show that our proposed FT-EALU method is capable of correcting 84.93% of injected permanent faults in arithmetic and logic operations. The main contributions of our proposed fault-tolerant approach could be summarized as follows:

- Presenting a time redundancy-based fault-tolerant approach to correcting transient and permanent faults of computation in arithmetic and logic unit of embedded systems;
- Presenting an efficient weighted voting approach based on reward and punishment to detect and cover the effect of faults in execution versions and isolate them from the final result;
- Employing an autonomous learning-based approach to define the appropriate weight for each bit of the results in the proposed voting process based on various fault injection scenarios;
- Presenting an adjustable architecture to set the system detection and correction capability proportional to its defined design costs.
The rest of this paper is organized as follows: The related works are reviewed in Sect. 2. In Sect. 3, the details of our proposed fault tolerance architecture including the detection and correction approaches are explained. The experimental results and evaluation of FT-ALU are provided in Sect. 4. Finally, the conclusion remarks are mentioned in Sect. 5.

2 Related work

The arithmetic and logic unit is one of the most critical elements of embedded systems that directly affect the correctness and reliability of the system output [5, 14]. The fault-tolerant design of this component is very important and effective on system reliability. Based on the applications of embedded systems, their reliability and autonomous operation are very important. In this context, several studies concentrate on fault-tolerant computation and ALU operation that are based on the redundancy at different levels and aspects. Redundancy at ALU could be defined at information coding [15, 16, 18], hardware modules [4, 5, 19], or time repetition [20].

Information redundancy-based methods add error detection/correction codes to data to protect them against potential faults. Here, encoding and decoding modules are required because, in ALU, the focus is on computation, so the value of input data is important and should be unchanged to have the correct result. Various methods based on parity, residue, and Berger codes are proposed in this category. In [16], a fault-tolerant architecture for embedded processors based on employing Berger code is proposed. This method mainly tolerates transient faults and expands the operands by a logarithmic factor of their length. Extracting the raw and intended output from its encoded version requires decoding hardware that leads to hardware overhead which is hardly acceptable in embedded and real-time systems due to their limitations. In [21], a fault-tolerant architecture for ALU and adder based on adding parity codes is presented. In this approach, the hardware cost caused by parity codes is low, but the fault handling approach requires considering an extra ALU which is not applicable in most embedded systems. Moreover, in [18], reversible logic gates are employed and the ALU is protected by parity codes. Generally, information redundancy-based methods are not appropriate for arithmetic circuits because it is probable that the added codes modify during the computation in ALU and lose their efficiency.

In [22], a fault-tolerant approach based on residue codes is proposed. In this study to overcome the high hardware overhead of previous research, a new residue checker circuit is proposed by considering more compression. However, the final hardware overhead decreases, but still it is high for embedded systems. In [23], to make the ALU fault-tolerant, parity gates are employed which have limited capability in fault tolerance and are suitable to detect single-bit faults. Hardware redundancy-based methods enforce extra hardware to the embedded systems that are not acceptable in many applications where the area and performance of these systems are critical. In [15], a fault-tolerant design of ALU using enhance Berger code is presented. The focus of this approach is on reducing the hardware and performance overhead of Berger code and its effectiveness in mitigating the
permanent faults of ALU is studied. Moreover, in [19], an information and physical redundancy approach are applied to the processor to avoid the error propagation. Employing triple module redundancy in ALU and CU along with applying the Hamming code to register files is the main fault-tolerance approach of this research.

In [4], a fault-tolerant ALU architecture for embedded systems is proposed. In this approach, the unused sections of the ALU system are utilized for redundant computations and make the system fault-tolerant. This method shows good performance in terms of area and fault correction abilities but still adds extra hardware and delay to the system. In [5], a narrow width value-based method for handling hard faults of ALU is presented. In this method, two half-word ALUs are employed to detect system faults. After fault detection, each ALU is partitioned to locate and isolate the occurred fault. This approach uses the unused parts of the system to perform operations in parallel, but its detection and isolation capabilities are limited based on the system capacity.

Employing duplication with comparison (DWC) and triple duplex redundancy (TMR) is effective approaches for fault detection and correction, and these methods are employed along with various re-execution and redundancy approaches. Employing these methods in hardware redundancy-based approaches leads to high cost and power consumption which is not acceptable in embedded and real-time systems. Time redundancy techniques decrease the hardware cost but prolong the execution time that could be managed in most real-time applications due to their WCET estimation. Convenient time redundancy methods such as RESO, RESWO, and REDWC [24–26] consider various modifications in different versions of computation to make them different and improve the detection capability. In [20], a TMR-based fault tolerance method based on considering the basic time redundancy approaches such as RESO [24] is presented. The considered voting approach of this study is designed based on majority voter. This majority voting scheme is the most utilized approach that tolerates the faults passively.

Moreover, in [27], a hardened ASIC-based ALU for space applications based on employing duplication and comparison in computation results is presented. Here, the results are compared by an proposed ELECT block that corrects the final output of the system. Space applications are also considered in [28], and a TMR-based physical approach to harden ALU is presented in this research. These methods are effective in fault detection at low cost and overhead, but their coverage is improvable. Figure 1 summarizes the explained related approaches in three main categories based on their redundancy aspects. In this figure, important approaches along with their fault tolerance schemes are mentioned. Here, pink boxes are the passive fault-tolerance approaches that utilize majority voting to cover the effect of errors. Blue boxes represent employing the compare and vote to derive the final output and green box shows utilization of the learning-based voting scheme. As this figure shows, our proposed FT-EALU is a time redundancy-based approach that employs learning to protect ALU against permanent and transient faults more efficient. These voting approaches are studied and compared later in Sect. 3.3.
3 Proposed method

The design of reliable and autonomous embedded as well as real-time systems is very important due to their wide applications. Fault tolerance approaches are defined at different levels of system design and manage the occurred errors and keep them away from the system output. This section describes our defined problem, and the details of our proposed FT-EALU approach are presented.

3.1 Architecture of FT-EALU

Applying redundancy in a system realizes the active or passive fault tolerance feature. Active fault-tolerance approaches have fault detection, localization, and isolation/correction phases. Implementation of active methods makes some delays in the system due to the required system reconfiguration. Conversely, passive methods cover the fault from the system output and do not make any interruption in its operation. Our proposed fault-tolerant architecture is based on passive approaches and the learning-based voting among multiple versions of operations. Time redundancy-based approaches are the proper choice for embedded and real-time systems due to their worst-case time analysis during the design, generally leading to some slacks in the runtime execution. Our proposed method hardens the ALU against transient and permanent faults by employing enhanced time redundancy-based approaches and performing learning-based voting among them to isolate the errors that occurs from the system output.

Fault detection is the main prerequisite of the fault tolerance process. When this step is performed correctly, the fault tolerance method will be more efficient. Due to the low durability of transient faults, it is assumed that they are mitigated during sequential executions of time redundancy-based approaches [29]. Further, time redundancy-based fault Fun approaches diversify the operations through simple modifications to enhance their detection capability when confronting permanent...
faults. These modifications include simple actions such as complementing the operands, shifting or swapping them causing variations among original and replicated executions’ results to improve the detection coverage. Thus, in our proposed fault-tolerant approach, we utilize the original and modified versions of input data based on the mentioned enhanced methods as the basic blocks of the system.

First, our proposed method employs multiple versions of execution with diversifying operands and performing voting among them. To this aim, three various versions of input data are taken into account in FE-EALU to perform the considered serial operations: (a) raw operands (b) left-shifted operands [24] and (c) left-shifted and swapped operands [30]. Then, the generated results of three versions of executions are adjusted to eliminate the effect of shift and swap.

To improve the detection capability, in our proposed approach, the potential carry of arithmetic operations is also considered and adjusted after modifications. Use of simple operations such as shift and swap would enforce less overhead to the system while distributing the effect of stuck-at faults, making their detection very probable. It is possible to consider different variations on input data, but based on our experiments, consideration of three selected versions of execution would lead to the most proper correction capability. This is because their final results have less similarity to each other, whereby more shuffling is applied to input data making the effect of permanent fault occurrence clearer in voting. The effect of other variations of input data on final output is studied in detail and reported in Sect. 4.

Afterward, to cover the effect of permanent faults that are resistant to repeated executions from the final output, voting is required among the results of defined operations on various input data. Majority voting is the simplest choice though its efficiency is limited due to the probability of wrong result repetition and its propagation to the final output. To overcome these issues, weighted voting is appropriate, but its efficiency is heavily dependent on its required weights. In FT-EALU, to specify the weights, we employ a trust factor parameter based on the correction coverage of each version of execution through learning. In this context, we propose two efficient weighting approaches that are based on reward/punishment, depending on the function of execution operation confronting the injected permanent faults. The first weighting approach is punitive, and the second one employs the combination of reward and punishment in the evaluation and determining the trust factor. These weights are determined statically by studying the correction capability of various operation versions facing several injected permanent faults. As the result of this phase, a score is assigned to the results of various operation versions. This weight is utilized in the proposed voting process that is explained in Sect. 3.2 in details.

In the proposed approach, the granularity of assigned weights is considered as fine as single bits to make it more precise. The architecture model of our proposed fault-tolerant approach is shown in Fig. 2. In this figure, $W_{ij}$ shows the computed weight of $j$th version of execution for $i$th bit of the result ($result_{ij}$). The various versions of execution in Fig. 2 perform identical operations but on various operands that apply sequentially in time on a unique hardware platform. Employing various operands improves the permanent fault detection and correction capabilities, respectively. The details of determining the weights of each execution and the employed voting mechanism are explained in the following subsection.
3.2 The proposed weighted voter architecture

To determine the weights of various executions in the voting process, we propose and study two strategies based on reward and punishment. The details of these schemes are presented in this part.

3.2.1 The first proposed weighting approach

Our first proposed weighted approach determines the weight of each execution (trust factor) based on a punitive strategy. Here, the detection capability of considered executions confronting stuck-at faults in different bits of operands is examined. Since the operations are performed on various forms of operands, the effects of the fault and its propagation on different bits of the data are not the same. Thus, we define the weight for each bit of operands in execution versions based on their ability to correct the injected stuck-at faults.

In this proposed punitive weighting approach, the score of ideal executions that cover the injected stuck-at fault does not change and the one(s) that propagate the error receive a negative score. This score is to each bit of operand and for each fault scenario, the unit negative score (“− 1”) divides among all versions of operation that are incapable of correcting the injected fault. This score is a representation of punishment that reduces the trust factor and the effect of the unreliable executions in the final output, consequently. In the other words, the score of unreliable execution is more negative and has less impact on deriving the final result of the system.

The total weight of each bit of operands in every version of execution is computed from its scores assigned in various fault injection scenarios. In this context, the total score of every single bit of input data, from all fault injection scenarios, is divided by the considered faulty cases and then normalized. The normalization process scales the weights within a specified range and it is possible to consider various normalization approaches for deriving the final weights. Here, we study three popular normalization approaches that are compared in Sect. 4 among which standard normalization scheme had the best results. Since standard
normalization provides more scattered weights, it separates the results properly and reflects their unique effect on voting.

Afterward, for each execution version, every bit of the result is multiplied by its corresponding weight determined through the punitive approach and passes to the voter. In the voter, the received results from three versions of execution are added together bit by bit and then divide by the total weights of the target bit in various versions. Then, the derived result of the defined weighted average scheme should be interpreted. Due to the normalization process and final division, the final output of the voter is in the range of “zero” to “one.” To map this bit-level result to binary values, we consider a threshold based on the performed operation. In this context, if the output of voter is greater than or equal to “0.5,” it will be considered as one, otherwise, its value will be specified as zero in the binary scale. Here, the value greater than “0.5” represents the case in which the operations with more than half of the total weights of the system indicate that the final result should be one and vice versa. Thus, based on their domination in weight and the defined trust factor parameter, the most reliable versions of execution determine the final output of the system. It should be noted that other values of threshold are also studied; however, based on the mentioned explanations, the selected bound leads to the best results.

The defined punitive weighting approach is somehow strict and biased which assigns the scores greedily in one direction. In this approach, the weights are determined independent of the complexity of occurred fault, and the weight of reliable execution is always zero. Thus, the effect of reliable versions is always constant in the final output and is not determinant.

3.2.2 The second proposed weighting approach

Our second weighting approach is more efficient and defined as a combination of reward and punishment. Here, the assigned scores are negative for faulty cases (punishment) and positive for the corrected ones (reward). Further, in this approach, the scores are dynamically adapted based on the complexity and hardness of the faulty scenarios. To this aim, the maximum score of each faulty scenario, “+1” is divided among the execution versions that can mitigate it. For instance, if only one of the considered versions of execution is capable of correcting the injected fault, its corresponding score will be the highest, “+1,” but in the case of two or three simultaneous fault corrections, this score is divided by two or three, respectively. Thus, the scores are set in proportion to the reliability of executions and the complexity of faulty scenarios. The same procedure is also considered for punishment, and the complicated scenarios with hardly correctable errors have less punishment. The positive or negative score of each operation shows its assigned reward or punishment based on its fault correction capability.

Here again, after determining the scores of each bit, their corresponding weights are generated through normalization and division similar to the procedure already explained earlier for the first weighting approach (see Sect. 3.2.1). The weights are determined for each bit of data to make the voting process more precise. These weights are extracted statically and at the design-time through a learning approach considering various permanent fault scenarios.
Figure 3 shows the details of our proposed weighting process based on a reward/punishment learning scheme through several fault injection scenarios.

Figure 3 shows the details of our proposed reward/punishment weighting approach. In this figure, the faulty scenarios are saved in fault injection storage and are applied to the input data during the training phase. Moreover, $\epsilon R_i \epsilon$ represents the intermediate results generated by the execution versions. At the end of the fault injection process, the final integrated scores are applied to the normalization box, and the weights are computed.

As this figure shows, based on comparing the results of each execution version to the correct output, its score is updated through a reward/punishment scheme. The reward of corrected results is determined as a positive score based on the complexity of the faulty situation. Similarly, the punishment is applied to the faulty results to decrease their score and affect the final output of the system. After applying all considered faults, the integrated final score of various executions is normalized and specified as the weight of each replicated operation.

It should be noted that the explained voting process is mainly implemented in software and enforces very limited complexity to hardware. Since the execution versions are distributed in time, they are implemented on the same hardware platform. Moreover, the weighted voting process is performed in software. However, if the hardware platform of the target system consists of redundant modules that could be utilized during the voting process, it will be appropriate and reduce the performance overhead. Since the target application of our proposed approach is embedded systems, extra hardware does not apply to them due to the cost and space constraints. Besides, extra hardware modules make the system more vulnerable to potential permanent faults. Thus, the hardware complexity of our proposed FT-EALU is very limited and could be expanded based on the requirements of the target system.

The details of the defined voting process and our proposed fault tolerance approach (FT-EALU) are presented in algorithm 1. In this algorithm, “F” and “A,B” demonstrate the operations and input operands that are considered in various versions based on the proposed method. Result of each execution version after adjusting based on the employed modifications is shown with $\epsilon R_V \epsilon$. Moreover, the shift and concatenation operations are presented by $\epsilon >> \epsilon$ and “|”, respectively. The trust factor of each execution version based on the proposed learning-based weighting approach is saved in “Score(t)(Vxi)” based on the injection.
scenarios and each bit of result in various versions of execution. The standardized scores lead to execution weights that send to the voter to generate final result.

**Algorithm 1**: Pseudo-code of our proposed time redundancy-based fault tolerance approach (FT-EALU)

```
// Replicate any operation three times in three various versions
- Operation (F) on original input data (V1):
  \( R.V_1 = F(A,B) \)
- Operation (F) on left shifted input data (V2):
  \( AS = A \ll 1, BS = B \ll 1 \)
  \( RS.V_2 = F(AS,BS) \)
  \( R.V_2 = RS.V_2 \gg 1 \)
- Operation (F) on left shifted and swapped input data (V3):
  Halve the input data to AL, AH, BL, BH
  \( ALS = AL \ll 1, AHS = AH \ll 1, BLS = BL \ll 1, BHS = BH \ll 1 \)
  \( RHS.V_3 = F(ALS,BLS), RLS.V_3 = F(AHS,BHS) \)
  \( RH.V_3 = RLS.V_3 \gg 1, RL.V_3 = RHS.V_3 \gg 1 \)
  \( R.V_3 = (RL.V_3 | RH.V_3) \)

// Perform the proposed weighted voting among the results
- Determine the corresponding score of each bit based on the proposed reward-punishment approach
  for \( t=1:N \) (number of fault injection scenarios)
  for \( i=1:input\ data\ length \)
    if (three results are correct)
      \( score(t)(V1_i) = score(t)(V2_i) = score(t)(V3_i) = +\frac{1}{3} \)
    else
      if (two results are correct (i.e. \( V1, V2 \) and one is faulty (i.e. \( V3 \))
        \( score(t)(V1_i) = score(t)(V2_i) = +0.5, score(t)(V3_i) = -1 \)
      else
        if (two results are faulty (i.e. \( V1, V2 \) and one is correct (i.e. \( V3 \))
          \( score(t)(V1_i) = score(t)(V2_i) = -0.5, score(t)(V3_i) = +1 \)
        else
          \( score(t)(V1_i) = score(t)(V2_i) = score(t)(V3_i) = -\frac{1}{3} \)
    end
  end
- Determine the normalized weight of each bit of input data in various versions of operation
  \( W_i(V1) = \text{Standardize} ( \frac{\sum_i score(t)(V1_i)}{N} ) \)
  \( W_i(V2) = \text{Standardize} ( \frac{\sum_i score(t)(V2_i)}{N} ) \)
  \( W_i(V3) = \text{Standardize} ( \frac{\sum_i score(t)(V3_i)}{N} ) \)
- Performing the weighted voting among various versions of operation to generate final output
  for \( i=1:input\ data\ length \)
    total\ weight_i = (W_i(V1) + W_i(V2) + W_i(V2))
    Result \( (i) = \frac{W_i(V1) \ast R.V1_i + W_i(V2) \ast R.V2_i + W_i(V3) \ast R.V3_i}{\text{total weight}_i} \)
    if (Result \( (i) \geq 0.5 \))
      Final Result \( (i) = 1 \)
    else
      Final Result \( (i) = 0 \)
  end
```

### 3.3 Illustrative example

To clear the workflow and details of our proposed fault-tolerance architecture (FT-EALU), in this section, we apply it to an illustrative example. In this example, we focus on a four-bit input data with “stuck at one” permanent fault and the “add” operation as one of the key functions of ALU. As Fig. 4 shows, in this example, the injected fault affects the second bit of data and makes it corrupted. Moreover, to clarify the ability of this method in mitigating transient faults, a temporal fault on the fourth bit of the first execution version. Due to the short duration of transient
faults, the effect of this injected fault is reflected on a single execution version. Here, based on the explained approach, input data are applied to three distinct versions of operations: the original data, the shifted data, and the swapped-shifted data. Then, the add operation is performed on the prepared input data and the replicated versions are executed in serial. In this figure, the yellow highlighted bits represent the faulty ones due to the permanent injected fault and the bit affected by the transient fault is shown in blue highlight. Here, the result of each execution is passed to the voter, separately and the effect of shift and swap operations on data is shown by purple bits. Based on the explained process, in FT-EALU, the corresponding weight of each bit of data is prepared through the learning scheme and based on its capability in mitigating the injected faults.

In this example, the results of the majority and our proposed weighted voters are computed and compared. In the considered case, the final result of the second and third executions of this example is the same and the majority voter obeys them mistakenly. Nevertheless, in our proposed weighted voter, determining the score of each bit of results based on the defined trust factor leads to the correct result that covers the effect of injected fault, accurately. In this example, the assigned weight of each bit of result is computed and normalized based on our proposed scoring algorithm and shown as \( W \) vector in the figure. The final result is computed from performing a bit-wise weighted sum on the assigned weights and results of each version of execution.

As this example shows, our proposed FT-EALU is able to mitigate transient and permanent faults by determining the role of each bit in final result. The overhead of our proposed fault tolerance approach is mainly related to the execution time of various versions of execution. Since the weighting process is mostly performed statically, it does not prolong the final execution time. If the system hardware allows, modifying the input data for each version of execution could be performed in parallel with other executions. Eventually, this time overhead is managed by utilizing the extra time of the estimated deadline in real-time applications.

Fig. 4 An illustrative example showing the details of our proposed fault-tolerance approach on a tiny sample case. Yellow highlighted bits are affected by permanent faults and the blue highlighted one is flipped due to transient fault
It is to be noted that our proposed \textit{FT-EALU} approach could be adaptively extended to five versions of operations and perform them voting among them like a 5MR system. In this case, the correction ability increases in the cost of more time overhead caused by employing more versions of execution. Adjusting the appropriate overhead and fault coverage levels is related to the application requirements and could be adaptively set by the designers based on the target system characteristics.

4 Experimental results

In this section, to demonstrate the effectiveness of our proposed fault-tolerance approach in making the ALU of embedded systems more reliable, several experiments and comparisons are presented.

4.1 Simulation setup

To simulate and evaluate the proposed fault mitigation approach, our considered underlying platform is an ALU which implements basic operations such as “and,” “or,” “xor,” “not,” “add” and “subtract.” The considered input data length is 4 and 16-bit where a comprehensive set of stuck-at faults is injected into them. Since our target underlying platform is an embedded system, the considered data length is appropriate and close to real cases, yet it could be easily extended to any other input length and fault pattern based on the target application requirements.

Our simulation platform implements the complete system architecture and focuses on permanent stuck-at faults that corrupt one or two bits of operands forever. The transient faults are fully mitigated during the time redundancy-based fault-tolerant approaches due to their short lifespan. Thus, we exclude transient faults from the experiments and focus on permanent ones. Since we study the effect of permanent faults on ALU and at the instructions level, they mainly affect input data or target operation. In both cases, the occurred fault modifies the value of data bits permanently. Stuck-at faults have a persistent effect on data bit and are proper to model the effect of permanent faults as they are employed in related research [4, 20, 30]. In the following experiments, all permutations of single and double bit stuck at faults on 4-bit input data are considered. Further, all possible stuck-at faults (0 and 1) are applied to evaluate the proposed approach comprehensively. Since considering all possible 16-bit numbers and applying all single and double bits stuck-at faults on them make the design space very large (2^{21} various cases for single bit stuck at faults) which is not feasible in our simulation platform, we have limited them in our experiments. Here, selecting a limited number of random inputs is not representative so to make our experiments more precise and extended, we have selected 100 random 16-bit numbers ten times (based on various randomness criteria) and applied all possible stuck-at faults on them. Our investigation shows that in this case, 100 random numbers are an appropriate choice which reflects the intended properties of the system. Figure 5 shows that by increasing the number of selected 16-bit numbers, the correction capability pattern of our proposed FT-EALU converges. The elbow
point of this figure is in 100 point, and we consider it in our experiments. It should be noted that, due to applying all possible stuck-at faults on the considered data set, its size grows dramatically. Thus, the processing power of our underlying platform does not let us extend this study to more than 300 samples. However, the efficiency of our choice could be concluded from the final constant pattern of this figure.

As this figure reveals, selecting 100 random 16-bit samples is representative and reflects the correction capability of our proposed method appropriately. The proposed approach and the simulation platform are implemented by Python. In this context, a high-level model of ALU that is capable of performing the main logical and arithmetic operations is simulated to which various combinations of faulty input data are applied. The simulated platform is implemented in Jupyter Notebook environment on an Intel Core i7 CPU with 8 GB RAM. During the simulation, we have implemented the considered operations of ALU, where some built-in modules such as math, random, and copy are utilized.

4.2 Experiments and discussion

To demonstrate the effectiveness of our proposed fault mitigation approach and compare it to the related research, in this section, three distinct categories of experiments are considered as: (I) selecting the most proper variations of input data to enhance the correction capability of our proposed fault-tolerant approach by providing more variations in executive versions and covering the fault more efficiently, (II) evaluating the efficiency of our proposed voting approach in covering the permanent faults and determining the appropriate weights statically or permanently based on a
learning-based strategy, (III) assessing the overhead of our proposed approach and comparing its correction capability to the related fault-tolerant approaches.

A. Selecting the proper combination of execution versions for the replicas

The first experiment aims at determining the most proper combinations of input data variations that lead to the best correction coverage. As explained earlier, in time redundancy-based approaches, it is required to make some variations in the replicated blocks of the system to establish appropriate diversity to detect permanent faults. Low-cost and lightweight operators are the best choice for making the required variations and improving the correction ability. Meanwhile, the precise selection and combination of replicated versions are very important. Consideration of the similar operands in various versions of execution leads to repeated operations and analogous results which would reduce the correction ability during the voting process. To study the effect of combining different versions of execution in our proposed time redundancy-based approach, we evaluate it with various combinations of diversified operations. The modifications are mainly based on applying simple and low-cost operators such as shift, swap, and rotate on the input data of the operations.

In this context, we consider seven distinct versions of operands for each operation: original input data, shifted input data (RESO) [24], swapped input data (RESWO) [25], enhanced shifted input data (E-RESO) [30], shifted and swapped input data (E-RESWO) [30] and two rotated input data approaches. These modifications of operands in replicated executions are required to preserve and propagate the effect of injected faults in their corresponding results. In this way, the results of replicated executions are not similar and the occurred fault will be masked during the voting process.

In our TMR-based approach, the first version of execution is performed on original input data and the mentioned variations are applied to the replicated operations. Based on applying the mentioned candidate variations on input data of the replicated execution versions, six meaningful cases are formed. Here, applying similar variations such as shift and rotate in combinations has not been considered due to their analogous results which would reduce the correction ability. The correction capability of the defined combinations of executions is presented and compared in Table 1. The reported correction coverage of this table is computed in average for all considered arithmetic and logical operations of our simulated ALU.

| Correction approach | Correction coverage |
|---------------------|---------------------|
|                     | 4-bit   | 16-bit   |
| Basic operands  RESO RESWO | 79.06% | 80.17%  |
| Basic operands E-RESO E-RESWO | 60.28% | 54.41%  |
| Basic operands  Rotated RESO E-RESWO | 78.68% | 80.93%  |
| Basic operands  Rotated E-RESO E-RESWO | 43.99% | 59.76%  |
| Basic operands  Rotated E-RESO RESWO | 37.45% | 32.33%  |
| Basic operands  RESO E-RESWO | 80.35% | 82.38%  |
As this table shows, combining different versions of executions in replicated versions leads to various correction capabilities. Based on this table, combining the basic operands with their shifted (RESO) and shifted-swapped (E-RESWO) versions leads to the best correction coverage. Thus, our proposed FT-EALU considers this combination of execution that provides appropriate diversions in the results. As the result of this diversion, the effect of injected fault is kept in the intermediate results and then covered successfully during the voting process. Further, based on the presented results, the correction coverage for 4 and 16-bit input data is almost consistent, which confirms the extension capability of this study. It should be noted that the correction coverage is not the same for all ALU operations. The coverage of arithmetic operations is lower than that of logical ones due to their complexity. Figure 6 shows the correction capability of the studied combinations of Table 1 categorized to standard benchmarks of ALU. In this figure, the correction approaches are presented in the order of Table 1.

As this figure shows, the effectiveness of correction approaches in mitigating faults of logical operations is better than that of the arithmetic functions such as add and subtraction. That is because, arithmetic operations are more complex and there is a potential of error propagation due to the carry and borrow transfer among data bits. Our proposed fault-tolerance approach, considered as “Correction Approach 6” in this figure, reveals the best coverage capability in all ALU operations due to its proper variations of input data and considering the effect of the carry bit on data swap and shifting. It should be noted that the presented results of this figure are based on 4-bit data, and the same pattern is valid for other data sizes based on our experiments. Further, in this experiment, the basic

![Fig. 6 Correction capability of various combinations of diversified operands categorized by ALU operations as benchmarks](image-url)
majority voter has been employed to show the effectiveness of various combinations of input data for different ALU operations.

B. Evaluation of the proposed weighted voter

The second experiment aims at demonstrating the effectiveness of our proposed weighted voting approach. Until now, we have used majority voting approach in our experiments. Sometimes in complex cases, various versions of executions propagate the errors to the same position of the system output, so consideration of majority voting is unreliable. Our proposed weighted voting approach evaluates the fault detection capability of various versions of computation and determines a weight for each of them statically. This weight represents the trust coefficient during the voting process and is determined per bit of input data. In this way, the corresponding results of various versions of executions are not the same for the voter, and their effect on final output is determined based on their trust factor. In this experiment, the fault tolerance ability of a conventional TMR that utilizes majority voters is compared to that of our proposed weighted voter’s schemes. The weights of various executions in our proposed voter are determined statically through a learning-based approach. This weight represents their capability in error detection which is derived from their effectiveness in various cases. As explained in Sect. 3.2, the first weighting approach is designed based on punishment of the incapable executions that propagate the errors, while the second scheme employs reward and punishment schemes simultaneously. More capable approaches obtain higher weights, and their corresponding result is more effective in final output due to their higher trustworthiness. Figure 7 shows the robustness of the considered execution versions in FT-EALU method separated by each bit of data for 4-bit input samples. In this figure, the average effect of single and double bit stuck at faults on input data during

![Graph showing robustness of different bits of various versions of input data against single and double stuck at faults during add operation (Bit 0: LSB and Bit 3: MSB)](image)
the add operation is studied. The effectiveness of various versions of execution in logical operations is very high and close to each other. Thus, to show the different trustworthiness of input bits more clearly, this experiment focuses on add operation which is more complicated.

Based on this figure, the robustness of different bits of input data against injected faults is not the same. Since some faults are covered during the operation and do not propagate to the output. These faults are not our target in our proposed fault-tolerant approach, but they improve our system reliability. We try to focus on uncovered faults, but the correction capability of each version of execution could be an appropriate guide for determining our target trust factor. Based on Fig. 7, our considered execution versions in FT-EALU provide various robustness on input data and this could be utilized as the trust factor of each bit of result, laterally. The reason behind this difference in robustness of input data bits is the effect of their employed variations. For instance, the most vulnerability of the shifted and swapped input data is in bit 2 that has the constant zero value due to shift operation. This constant value makes the data more vulnerable against stuck at faults. The presented results of this experiment verify that the reliability of different execution versions is not the same so applying them in the final result with various weights could be beneficial in fault correction.

As already explained in Sect. 3.2, in this research, we propose two weighting approaches for the voter unit. These weights are generated by a learning-based method and then normalized to apply during the voting process. In this context, we consider various efficient normalization approaches and study their effects on the correction capability. Accordingly, the correction capability of our proposed voting approach in both proposed weighting schemes is examined considering three different normalization methods: (1) linear (min-max) normalization, (2) adding the weight of each bit of data to the absolute minimum value of that bit in all experiments and (3) standard normalization. The results of this study for all operations of our considered ALU and the first and second weighting scheme are presented in Table 2, averagely.

As this table shows, employing the weighted voter and increasing the effect of more reliable executions in the final output lead to better correction capability than majority voting. Based on the results of this table, the first proposed weighting approach is less efficient in correcting permanent faults than the second proposed method. Since the punishment-based scheme only reduces the effect of unreliable

| Table 2 Correction capability of our proposed weighted voting approaches: punishment-based and punish-reward weighting schemes normalized by various normalization methods |
|-----------------------------------------------|
| Correction coverage                         | Normalization 1 | Normalization 2 | Normalization 3 |
| Proposed voting scheme 1 (Punishment-based) | 81.76 %         | 78.22 %         | 82.57 %         |
| Proposed voting scheme 2 (Reward-Punishment)| 82.37 %         | 80.23 %         | 83.61 %         |

© Springer
executions and adjusts the final output in one direction. Assigning negative scores to unreliable results decrease their effect on the final output partly. Since keeping the scores of reliable versions constant in the final output makes the final improvement limited. Meanwhile, the punishment/reward weighting approach adjusts the weights in both incremental and decreasing directions and reinforces the effect of reliable results in final outputs. It increases the scores of reliable results while reducing the effect of unreliable ones simultaneously where the weighting process is performed more efficiently. Thus, in FT-EALU, the second weighting scheme along with the standard normalization is selected.

Until now, the corresponding weights of each execution have been determined dynamically according to the system feedback. Since our target application is autonomous and real-time embedded systems, it is important to reduce the performance overhead and estimate the weight of each operation statically. As explained earlier, the weighting process could be more efficient and performed statically through a learning-based approach. To this aim, the corresponding and proper weights for each bit of various versions of execution are determined based on several random stuck-at faults. To show the details of this learning approach and determine the appropriate weights for each bit of operands, in this experiment, a dataset including several randomly stuck at faults along with the errors that aim some vulnerable bits of data is considered. Vulnerable bits are the ones that are at risk due to our considered variation techniques. For instance, the middle bits of data that are affected by the “swap” operations or the MSB and LSB that are modified by “shift” are considered vulnerable bits. Propagating the stuck-at faults on these vulnerable bits to the final output is more probable, and their correction is harder. Our considered dataset for the learning approach is formed of several random input data and various injected stuck at faults. During the learning process, the dataset is ten-folded and nine parts are employed for training and the remaining onefold is used to test the extracted values. Table 3 shows the correction capability of FT-EALU based on the weights that are extracted through the described learned process. In this experiment, the weights are computed by the reward/punishment scheme due to its efficiency. It should be noted that in this experiment, several random and specified stuck-at faults on single and double bits of 16-bit input data are considered. To perform the learning approach more efficiently, the dataset should be as large as possible and contain several faults. By considering more samples, the training phase of learning is implemented more efficiently. Thus, in this experiment, we expand our dataset to 400 random 16-bit data samples, and all permutations of single and double bits stuck-at faults are applied to it. Moreover, in Table 3, the effect of defined normalization schemes is also studied.

| Normalization 1 | Normalization 2 | Normalization 3 |
|-----------------|-----------------|-----------------|
| 1-bit SA        | 2-bit SA        | 1-bit SA        | 2-bit SA        | 1-bit SA        | 2-bit SA        |
| FT-EALU         | 82.09%          | 68.83%          | 83.42%          | 68.88%          | 84.93%          | 69.71%          |

Table 3 Correction capability of FT-EALU based on the static weights of voter determined by a ten-folding cross-validation approach
As this table shows, the extracted weights provide appropriate correction coverage and are applicable to utilize as predefined weights in other unknown scenarios. Based on the results, the standard normalization has the best efficiency and could be integrated with our proposed weighting approach. These results are extracted based on various cases and could be simply utilized in other data lengths and various applications.

To indicate the efficiency of our proposed learning approach in determining appropriate weights, its correction capability is compared to the results of Table 2 which are based on the method that extracts the weights directly from the input data. By applying the extracted weights of learning to the input data of the previous experiment, the correction capability differs by about 0.51%. Thus, our learning-based weighting approach could be efficiently employed in any application by its predefined weights. This process reduces the performance overhead of our proposed fault tolerance approach dramatically and makes it more applicable for real-time applications.

C. Comparison of FT-EALU to related approaches

The last class of experiments compares the correction capability and performance overhead of our proposed fault-tolerance approach to the related research. To this aim, first, the information redundancy-based method of [15], hardware reuse approach of [4], and TMR-based method of [20] which utilizes the basic time redundancy schemes are considered. In [15], the fault mitigation of ALU is performed through an enhanced version of Berger code. This method modifies the operations and operands to show the faults better during the computation. In [4], a hardware redundancy-based approach to duplicate the operations is considered and its cost is moderated through the idle module reuse scheme. The time redundancy-based approach of [20] protects ALU through replicated operations in time and performs a majority voting among the results.

The comparison results of our proposed FT-EALU to these approaches in terms of correction capability and the performance overhead are presented in Table 4. Since the reported results of the considered related studies are mostly based on addition, in this experiment, we have limited the ALU operation to “add.” Moreover, the overhead of related studies is different due to their various redundancy schemes. Hence, to have a fair comparison, we have presented their own reported results.

As this table shows, the correction capability of our proposed FT-EALU outperforms the related approaches by about 9.4% on average. Since our research aims at

| Fault-Tolerance Approach                           | Average Correction Coverage | Performance overhead over baseline |
|----------------------------------------------------|-----------------------------|-----------------------------------|
| Information redundancy approach [15]              | 46.2%                       | 16.3%                             |
| Hardware reuse approach [4]                        | 38.4%                       | 9.38%                             |
| TMR-based time redundancy approach [20]           | 48.7%                       | 35.99%                            |
| FT-EALU                                            | 52.5%                       | 37.74%                            |
critical embedded systems, their correction capability and robustness in the presence of permanent faults are the main objectives. In addition, the system constraints such as application deadline, cost and power consumption should also be considered. As explained in Sect. 3, to meet these constraints, our proposed weighted voting unit is implemented in software and the weights are determined statically through a learning-based scheme. Further, the performance overhead of our proposed approach consists of triplicating each operation and performing a bit-wise voting among the results considering the determined weights of the learning approach. Based on our implementation, the performance overhead of FT-EALU is about 37.74% in cost of 84.93% correction capability for all ALU operations which is reasonable in critical embedded systems that require high reliability. In other cases, based on their employed redundancy, performance overhead is better but in cost of lower correction capability and other costs.

To perform a thorough study on related research and compare FT-EALU to them in terms of the important features of embedded systems, Table 5 is presented. Here, various related researches with different redundancy schemes are considered and their capability in terms of hardware, time, power consumption costs and correction ability is compared. In this table, the considered methods of the past experiment along with the methods of [18, 19, 27] are evaluated.

As this table shows, different researches with various redundancy approaches are compared in important features. To correct the faults, hardware redundancy-based methods employ extra hardware modules that lead to high cost and power consumption. But their parallel execution feature decreases the execution time overhead and provides appropriate correction capability. Moreover, information redundancy-based schemes add extra codes to the system and processing these codes need some added hardware units. Thus, these schemes force extra hardware units to the system that use additional power consumption, but their time overhead is low and provides good correction capability to the system. It should be mentioned that the correction capability of these methods is tightly dependent to their employed protection code. Last, the time redundancy-based schemes employ serial execution on the

| Fault tolerance schemes | Hardware cost | Time cost | Power consumption | Correction capability |
|-------------------------|--------------|-----------|-------------------|-----------------------|
| Information redundancy-based method ([15]) | ✓ | ✓ | ✓ | ✓ |
| Hardware redundancy-based method ([4]) | ✓ ✓ | – | ✓ ✓ | ✓ |
| Time redundancy-based method ([20]) | – | ✓ ✓ | – | ✓ |
| Information-physical redundancy-based method ([19]) | ✓ ✓ | ✓ | ✓ ✓ | ✓ |
| Hardware redundancy-based method ([27]) | ✓ ✓ ✓ | – | ✓ ✓ ✓ | ✓ |
| Information redundancy-based method ([18]) | ✓ | ✓ | – | ✓ |
| Our proposed FT-EALU | – | ✓ | – | ✓ ✓ ✓ |
existing hardware and force no extra modules and power consumption to the system due to tolerating the faults. In embedded systems that applying redundant hardware is not applicable this feature is required. However, their time overhead because of serial execution of various versions is high, but their correction capability is good due to the considered diversions of input data. In FT-EALU, correction capability is enhanced due to the proposed learning-based weighted voting which adjusts the effect of faulty and correct versions in the voting and final output.

The mentioned characteristics of FT-EALU make it appropriate for embedded and real-time systems. Applying redundant hardware and power consumption is not applicable in embedded systems because of increasing their space and vulnerability of errors. Further, the worst-case time estimation in design of embedded systems is employed to cover the time overhead of FT-EALU. It should be noted that the considered weight learning approach of FT-EALU is performed statically and utilized in runtime, so it does not extend the system execution time.

Finally, the correction capability of our proposed approach could be improved by upgrading it to 5MR-based architecture. If the requirements of the target system in terms of timing constraints allow, it is possible to improve the correction capability of FT-EALU. In this context and based on our experiments, the correction capability of FT-EALU could be improved to 87.3% at cost of 68.59% performance overhead. Thus, the existing trade-off between correction capability and performance overhead of FT-EALU could be adjusted based on the system requirements and the target application.

5 Conclusion remarks

In this paper, a new fault-tolerance approach for ALU of embedded and real-time systems is presented. Due to the expanded applications of embedded systems, their precise and autonomous operation is critical. The arithmetic and logic unit (ALU) as the main part of the processor leads the basic operations of the system and affects its final output directly. Thus, our proposed fault tolerance approach (FT-EALU) aims at making the ALU reliable through a time redundancy-based scheme. FT-EALU mitigates transient faults by replicating the execution versions in time. Moreover, it covers permanent faults by applying variations in input data of replicated computations in time. The mentioned variations are applied on input data through simple modifications such as shift and swap that improve the correction capability along with forcing low-performance overhead to the system. To mitigate the faults and determine the final result, FT-EALU performs weighted voting among the serial replicated executions. In this way, the effect of the occurred faults will be handled during the operation and autonomously as is expected in embedded applications.

The efficiency of the considered weighted voting approach is tightly related to determining the weights. In FT-EALU, this weight is specified based on the fault mitigation capability of various versions of execution and through an adaptive reward/punishment approach. The weights are determined statically and during system design based on several fault injection scenarios. The extracted weights are extracted for each bit of data to improve the precision of our proposed
approach. Then, these weights are applied to the results of the execution versions separately, and the final output is generated by their integration.

Several experiments are performed to demonstrate the effectiveness of our proposed fault-tolerance approach considering the main ALU operations and many fault injection cases. Experimental results show that our proposed approach is capable of correcting 84.93% of the injected single bit stuck-at faults on arithmetic and logical operations on average. Moreover, in the case of occurring double faults, 69.71% of them are mitigated by FT-EALU. The time overhead of FT-EALU is mainly related to the serial executions and could be reduced by reusing the redundant hardware of the system. This overhead is estimated about 37.74% for add operation in cost of correcting 52.5% of injected faults. As a future trend, applying more low-cost variations on operations and decreasing the time overhead of the proposed method by effective approaches such as approximate computing are proposed. Moreover, considering a dynamic threshold to determine the final result of the proposed weighted voter by evolutionary schemes could be interesting.

Data Availability  Please contact the corresponding author for data requests.

Declarations

Conflict of interest  The authors declare that they have no competing interests.

References

1. Wang J (2017) Real-time embedded systems. John Wiley & Sons, Hoboken
2. Li Q, Yao C (2003) Real-time concepts for embedded systems. CRC Press, Florida
3. Veeravalli VS (2009) Fault tolerance for arithmetic and logic unit. In: IEEE Southeastcon 2009, IEEE, pp. 329–334
4. Fazeli M, Namazi A, Miremadi S-G, Haghdoost A (2011) Operand width aware hardware reuse: a low cost fault-tolerant approach to alu design in embedded processors. Microelectron Reliab 51(12):2374–2387
5. Hong S, Kim S (2015) A low-cost mechanism exploiting narrow-width values for tolerating hard faults in alu. IEEE Trans Comput 64(9):2433–2446. https://doi.org/10.1109/TC.2014.2366743
6. Xia Y, Guo S, Hao J, Liu D, Xu J (2021) Error detection of arithmetic expressions. J Supercomput 77(6):5492–5509
7. Majumdar A, Nayyar S, Sengar JS (2012) Fault tolerant alu system. In: 2012 International Conference on Computing Sciences, IEEE, pp 255–260
8. Gracia-Morín J, Saiz Adalid LJ, Baraza Calvo JC, Gil Tomás D, Gil Vicente PJ (2021) Design, implementation and evaluation of a low redundant error correction code. IEEE Latin America Trans 19(11):1903–1911
9. Tay TF, Chang C-H (2017) Fault-tolerant computing in redundant residue number system. Embedded systems design with special arithmetic and number systems. Springer International Publishing, Heidelberg, pp 65–88
10. Nelson VP (1990) Fault-tolerant computing: fundamental concepts. Computer 23(7):19–25
11. Abdi A, Souzani A, Amirfakhri M, Moghadam AB (2012) Using security metrics in software quality assurance process. In: 6th International Symposium on Telecommunications (IST), IEEE, pp 1099–1102
12. Abdi A, Zarandi HR (2018) Hystery: a hybrid scheduling and mapping approach to optimize temperature, energy consumption and lifetime reliability of heterogeneous multiprocessor systems. J Supercomput 74(5):2213–2238
13. Álvarez I, Proenza J, Barranco M (2018) Mixing time and spatial redundancy over time sensitive networking. In: DSN Workshops, pp 63–64
14. Valinataj M, Mohammadnezhad A, Nurni J (2018) A low-cost high-speed self-checking carry select adder with multiple-fault detection. Microelectron J 81:16–27
15. Towhidy Gol A, Omidi R, Mohammadi K (2020) Fault tolerant alu designing based on new implementation of berger code. TABRIZ J ELECTR ENG 50(2):633–644
16. Acharya GP, Rani MA (2018) Berger code based concurrent online self-testing of embedded processors. J Semiconductors 39(11):115001
17. Barredo Ferreira A, Cebrián González JM, Valero Cortés M, Casas Guix M, Moreto Planas M (2020) Efficiency analysis of modern vector architectures: vector alu sizes, core counts and clock frequencies. J supercomput 76:1960–1979
18. Pahuja S, Kaur G (2021) Design of parity preserving arithmetic and logic unit using reversible logic gates. In: 2021 International Conference on Intelligent Technologies (CONIT), IEEE, pp 1–9
19. Santos DA, Luza LM, Dilillo L, Zeferino CA, Melo DR (2021) Reliability analysis of a fault-tolerant risc-v system-on-chip. Microelectron Reliability 125:114346
20. Gade MSL, Rooban S (2020) Run time fault tolerant mechanism for transient and hardware faults in alu for highly reliable embedded processor. In: 2020 International Conference on Smart Technologies in Computing, Electrical and Electronics (ICSTCEE), IEEE, pp 44–49
21. Nikolaidis M (2003) Carry checking/parity prediction adders and alus, IEEE Trans Very Large Scale Integr (VLSI) Syst 11(1):121–128. https://doi.org/10.1109/TVLSI.2002.800526
22. Towhidy A, Omidi R, Mohammadi K (2019) An efficient current mode mvl residue code checker for fault-tolerant arithmetic. J Circuits Syst Comput 28(14):1950244
23. Thakral S, Bansal D (2020) Novel high functionality fault tolerant alu. Telkomnika 18(1):234–239
24. Patel Fung (1982) Concurrent error detection in alu’s by recomputing with shifted operands. IEEE Trans Comput C 31(7):589–595. https://doi.org/10.1109/TC.1982.1676055
25. Hana HH, Johnson BW (1986) Concurrent error detection in vlsi circuits using time redundancy. In: Proc. IEEE Southeastcon, Vol. 86, pp 23–25
26. Johnson B, Aylor J, Hana H (1988) Efficient use of time and hardware redundancy for concurrent error detection in a 32-bit vlsi adder. IEEE J Solid-State Circuits 23(1):208–215. https://doi.org/10.1109/4.281
27. Shukla S, Ray KC (2019) Design andasic implementation of a reconfigurable fault-tolerant-alu for space applications. In: IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS), IEEE 2019:156–159
28. Ahmad U, Ali S, Ahmed R, Qadri MY, Saif H (2021) Fault-tolerant reconfigurable 32-bit alu for space applications. In: 2021 1st International Conference on Microwave, Antennas & Circuits (ICMAC), IEEE, 2021, pp. 1–4
29. Dubrova E (2013) Fault-tolerant design. Springer, Heidelberg
30. Shahoveisi S, Abdi A (2021) E-reso: An enhanced time redundancy-based error detection approach for arithmetic operations. In: Proc. IEEE Iranian Conference on Electrical Engineering(ICEE)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Athena Abdi1 · Sina Shahoveisi1

Athena Abdi
a_abdi@kntu.ac.ir

Faculty of Computer Engineering, K. N. Toosi University of Technology, Tehran, Iran