Chapter 12
Design, Verification, Test and In-Field Implications of Approximate Digital Integrated Circuits

Alberto Bosio, Stefano Di Carlo, Patrick Girard, Annachiara Ruospo, Ernesto Sanchez, Alessandro Savino, Lukas Sekanina, Marcello Traiola, Zdenek Vasicek, Arnaud Virazel

Abstract Today, the concept of approximation in computing is becoming more and more a “hot topic" to investigate how computing systems can be more energy-efficient, faster, and less complex. Intuitively, instead of performing exact computations and, consequently, requiring a high amount of resources, Approximate Computing aims at selectively relaxing the specifications, trading accuracy off for efficiency. While Approximate Computing allows many improvements when looking at systems' performance, energy efficiency and complexity, it poses significant challenges regarding the design, the verification, the test and the in-field reliability of Approximate Digital Integrated Circuits. This chapter covers these aspects, leveraging the authors' experience in the field to present state-of-the-art solutions to apply during the different development phases of an Approximate Computing system.

12.1 Introduction

Despite significant energy efficiency improvements in the semiconductor industry, computer systems keep consuming more and more energy [53]. Interestingly, many widely used applications – such as Recognition, Mining, and Synthesis (RMS)

Alberto Bosio, Marcello Traiola
Univ Lyon, ECL, INSA Lyon, CNRS, UCBL, CPE Lyon, INL, UMR5270, 69130 Ecully, FR , e-mail: [firstname.lastname]@ec-lyon.fr

Patrick Girard, Arnaud Virazel
LIRMM, Université de Montpellier, CNRS, FR e-mail: [firstname.lastname]@lirmm.fr

Stefano Di Carlo, Annachiara Ruospo, Ernesto Sanchez, Alessandro Savino
Control and Computer Eng. Dep., Politecnico di Torino, Torino, IT e-mail: [firstname.lastname]@polito.it

Lukas Sekanina, Zdenek Vasicek
Faculty of Information Technology, Brno University of Technology, Brno, CZ e-mail: [sekanina, vasicek]@fit.vutbr.cz

applications – now target a deployment toward mobile devices and on Internet of Things (IoT) structures. Therefore, it is necessary to improve the next-generation silicon devices and architectures on which these applications will run. The *inherent resiliency property* of RMS applications has been thoroughly investigated over the last few years [53, 22, 12, 9]. This interesting property leads applications to be already (partially) tolerant to errors – as long as their results remain close enough to the expected ones. As reported in [9], the main sources of error tolerance for these applications are:

- noisy real-world inputs,
- redundant data,
- perceptual limitations of individuals who will use the computation output,
- non-deterministic algorithms which lead to non-unique outcomes, and
- self-healing capable systems.

As already pointed out in previous chapters, *Approximate Computing* (AxC) [53, 22] is an emerging computing paradigm that takes advantage of the inherent resiliency property. AxC has garnered increasing interest in the scientific community in the last years. It leverages the intuitive observation that selectively relaxing non-critical specifications may lead to improvements in power consumption, execution time, and/or chip area. AxC has been applied to the whole digital system stack, from hardware to applications.

This chapter focuses on *Approximate digital Integrated Circuits (AxICs)* design and manufacturing flow. Figure 12.1 depicts the main phases of the design flow. The starting points are the requirements, i.e., which functionalities have to be designed coupled with the energy, performances and area requirements, and the AxC metrics, i.e., how to estimate the quality of the outcomes due to approximation. **AxC design** stems from the application of AxC at hardware level. A widely used method to design those circuits is *functional approximation* of conventional integrated circuits (ICs) as described in Chapters 3 and 4.

AxC verification aims at verifying that the approximate design satisfy both the requirements and AxC metrics. If so, the AxC design goes through the manufacturing flow, and the fabricated AxIC will be eventually tested. While **AxC Testing** aims at screening defective circuits, it is interesting to note that AxC metrics have to be considered during the testing phase. Indeed, manufacturing defects may not significantly impact the functionality of the AxIC. AxC testing thus will focus on detecting only defects causing unacceptable degradation of the circuit, usually referred to as **critical-defects**. All critical-defect-free circuits will be ready to be employed in the **in-field** application.

This chapter overviews the AxIC design and manufacturing phases by presenting the main challenges and state-of-art solutions. The chapter is structured as follows: Section 12.3 considers the design phase, Section 12.4 the verification phase, Section 12.5 the testing phase, and Section 12.6 the implication of Approximate Computing on in-field operation. Finally, Section 12.7 summarizes the main contributions of the chapter.

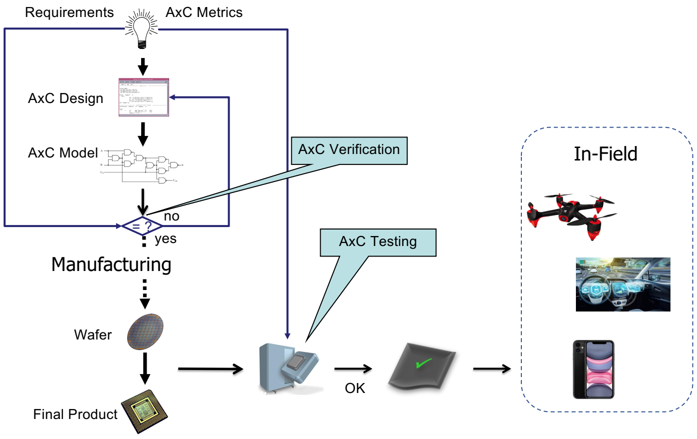


Fig. 12.1: AxIC design and manufacturing flow.

12.2 Background

This section introduces the basic concepts about testing, fault modeling, test generation, and fault simulation. These concepts will serve as background in the rest of the chapter.

12.2.1 Conventional IC testing

To understand the impact of approximate computing on the design and manufacturing, it is necessary to recall some basic principles of conventional IC testing. The reported concepts are not intended to be exhaustive; for an extensive introduction to them, readers may refer to [15]. As sketched in Figure 12.2, in digital testing, the test is carried out in the form of binary patterns (or *test patterns*) applied to circuit's inputs. The circuit's outputs are compared with the expected ones (*golden* responses): if they do not match, the circuit is marked as faulty. The idea came in 1959, when Eldred proposed tests capable of observing the internal state of signals in large digital system, by propagating their effect to primary outputs [10].

Very large-scale integration (VLSI) testing can be classified depending on the goal it is intended to serve:

- **Production testing:** After chip manufacturing, the production testing determines whether the actual manufacturing process produced correct devices or not. This process is performed by the device manufacturer that owns full details about the internal structure of the manufactured system and usually exploits Automated Test Equipments (ATEs) for performing the tests. Different test types are usually performed at the end of manufacturing and their implementation or not may

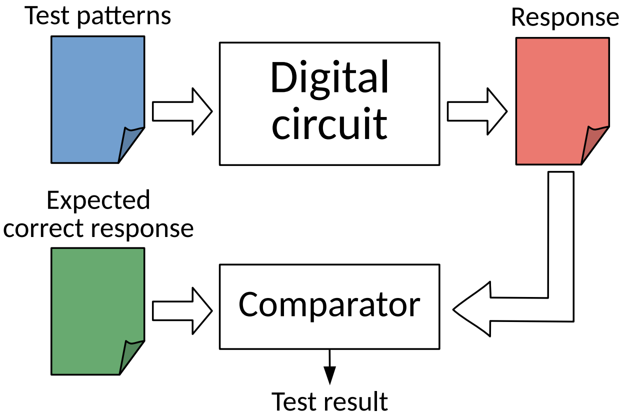


Fig. 12.2: Digital testing [14]

depend on the targeted market of the device. Some of the most common testing processes are: Wafer testing before the wafer is sliced and any device is packaged as a stand alone device. Manufacturing testing that checks for the key parameters of the device and its main functionalities. Burn in testing, where the produced chips are stressed by placing them in high temperature environments, while applying functional and post-production tests; by doing this over a certain period of time, it is possible to guarantee the reliability of the tested devices since the weak devices are eliminated during the burn-in process. During these testing procedures, the devices are stressed by using structural and functional approaches targeting different fault models and the test procedures are performed either at nominal speed or at the speed required by the testing process.

- **In-field testing:** On the other side, when the device is already integrated in the final application and under certain conditions, it is necessary to test the device during its normal operational life, it is required to implement a periodic testing strategy named In-field testing. In this case, the test cannot be performed supported by an ATE, but it should be done through the available mechanisms included in the device itself. In-field testing may require to test the device at the turn-on and turn-off, or periodically while running concurrently with the actual application. Today, some industrial standards such as ISO26262 for automotive, and DO254 for avionics provide the guidelines for implementing these kind of test strategies for different safety-critical applications.

Usually, two are the types of tests performed on VLSI chips:

- **Functional test:** it is possible to define functional test considering the way in which the test procedure is applied and the information used to develop the whole testing procedure. In the first case, the test procedure is performed acting on the functional inputs of the device under test and observing the functional outputs, only, without resorting to any kind of special mechanisms as the ones called

Design for Testability (DfT). On the other side, the test procedure is developed on the basis of the functional information regarding the module under test, only: therefore, it aims at testing the functions rather than the faults, this kind of test can be considered as a kind of black box testing strategy.

- **Structural test:** a structural test exploits the structural information of the device under test to generate the resulting stimuli set. In most of the cases, the strategies based on structural test use the circuit netlist, that represents the topological distribution of all the logic gates composing the circuit, and the circuit fault list to create specific test patterns able to cover the complete list of faults. Very sophisticated algorithms and strategies have been proposer to efficiently exploit the structural information of the circuit to generate the stimuli set, this is the case for example of modern ATPG tools.

Structural testing is usually considered opposed to functional test since in general these strategies do not use the functional inputs of the device to apply the test patterns, but exploit for example the circuit scan chains and more sophisticated strategies such as logic Built In Self Test (BIST), being in this way contrary to the first definition of functional test. On the other side, since the stimuli set is generated resorting to the structural circuit information and not use the circuit functionalities, this makes this test strategy contrary to the second definition given before.

12.2.2 Defect modeling

To correctly describe a *faulty* electronic circuit, different terms have to be defined. Below, we report the common definitions of *Defect*, *Error* and *Fault*.

- **Defect**: Unintended difference between the implemented hardware and its design. Defects can occur during manufacture, as well as during the device lifetime.
- **Error**: A wrong output signal produced by a defective system. An error is caused by some defect in the hardware.
- **Fault**: An abstraction model of a defect.

It is important to highlight that even if a defect is present within an IC, its effects might not affect the IC behavior. In general, given the list of all possible defects (modeled as faults) that can occur within an IC, a fault is defined as *detectable* if it exists an input pattern sensitizing and propagating the fault effect to outputs. All faults being detectable are referred to as *detectable faults*. From now on in the text, we will refer to defect and to its model – the fault – interchangeably.

Fault modeling can be described at different levels of abstraction:

- **Behavioral level**: Sometimes referred to as *high level faults*, behavioral level fault models may not have correspondence in manufacturing defects because the model the general behavior. Mostly, they are used in design verification rather than testing.

- **Logic level or Register-transfer level (RTL)**: At this level, we find fault models usually built by considering the *netlist*, i.e., the circuit component list and their interconnections. **Stuck-at** fault model is the most popular and used one in digital testing. Among others, we find *delay* fault model and *bridging* fault model.
- **Component level**: this is the lower abstraction level, such as the transistor level. Stuck-open fault model, which is a technology-dependent model, is mainly used at this level. Mostly, analog circuit testing resorts to component level fault models.

In this chapter, we focus on logic level fault models, since we address digital integrated circuit testing. In the following, we report some definition's concerning faults, in order to provide readers with some useful terms for the rest of the chapter.

- **Stuck-at fault model (SaF)** – In this abstraction, a circuit net is considered to be permanently set at a constant value. By assigning a fixed (0 or 1) value to an input or an output of a logic gate or to a flip-flop in the circuit, the SaF model represents this condition. The SaF model is the most popular fault model used in practice for digital IC testing. The most popular forms are the *single stuck-at faults*. In this abstraction, a single faulty line is assumed to be present in the IC, either stuck-at-1 (Sa1) or stuck-at-0 (Sa0).
- **Delay fault model** – Defects modeled by delay fault model prevent the correct data from reaching outputs at the right time. Among different types of delay faults models we find transition faults, gate-delay faults, path-delay faults.
- **Redundant fault** – In a combinational circuit, a redundant fault does not modify the circuit's output for any input combination. Thus, a test detecting a redundant fault cannot exist. Redundant faults are a subset of the more general *untestable faults*. In sequential circuits, faults for which no test pattern can be found fall into the untestable fault category.
- **Multiple fault** – The condition that simultaneous single faults affect the same circuit is referred to as multiple fault. Multiple Stuck-at faults model is usually not considered, due to the tremendous complexity. Moreover, a very high percentage of these faults are covered by single stuck-at faults tests.
- **Equivalent faults** – If two faults f_1 and f_2 lead a circuit to exhibit the exact same behavior, they are defined as *equivalent*. A test detecting f_1 detects also f_2 and vice-versa. This leads to *fault collapsing*: partitioning all the faults of a circuit into disjoint equivalence sets and selecting one fault from each equivalence set to test. For a circuit having n lines (thus $2n$ single stuck-at faults) the equivalence between $2(n^2 - n)$ pairs of faults should be determined, which is complex. Therefore, for stuck-at fault model, the fault equivalence is usually determined between faults affecting each Boolean gate.

12.2.3 Fault simulation

In the design of VLSI circuits, the concept of *simulation* is of great importance. Firstly, it serves the purpose of verifying the circuit correctness. Secondly, it verifies whether and how efficiently a test set fulfills its purpose.

The circuit correctness verification is a fundamental step of the design activity. After the synthesis process, the produced netlist is verified by a *true-value simulator*, i.e., it produces the responses of the defect-free circuit. Since the goal is to verify the circuit functionality according to the specification, the input stimuli (or vectors) applied by the simulator to the circuit are based on the specification. The main assumption is that circuit errors lead to change the design to make responses to all stimuli different than the ones expected by the specification.

Simulation is also used for the development of manufacturing tests. A *fault simulator* acts like a true-value simulator with the capability to simulate a faulty-circuit. Once the verified circuit netlist is available, the fault simulator can measure the percentage of faults that are detected when a given set of input stimuli (usually, the verification ones) is applied to the circuit. Faults are organized in *fault lists* and input stimuli in *test sets*. Faults covered by the given test set are marked as *detected* and the *Fault Coverage* is measured.

Definition 12.1 Fault Coverage (FC): the ratio of the number of faults detected by a set of test patterns to the total number of faults in the fault list.

An adequate FC (98% - 100%) is usually required in order to ship high quality devices to the customers. A good-quality test is a test that can minimize the number of faulty circuits sold, while keeping the test cost acceptable.

Definition 12.2 Test quality: the fraction of chips that, despite having passed the test, are actually faulty. It is usually referred to as *defect level* (or *field reject rate*). Defect level is expressed as *parts per million (ppm)*. High quality tests are considered as providing chips with a defect level of 100 ppm or lower.

Process variations, such as impurities in materials, dust particles, etc., can produce defects during the manufacture. In turn, defects can cause circuits to fail. Process variation effects reflect on the *process yield* defined as follows:

Definition 12.3 Process yield: the fraction (or percentage) of acceptable parts (thus, sold) among all fabricated parts. It is also commonly referred to simply as *yield*.

In a typical case, a newly designed chip has a low yield at its early manufacturing period. Thanks to process diagnosis and correction, a higher process maturity is achieved and, thus, a significantly higher yield.

12.2.4 Test generation

In late-fifties, Eldred highlighted the necessity for the structural testing of logic circuits to prevail over the classic functional test [10]. He argued that formulating test conditions at the level of the components is *“the only way in which all conditions of operation of each logical function can be uniquely [...] defined and all logical components within each logical function can be made to perform the task to which they are assigned [...] thereby producing a minimum program which tests and detects failure”*. The goal of structural test is to verify the presence of the minimal set of faults in the circuit. Therefore, the application of fault equivalence is important to reduce the final set of faults to test. The *Automatic Test Pattern Generation (ATPG)* serves the purpose of producing patterns to test the internal structure of a digital circuit, starting from its netlist description. The commonly used method in ATPG, namely *path sensitization*, is based on three steps:

1. **fault *injection*** in the circuit netlist;
2. **fault *activation***;
3. **fault effect *propagation*** toward circuit outputs.

To briefly describe path sensitization, let us resort to the stuck-at fault model (see Subsection 12.2.2). Let us assume that we want to test if a line *l* is stuck to a constant value (say 1). The test vector *v* detecting that fault is composed of input values such that:

- the line *l* is set to the opposite value of the fault (say 0). This is commonly referred to as fault *sensitization* or *activation* or *excitation*;
- the effect of the previous action is propagated to circuit outputs. This is commonly referred to as fault *propagation* or *path sensitization*.

By simulating the pattern with the fault-free circuit, we obtain the fault-free output value (expected output). Now, let us assume that an actual stuck-at fault (say Sa1) occurs at line *l*. In presence of the fault, the circuit outputs will be different from expected. Therefore, by applying the test vector *v* to the circuit and knowing the expected output, we are able to detect the fault by observing a difference between actual and expected outputs.

In the context of conventional IC test, even a little difference between the nominal behavior and the manufactured IC behavior leads to reject the circuit. Later in this chapter, we will discuss this aspect when approximate computing is considered. In this particular context, the magnitude of the difference between the nominal behavior and the manufactured IC behavior is important. In fact – under specific conditions – the manufactured circuit may be still accepted even if some defects occur.

The above described ATPG method works correctly only for combinational circuits, i.e. without cycles. In fact, any circuit with cycles will lead the aforementioned method to fall into an infinite loop. ATPG methods for sequential circuits exist but are usually very resource-consuming and sometimes inefficient. The main difficulty for *sequential ATPG* is to control and to observe the internal state of the circuit.

Therefore, *design-for-testability (DfT)* comes into play. As stated by Agrawal and Seth [1], "*testability is the property of a circuit that makes it easy (and sometimes possible!) to test"*. DfT refers to the set of design techniques for ICs aiming at improving the testability of the target design. The most popular DfT technique is the scan design. However, DfT techniques are out the scope of this chapter. More details can be found in [1].

12.3 Approximate Integrated Circuits Design phase

The AxC paradigm has been successfully applied to digital ICs. The first technique was referred to as *over-scaling based approximation*. Basically, the IC is forced to work outside its specified operating conditions [23]. The classical example is the reduction of the supply voltage under the minimum value. This will result in energy saving but it will introduce timing errors. The second technique is the *functional approximation* [23]. Functional approximation aims at modifying the circuit so that its original behavior F is replaced by a similar one F', whose implementation leads to area/energy reduction at the cost of a reduced accuracy. In other words, being some responses of F' different compared to F, the circuit output is sometimes erroneous but it is computed more efficiently. The accuracy loss is always measured by means of quality metric(s). The most adopted ones are the Error Rate (i.e., how many times an error is observed at circuit outputs) and the Error Magnitude (i.e., the difference between the golden and erroneous outputs), formally defined in [23].

In the literature, several approaches for functional approximation have been proposed so far, and they can be classified as manual or automated [23]. Manual techniques target specific (small) circuit designs like adders and multipliers [24], [13]. On the other hand, to manage more complex circuit designs, automated approaches are mandatory. State-of-the-art techniques for Approximated Logic Synthesis (ALS) can be summarized as follows.

The approaches in [32] and [20] target two-level circuits and considers both the error rate and error magnitude as quality metrics. Concerning multi-level circuits, SALSA [47] encodes the quality metric as a function and it further simplifies the circuit exploiting the resulting *don't cares*. This approach can only be applied by taking into account error magnitude metrics. In [21], the authors propose to consider both error rate and error magnitude during the ALS. SASIMI [46] aims at identifying internal circuit net pairs with high probability to have the same logical value. Then, it replaces one net with the other in order to simplify the circuit structure. It considers Error Rate as the quality metric. In [52], the authors propose an approach based on local node simplification in the circuit structure.

Shin et al. introduce in [33] a different approach based on the idea to inject stuck-at-faults into the circuit to simplify it under a composite constraint on error rate and error magnitude. All the above approaches target combinational circuits only. To the best of our knowledge, ASLAN [28] is the only one targeting ASL for sequential circuits. The basic idea is to "unroll" the sequential circuit in time frames also called an iterative logic array of the circuit. For each time frame, the flip-flop inputs from the previous time frame are often referred to as pseudo primary inputs with respect to that time frame, and the output signals to feed the flip-flops to the next time frame are referred to as pseudo primary outputs [51]. In this way, the sequential circuit can be represented as a combinational one on which it is possible to apply the SASIMI approach (on each time frame). Clearly, the complexity of ASLAN is very high and generally depends on the sequential depth (i.e., how many time frames have to be considered). Therefore, a faster and simpler ASL approach needs to be introduced.

A different approach for functional approximation of multi-level circuits leverages on the concept of **fault simulation** [33]. To better show the main principle, let us resort to the example depicted in Figure 12.3. The circuit example is a two-bit multiplier described in [13]. On this simple circuit, we first define as the quality metric the error magnitude and, more in detail, we set to 4 the maximum acceptable error magnitude. In order to approximate the two-bit multiplier structure using the approach of [33], we consider the output *out3* as affected by a stuck-at-0 fault. In other words, we force this output to be always at logic '0'. The example will clarify why considering the stuck-at-0 fault affecting *out3* ensures that the error constraint will be satisfied. Now, the approach of [33] performs a back-track propagation of the forced fault from the affected output back to a "barrier". Each time that a logic gate is traversed, faulty values are forced to its input to justify the faulty value at its output, and the traversed gate is removed. The barrier is either a primary input or a branch node. In Figure 12.4, we report in bold the back-track propagation. When a branch is found, a forward propagation is performed. The faulty value is now propagated to reach other outputs and each time that a logic gate is traversed, it is "simplified" depending on the traversed gate and the faulty value. The forward propagation is reported in bold red in Figure 12.4. Here, the logic value '0' is set as input of the XOR gate, that can be simply removed since $x \oplus 0 = x$. Finally, Figure 12.5 reports the final result.

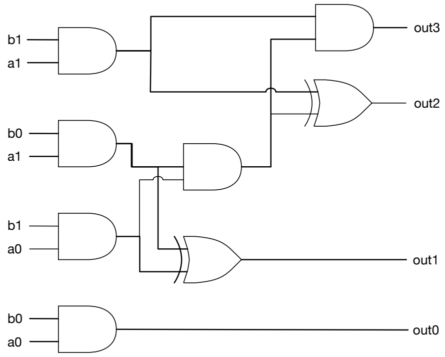


Fig. 12.3: Two-bit multiplier example

Looking at the obtained circuit, the error margin constraint can be satisfied by removing the most significant output bit (*out3* in Figures). Table 12.1 gives all the possible results for all the possible inputs. The only error (highlighted in **red**) appears during the computation of 3 x 3. The result should be 9, but rather we get 5 due to

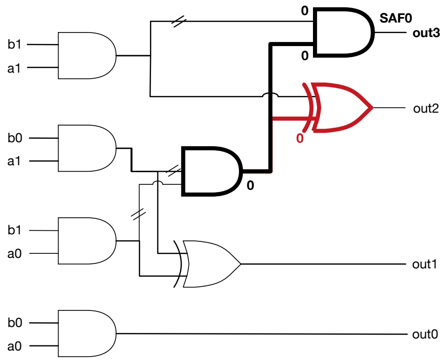


Fig. 12.4: Fault Injection based functional approximation

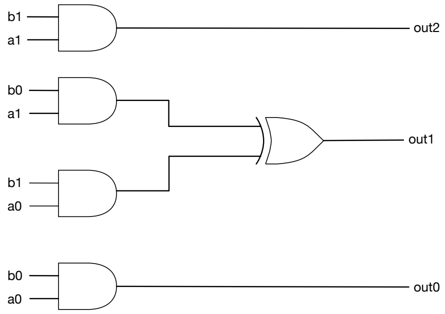


Fig. 12.5: Approximate two-bit multiplier

the approximation. However, the erroneous result is still acceptable since the error magnitude is 4 (9 - 5).

Table 12.1: Error Magnitude Example

A x B	**0**	**1**	**2**	**3**
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	**5**

Please note that in this simple example only one fault is considered. However, if more than one output has to be considered, multiple faults have to be used (one fault per output), thus leading to increase the complexity of the fault analysis (backtrack as well as forward propagation). Moreover, the fault affecting output *out3* has been selected because it allows to achieve the best approximate solution. Actually, we performed the same analysis for the other faults and we found out that the area reduction was lower compared to the one shown in the reported example. In general, there are no guidelines for the selection of the fault. In [35], an extension has been published including the strategy of considering a single fault instead of multiple ones. Therefore, both processes of fault selection and fault analysis are significant simplified compared to [33]. Additionally, the whole methodological flow does not involve any iteration, but rather it requires to run a fault simulation once. Moreover, it can be taken into consideration for approximating both sequential and combinatorial circuits and it can be used with an arbitrary quality metrics, including the Error Rate and Error Magnitude.

12.4 Approximate Integrated Circuits Verification phase

The verification phase of a digital circuit design typically employs a method capable of determining whether the circuit exhibits the same behavior as the so-called golden model. The verification can be conducted by means of simulation, but reaching all possible states of a complex circuit is usually intractable for any simulation algorithm, i.e., the simulation does not guarantee that the circuit perfectly meets all requirements. Hence, formal equivalence checking methods have been developed that try to formally prove that two representations (the golden one and the proposed one) of a circuit design exhibit exactly the same behavior. In the context of approximate computing, the verification problem is reformulated in such a way that we try to prove that the golden model and an approximate circuit are equal up to some bound with respect to a chosen distance (error) metric [48]. A particular equivalence checking method's success depends on several factors, primarily including the circuit type, the circuit complexity, and the error metric. Current methods are capable of an exact error analysis only for some circuits and error metrics. On the other hand, complex approximate circuits (such as 128-bit adders, 32-bit multipliers, 32-bit Multiply and Accumulate circuits, and 31-bit dividers) have already been reported together with determining their exact worst-case errors [6]. Most research deals with formal error analysis of arithmetic circuits as they frequently appear in the most popular error-resilient applications such as deep learning and video processing. Formal methods can also be applied to analyze the errors of other combinational circuits effectively (e.g., complex median networks [45]) as well as sequential systems [7].

Two main approaches have been developed for equivalence checking techniques based on Reduced Ordered Binary Decision Diagrams (ROBDD) and satisfiability (SAT) solvers [44]. In both cases, an auxiliary circuit, the so-called *miter*, is constructed and then analyzed. The miter connects corresponding outputs of the candidate circuit (to be checked), the golden circuit, and an error-specific circuit to determine the approximation error. As ROBDDs are inefficient in representing classes of circuits for which the number of nodes in the BDD is growing exponentially with the number of input variables (e.g., multipliers and dividers), their use in equivalence checking of approximate circuits is typically possible for adders and other less structurally complex functions [44].

If the error analysis is based on SAT solving, the miter is represented as a logic formula in Conjunctive Normal Form (CNF) for which SAT solver decides whether it is satisfiable or unsatisfiable. The interpretation of this outcome depends on the construction of the miter, see Chapter 4. Common SAT solvers are, in principle, applicable to the worst-case analysis only. However, this approach is more scalable than ROBDDs for the error analysis of multipliers [31]. Specialized SAT solvers (#SAT) are capable of counting the number of satisfiable assignments. Still, their scalability is very limited, and thus they are currently less practical for the exact error analysis [44].

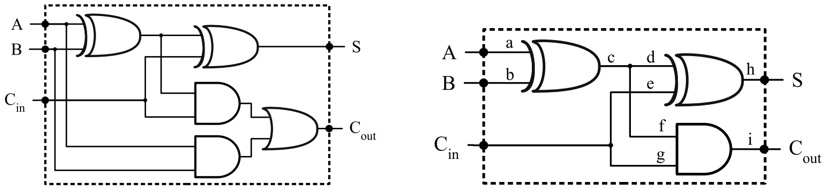
The performance of verification algorithms is critical if the circuit approximation process is based on a fully automated search in the space of approximate implementations and every candidate implementation has to be verified. A detailed overview of formal verification techniques for approximate arithmetic circuits is provided in Chapter 4.

12.5 Approximate Integrated Circuits Testing phase

This section focuses specifically on the testing aspects of functionally approximate circuits. These circuits are referred to as *Approximate Integrated Circuits (AxICs)*. Since approximating circuits alters their functional behavior, techniques to test them must be revisited [49, 50, 39, 38, 37, 2, 36, 40, 41, 42, 4]. As a matter of fact, extending the basic testing concepts to AxICs is not straightforward. In particular, as mentioned in Subsection 12.2.4, during the test of a conventional circuit, any change in its functional output signals with respect to the expected values leads to labeling the circuit as faulty and discarding it. When dealing with AxICs, the presence of a fault may lead the circuit to behave differently than expected, yet still in an acceptable manner. In this case the circuit should not be discarded. Acceptable behaviors are defined according to one or more error metrics and corresponding bounds, fixed in the design phase and usually expressed as thresholds. Mastering these mechanisms may lead to increase the production process yield, i.e., increase the percentage of sold AxICs among all fabricated AxICs.

To illustrate the issue related to the AxIC test, throughout the whole section we refer to a simple arithmetic circuit, shown in Figure 12.6. The figure depicts a 1-bit Full Adder (FA) (12.6a) and an approximate version of it (12.6b), which is more efficient, i.e., with reduced area (3 logic gates instead of 5) and lower delay (2 logic levels instead of 3), but shows some errors at outputs. Figure 12.6c reports the truth tables of both the circuits. We also report the integer representation of both the circuit outputs ("Int" column), calculated as $S * 2^0 + C_{out} * 2^1$. As reported in Figure 12.6d, by considering all the possible circuit inputs, we can calculate the error values according to the following metrics, *Worst Case Error* (WCE), *Mean Absolute Error* (MAE), *Mean Squared Error* (MSE), and *Error Probability* (EP) [19], defined as follows:

$$\text{WCE} = \max_{\forall i \in \mathcal{I}} \left| O_i^{\text{approx}} - O_i^{\text{precise}} \right| \tag{12.1}$$



(a) accurate 1-bit full adder

(b) approximate 1-bit full adder

	Inputs			Accurate			Approximate		
#	C_{in}	X	Y	C_{out}	S	Int	C_{out}	S	Int
0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	1	1	0	1	1
2	0	1	0	0	1	1	0	1	1
3	0	1	1	1	0	2	0	0	**0**
4	1	0	0	0	1	1	0	1	1
5	1	0	1	1	0	2	1	0	2
6	1	1	0	1	0	2	1	0	2
7	1	1	1	1	1	3	0	1	**1**

(c) Truth tables

Error metric	Threshold
WCE	2
MAE	0.5
MSE	1
EP	0.25

(d) Error thresholds

Fig. 12.6: (b): example of an approximate 1-bit full adder, obtained from the accurate 1-bit full adder in (a). The sub-figure (c) shows the truth tables of the two circuits: for each input, the output bit values are reported (S and C_{out}), as well as their unsigned integer representation, calculated as $S * 2^0 + C_{out} * 2^1$. Sub-figure (d) reports the error thresholds for the approximate circuit, for different error metrics.

$$\text{MAE} = \frac{\sum_{\forall i \in \mathcal{I}} \left| O_i^{\text{approx}} - O_i^{\text{precise}} \right|}{2^n} \tag{12.2}$$

$$\text{MSE} = \frac{\sum_{\forall i \in \mathcal{I}} \left| O_i^{\text{approx}} - O_i^{\text{precise}} \right|^2}{2^n} \tag{12.3}$$

$$\text{EP} = \sum_{\forall i \in \mathcal{I}:\ O_i^{\text{approx}} \neq O_i^{\text{precise}}} \frac{1}{2^n}. \tag{12.4}$$

where:

$i \in \mathcal{I}$ is the input value within the set of all possible inputs $\mathcal{I}$,
$O_i^{precise}$ is the precise output integer representation, for input i,
O_i^{approx} is the approximate output integer representation, for input i, and
n is the number of input signals to the circuit.

Values reported in Figure 12.6d are a direct consequence of the approximation. They constitute the error threshold values of the AxIC, fixed by specification and known at design time. Thus, after manufacturing, the produced AxIC must produce outputs respecting the boundaries set by the error thresholds. The issues shown in this section and the approaches to face them are generic and applicable to all kind of combinational circuits, provided that a measure of their approximation error is available and reproducible.

The following section illustrates a test flow – called *Approximation-Aware (AxA) test flow* – to properly deal with the test of AxICs. The flow is composed of three main steps: (i) AxA fault classification, (ii) AxA test pattern generation and (iii) AxA test set application. Briefly, the *fault classification* divides faults producing critical effects on the circuit behavior from those producing acceptable effects. The *test pattern generation* produces test stimuli able to cover all the critical faults and, at the same time, to leave acceptable faults undetected, as much as possible. Finally, the *test set application* labels AxICs under test as critically faulty, acceptably faulty, or fault-free. Only AxICs falling into the first group will be discarded, thus minimizing overtesting (i.e., minimizing AxICs discarded due to acceptable faults). Next subsections describe each AxA test step.

12.5.1 AxA fault classification

The first step of the AxA testing is the *fault classification*. It aims at separating acceptable faults from critical ones. The outputs of this phase are two fault lists (critical and acceptable). The part of detectable faults classified as acceptable constitutes the *expected Yield Increase* (eYI) with respect to the conventional test. The eYI is expressed as follows:

$$\text{eYI} = \frac{\text{acceptable faults}}{\text{total faults}} \tag{12.5}$$

The measure of the eYI is another outcome of the AxA fault classification. The purpose of such a metric is to establish an upper bound to the achievable yield gain. To turn eYI in an actual gain, we have to go through the other AxA testing phases, discussed throughout this chapter.

The key aspect to consider in the fault classification is the AxICs' output deviation measure. As mentioned in the previous subsection, different error metrics exist to measure AxIC output deviations [19]. In Table 12.2, in the left part we report the error threshold value alterations caused by all possible Stuck-at faults in the approximate FA (Figure 12.6b). The fault list was generated with a commercial tool [26] with the fault collapsing option active. We highlight in red solid-bordered boxes the non-acceptable error values, i.e. higher than the respective thresholds, shown in Figure 12.6d. We use the notation *SaX@N* to indicate a "stuck-at-X affecting the net N", where X can be either the value 1 or 0 and N is the label of the net. Please, refer to Figure 12.6b for the net labels. By observing the table, we can firstly remark that not all the metrics are impacted by the same faults. Furthermore, in some particular cases, faults even reduce the observed error (green dash-bordered boxes in Table 12.2).

Table 12.2: Error metric values in presence of different faults affecting the approximate circuit in Figure 12.6b, and fault coverage report for the exhaustive test set.

Fault	Error metrics				Test vectors*							
	WCE	MAE	MSE	EP								
Fault-free	2	0.5	1	0.25	0	1	2	3	4	5	6	7
Sa1@a	2	$\boxed{1}$	$\boxed{1.5}$	$\boxed{0.75}$	X	X			X	X		
Sa0@a	⬚ 1	0.5	⬚ 0.5	$\boxed{0.5}$			X	X			X	X
Sa1@b	2	$\boxed{1}$	$\boxed{1.5}$	$\boxed{0.75}$	X		X		X		X	
Sa0@b	⬚ 1	0.5	⬚ 0.5	$\boxed{0.5}$		X		X		X		X
Sa1@c	⬚ 1	0.5	⬚ 0.5	$\boxed{0.5}$	X			X	X			X
Sa0@c	2	$\boxed{1}$	$\boxed{1.5}$	$\boxed{0.75}$		X	X			X	X	
Sa1@d	$\boxed{3}$	$\boxed{0.75}$	$\boxed{1.5}$	$\boxed{0.5}$	X			X	X			X
Sa0@d	2	$\boxed{1}$	$\boxed{1.5}$	$\boxed{0.75}$		X	X			X	X	
Sa1@e	2	$\boxed{0.75}$	1	$\boxed{0.625}$	X	X	X	X				
Sa0@e	$\boxed{3}$	$\boxed{1}$	$\boxed{2}$	$\boxed{0.625}$					X	X	X	X
Sa1@h	2	$\boxed{0.75}$	1	$\boxed{0.625}$	X			X		X	X	
Sa0@h	$\boxed{3}$	$\boxed{1}$	$\boxed{2}$	$\boxed{0.625}$		X	X		X			X
Sa1@f	2	0.5	1	0.25					X			X
Sa1@g	2	$\boxed{1}$	$\boxed{2}$	$\boxed{0.5}$		X	X					
Sa1@i	2	$\boxed{1}$	$\boxed{2}$	$\boxed{0.5}$	X	X	X	X	X			X
Sa0@i	2	$\boxed{1}$	$\boxed{2}$	$\boxed{0.5}$						X	X	
eYI(%)	81.25%	25%	37.5%	6.25%								

*0="000", 1="001",..., 7="111"

□ = critical effect, ⬚ = beneficial effect

The complexity of the classification task depends on the complexity of the metric computation. For instance, in the example in Figure 12.6, let us suppose that the approximate circuit had a defect whose effect set the net h to 0 (Sa0@h). To measure the impact on the different metrics described by Equations 12.1, 12.2, 12.3, and 12.4 different procedures are required. To find out that the WCE threshold is not respected, it is only necessary to find a single input stimulus generating an erroneous output having $WCE > 2$, e.g., input 7 (111) that should give 3 rather than 0 ($3 - 0 = 3 > 2$). Conversely, to find out that the thresholds for MAE MSE and EP are not respected either, we should test all the possible input stimuli to observe the outcome and then calculate a mean to obtain the final results. More in general, for the WCE we have to demonstrate the existence or nonexistence of a single input vector whose application leads the circuit to produce an out-of-bounds error. This task is well achieved by Satisfiability solvers (SAT) or by using ATPG for integrated circuits. By using the previously mentioned auxiliary *miter* module, SAT solvers or ATPGs can fairly easily manage the problem. Conversely, to measure the impact of a fault on metrics based on average calculation, simulation approaches are preferred. When the complexity of the circuit does not allow using an exhaustive analysis, a random or workload-dependent subset of input is used to estimate the measure. SAT solvers counting the number of satisfiable assignments exist but they suffer of scalability issues.

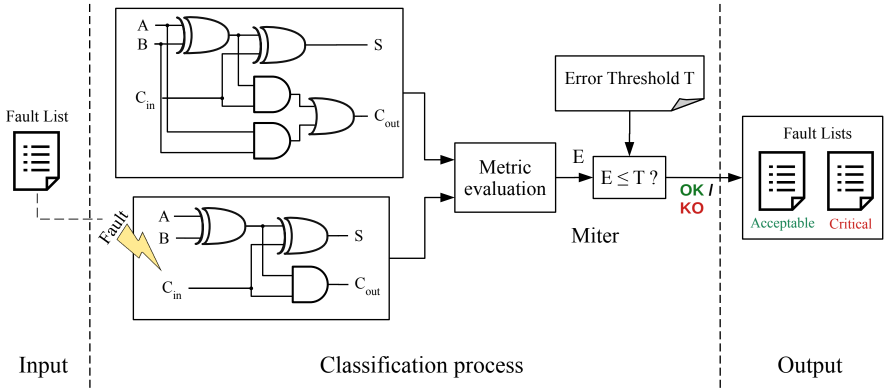


Fig. 12.7: Approximation-Aware (AxA) fault classification concept

Figure 12.7 sketches the necessary modules to obtain a classification similar to the one in Table 12.2: the original (precise) circuit, the AxIC under test, and the *miter* module that performs the evaluation on the circuit outputs with respect to the chosen error metric(s). The final output reports an erroneous condition when the AxIC produce output values outside the error metric bounds. Thus, when the AxIC is fault-free, no erroneous conditions are reported. The underlying idea is masking acceptable fault effects by using a filter (implemented by the miter). In this way, only critical faults generate an error condition at the output of the miter. Hence, the conventional test approaches mentioned at the beginning of the chapter can be used to classify the AxIC possible faults. In particular, for the WCE metric – and for all the metrics for which only a punctual condition must be demonstrated – a conventional ATPG approach (or a SAT-based one) can be used: given a fault, the procedure proves whether an input stimuli generating an error condition at the output of the miter exists or not and classifies the fault accordingly. For MAE, MSE, and EP – and for all the metrics requiring the calculation of a mean – fault injection and input simulation can be used: given a fault, it is injected in the AxIC, a set of input stimuli (exhaustive, random or workload-dependent) is simulated and the metric is calculated in the miter considering all the results. More details on the approaches are reported in [38, 36, 42]. Finally, it is worth mentioning that the above-discussed miter is never manufactured. It is only used in simulation to classify the AxIC faults.

12.5.2 AxA test pattern generation

The second step of the AxA testing is the *test pattern generation*. Historically, a lot of effort has been spent in providing test generation methodologies achieving higher *Fault Coverage (FC)* for conventional integrated circuits. In the context of AxICs, test patterns must cover all critical faults and as few as possible acceptable ones.

Respecting both these conditions is crucial to discard AxICs affected by critical defects and, at the same time, to avoid discarding those affected by acceptable defects. To achieve this goal, it is necessary to find, among the input vectors, the smallest subset covering all the critical faults and minimizing the acceptable fault coverage. Therefore, the concept of *Fault Coverage (FC)*, defined in Subsection 12.2.3 Definition 12.1, needs to be divided into *acceptable FC* and *critical FC*, as defined below:

$$\text{acceptable FC} = \frac{\text{acceptable faults detected}}{\text{total acceptable faults}} \tag{12.6}$$

$$\text{critical FC} = \frac{\text{critical faults detected}}{\text{total critical faults}} \tag{12.7}$$

Naturally, for conventional approaches a good test set aims at detecting as much faults as possible without considering their classification. Conversely, in the AxIC case, an ideal test set should achieve 100% critical FC and 0% acceptable FC. If no effort is spent towards achieving the second condition, a still-good AxIC affected by an acceptable fault will be rejected during the test phase. The phenomenon due to which a good product is considered as faulty by the test process is commonly referred to as *over-testing*.

If not properly managed, the over-testing will eventually cause some yield reduction.

Let us refer again to our example. The right part of Table 12.2 reports the input stimuli detecting (i.e., sensitizing and propagating to outputs) each possible Stuck-at fault for the AxIC in Figure 12.6b. Firstly, let us assume that the fault classification is performed by using the EP metric (threshold = 0.25). Table 12.2 shows that all the faults lead the error to be critical, except for Sa1@f, that leaves it to 0.25. Therefore, vectors 4 and 7 must be avoided, since they are the only ones detecting that fault. An example of test set achieving 100% critical FC and 0% acceptable FC is {2,3,5}. However, it is not always possible to find a suitable test set satisfying these conditions. For example, let us consider that the fault classification is performed by using the WCE metric (threshold = 2). In Table 12.2 we can observe that three faults lead the error to be critical, i.e., Sa1@d, Sa0@e, and Sa0@h increase the WCE to 3. Both vectors 4 and 7 independently detect the three faults. However, both vectors detect also five acceptable faults (38% acceptable FC). Moreover, there is no input vector combinations achieving 100% critical FC and 0% acceptable FC all at once. A further analysis highlights that vector 4 covers four acceptable faults leaving the WCE to 2 and one lowering it to 1, while vector 7 covers three acceptable faults lowering the WCE to 1. Therefore, this consideration would lead to the selection of vector 4 over vector 7 to avoid detecting too much faulty conditions that actually improve (i.e., lower) the WCE. In conclusion, it is not always possible to achieve 0% acceptable FC and 100% critical FC at all at once. As a consequence, an ideal *AxA test pattern generation* approach should produce a test set achieving 100% critical FC and an acceptable FC as close as possible to 0%. Unfortunately, conventional ATPG algorithms are not designed to produce test set with such particular properties.

One way to achieve an improved test set (i.e., with 100% critical FC and low acceptable FC) is to develop an exploration methodology to find, among the input vectors, the smallest subset covering all the critical faults and minimizing the acceptable FC. Such a methodology, sketched in Figure 12.8, measures both critical and acceptable FCs of the AxIC input vector set and formulates and resolves an *Integer Linear Programming (ILP)* optimization problem to find the smallest subset achieving 100% critical FC and minimizing the acceptable FC. The ILP solution is the final *ax-aware test set*. More in details, firstly a (sub)set $\mathcal{S}$ of the AxIC input vector set is generated. Then, fault simulation is used, taking as input $\mathcal{S}$, the AxIC and the two fault lists (critical and acceptable) generated in the AxA fault classification phase. The output of the fault simulation is a *fault coverage (FC) report* which records, for each fault, all the input vectors in $\mathcal{S}$ covering it, as shown in Table 12.2. Finally, an optimization problem is formulated, by using the fault coverage report and the fault lists. This leads to a system of linear inequalities whose solution will be the final *ax-aware test set*, i.e., the smallest subset $\mathcal{V} \subset \mathcal{S}$ which minimizes the acceptable FC and achieves 100% critical FC. If $\mathcal{S}$ corresponds to the exhaustive input set of the AxIC, the output solution will be the *globally optimal* one (i.e., the best possible vector combination). When $\mathcal{S}$ is a sub-set of the exhaustive vector set, the ILP solution will be *locally optimal* (i.e., the best combination, among vectors in $\mathcal{S}$). This approach is independent of the specific fault classification technique and of the error metrics and thresholds. Indeed, as long as a fault classification is correctly produced, the methodology is applicable. Further details on the approach and its mathematical basis are available in [40]. Although the methodology guarantees finding an optimal test set, the ideal outcomes (i.e., 100% covered critical faults and 0% covered acceptable faults) cannot be guaranteed, due to the structure of the

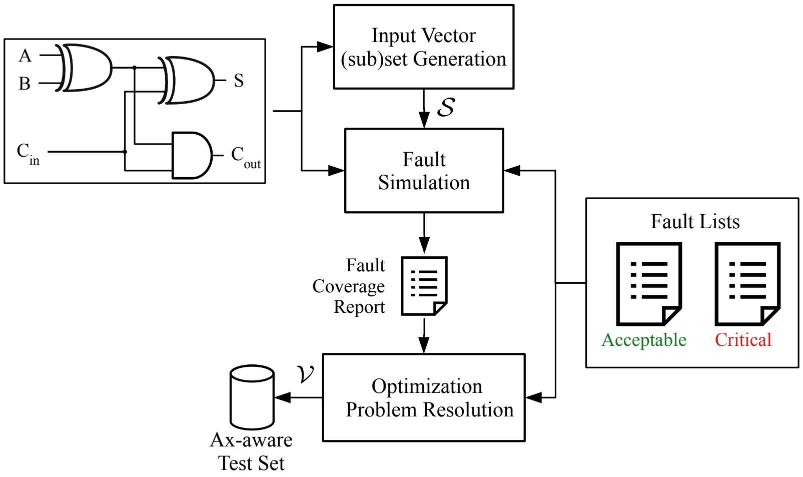


Fig. 12.8: Approximation-Aware (AxA) test generation methodology

AxIC. Therefore, further efforts must be spent in the test application phase, as shown in the next subsection.

12.5.3 AxA test set application

To push further the test outcomes, the third step of AxA testing, the *test pattern application*, comes into play. In the conventional test set application phase, observing a circuit response different from the expected one always leads to reject the circuit. On the contrary, in AxA testing, whether the erroneous response is due to an acceptable fault or to a critical fault must be taken into account. The test must reject the AxIC only if a critical fault caused the error. As shown in the previous subsection, often it is not possible to avoid detecting acceptable faults. Thus, the main solution is to verify, after the test application, whether the detected fault was acceptable or not. Another metric is used to evaluate the effect of the AxA testing procedures on the yield, the *Yield Increase Loss* (YIL), defined below:

$$\text{YIL} = \frac{\text{acceptable faults detected}}{\text{total faults}} \tag{12.8}$$

It describes the value of the yield increase **not achieved** due to the detection of acceptable faults. The YIL is in the range $[0, \text{eYI}]$. By considering Equations 12.5 and 12.8, we can observe that the YIL can be expressed also as follows:

$$\text{YIL} = \text{acceptable FC} \cdot \text{eYI} \tag{12.9}$$

This means that the acceptable FC metric represents the part of the possible yield increase (eYI) that **is not** actually achieved, due to the detection of acceptable faults. Therefore, if acceptable FC = 0 then YIL = 0 (i.e., maximum yield increase). On the contrary, if acceptable FC = 1 then YIL = eYI, thus the achieved yield increase is null.

We need a methodology able to observe the circuit's responses and distinguish between the detection of an acceptable fault (i.e., the test passes) and a critical one (i.e., the AxIC is rejected). Let us observe Table 12.3, which reports the output integer values of the AxIC in Figure 12.6b, obtained in presence of the different faults. For each fault, we report also its classification according to the WCE metric. In bold are reported the value observed when a particular vector (column) detects the presence of a particular fault (row), i.e., there is a difference between the expected output (fault-free) and the obtained output. In particular, we highlight with blue solid-bordered boxes the faulty values obtained by applying the vector 4. In subsection 12.5.2, we observed that vector 4 is a good solution to achieve 100% critical FC for the AxIC in Figure 12.6b. Unfortunately, it also achieves 38% acceptable FC (5 acceptable faults over 13). This means loosing a part of the possible yield gain, meaning $YIL = \frac{5}{16} = 31.25\%$.

Table 12.3: Values of the AxIC in Figure 12.6b when the input vectors are applied in presence of the faults.

Input vector		0	1	2	3	4	5	6	7	Classification
Output	Precise	0	1	1	2	1	2	2	3	WCE (2)
	Fault-free	0	1	1	0	1	2	2	1	
	Sa1@a	**1**	**0**	1	0	$\boxed{\mathbf{2}}$	**1**	2	1	acceptable
	Sa0@a	0	1	**0**	**1**	1	2	**1**	**2**	acceptable
	Sa1@b	**1**	1	**0**	0	$\boxed{\mathbf{2}}$	2	**1**	1	acceptable
	Sa0@b	0	**0**	1	**1**	1	**1**	2	**2**	acceptable
	Sa1@c	**1**	1	1	**1**	$\boxed{\mathbf{2}}$	2	2	**2**	acceptable
	Sa0@c	0	**0**	**0**	0	1	**1**	**1**	1	acceptable
	Sa1@d	**1**	1	1	**1**	$\boxed{\mathbf{0}}$	2	2	**0**	critical
	Sa0@d	0	**0**	**0**	0	1	**3**	**3**	1	acceptable
	Sa1@e	**1**	**0**	**0**	**1**	1	2	2	1	acceptable
	Sa0@e	0	1	1	0	$\boxed{\mathbf{0}}$	**3**	**3**	**0**	critical
	Sa1@h	**1**	1	1	**1**	1	**3**	**3**	1	acceptable
	Sa0@h	0	**0**	**0**	0	$\boxed{\mathbf{0}}$	2	2	**0**	critical
	Sa1@f	0	1	1	0	$\boxed{\mathbf{3}}$	2	2	**3**	acceptable
	Sa1@g	0	**3**	**3**	0	1	2	2	1	acceptable
	Sa1@i	**2**	**3**	**3**	**2**	$\boxed{\mathbf{3}}$	2	2	**3**	acceptable
	Sa0@i	0	1	1	0	1	**0**	**0**	1	acceptable

bold = output value obtained when an input vector (column) detects a fault (row)

□ = value produced by the test set (input vector 4) when detecting a fault

However, by looking at the critical values produced by applying vector 4 (i.e., in presence of Sa1@d, Sa0e, Sa0@h), we can notice that they are different from the values produced when an acceptable fault is present (Sa1@a, Sa1@b, Sa1@c, Sa1@f, Sa1@i). Therefore, if we observe an unexpected value when applying vector 4, depending on its value we can understand whether it is due to an acceptable or to a critical fault. This, in turn, avoids rejecting circuits due to acceptable faults, i.e., it reduces the YIL to 0%.

Starting from this observation, we can build a test application methodology to further improve the test results. The well-know *signature analysis* concept – successfully applied to built-in self-test (BIST) architectures in the seventies [11] and still used in modern BIST architectures – can be applied in this context. The conventional signature analysis approach compacts test responses of a fault-free circuit into a *golden signature* (i.e., the reference behavior). In the test phase, the test responses of the circuit under test are compacted together into a signature (i.e., the actual behavior). Hence, the latter is compared with the golden one. If the two signatures are identical, the circuit under test is considered fault-free; otherwise, a malfunction is detected.

This concept can be applied to AxIC test, as depicted in Figure 12.9. It is divided in two phases, as follows:

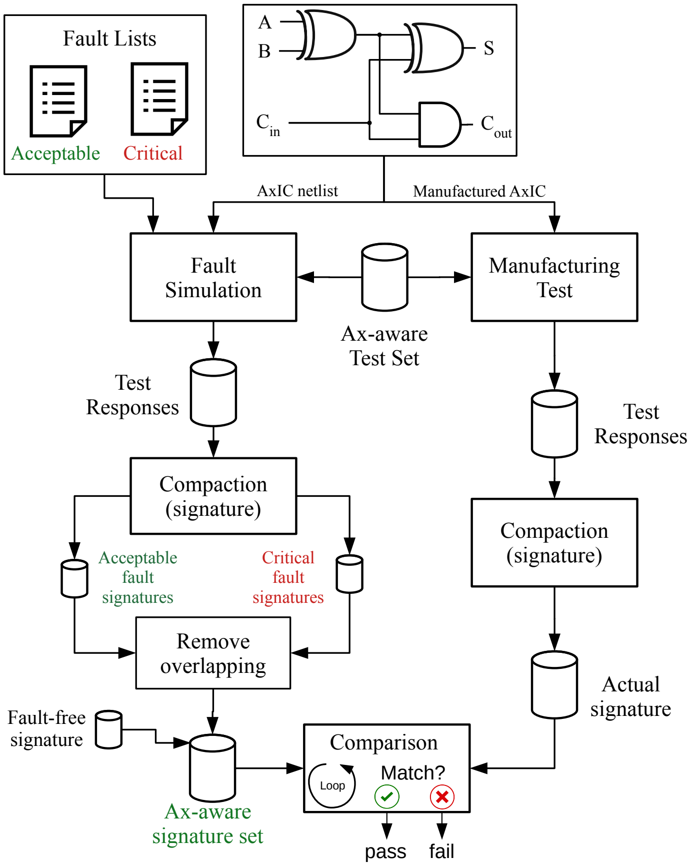


Fig. 12.9: Approximation-Aware (AxA) test application methodology

1. At design time (left branch in Figure 12.9), we perform a fault simulation by using test patterns and the AxIC's faults. For each fault, we compact simulation responses into a signature. We obtain acceptable and critical signatures. We remove from acceptable signatures those overlapping with critical ones (if any). We add the golden-signature (i.e., fault-free) and end up having an *ax-aware signature set*.
2. At test time (right branch in Figure 12.9), manufactured AxIC test responses are compacted into a signature. The latter is compared with the ones in the ax-aware signature set. If there is at least one match, then the AxIC is considered acceptable. Otherwise, the circuit is rejected.

The discussed approach can be used for external test, i.e., test are applied by using an Automatic Test Equipment (ATE) and it can be also adapted to a BIST context. With this approach, results close to the ideal ones are achieved. Further details are available in [41].

12.6 In-field Applications: DNN as case study

Approximate Computing techniques have been positively introduced thanks to the intrinsic resilience of many applications [8]; as a collateral resiliency effect, it has been stated that a resilient application is able to provide good enough outputs (i.e., acceptable) despite of the presence of hardware faults. The previous section described in detail how to characterize the impact of hardware faults on a given approximate circuit resorting to well established metrics, i.e., WCE, MAE, MSE, and EP. In this section, we intend to discuss the usage of the same fault impact characterization, but at application level instead of component/circuit level. Indeed, presented metrics may not be valid at application level and thus new application-dependent metrics and even characterization methods have to be defined. To support the investigation, we resort to a Deep Neural Network (DNN) as case study. The target DNN is the LeNet-5 [16] used as classifier for handwritten digit recognition task. We resort to the MNSIT database of training and validation. The end-goal is to characterize the impact of permanent faults affecting the LeNet through fault injection campaigns. The characterization is done on an original version of the DNN, i.e., without any approximation, and on approximate versions of the same DNN.

12.6.1 DNN Data Type Approximation

Chapter 15 presented several approximation techniques for the deveolpment of approximate neural networks. In this section, we exploit the reduction of the precision and data type for weight and activation's values. More in details, we intend to use custom floating point and fixed-point representations with different precision (i.e., bit-width) at inference time. To explore DNNs data type approximation we leverage the *darknet* open source DNN framework [17]. Implemented in C language, the library allows end-to-end deployment of neural network architectures in a very simple way. It further supplies a very simple environment where several configurations of DNNs, including CNNs, can be executed either to perform training or inference tasks. We modified *darknet* framework to (i) approximate the DNN and (ii) inject Stuck-at Faults at inference time. The targets of injections are the DNN weights. The description of how injection is implemented is out the scope of this chapter, the reader can found all details in [29].

Originally, the *darknet* framework leverages on 32 floating-point data types only. We thus modified the *darknet* source code to allow data type conversions. All the conversions between the standard 32 floating-point and **custom data types** have been carried out by integrating two open source libraries into the *darknet* framework: the *libfixmath* library [27] for managing fixed-point and the *FloatX* library.

Figure 12.10 represents our custom data type defined as following:

- N defines the data bit-width;

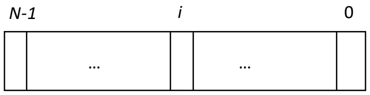


Fig. 12.10: Custom Data Type.

- *i* sets the dynamic and the precision of the data type depending on data representation:
  - Floating Point: *i* is the mantissa width, $N-1-i$ is the exponent width;
  - Fixed Point: *i* is the fractional width, $N-1-i$ is the integer width.

Figure 12.11 sketches the scenario in which the fault injection campaign is executed. The first step allows to determine the **Accuracy Loss** due to approximation. Starting from the trained DNN with the 32 bit floating point data types, called **Golden Standard**, the network is approximated using custom data type representation, called **Golden Custom**. The inference outputs are then compared to determine the **Accuracy Loss** induced by the approximation. Once the Golden Custom network has been built, the fault injection campaign is carried out on this DNN, and the inference outputs are compared with the faulty-free "Golden Custom" inference to first assess the accuracy loss due to fault injection. Eventually, the same outputs are compared with "Golden Standard" ones to assess the inner resiliency due to the approximation.

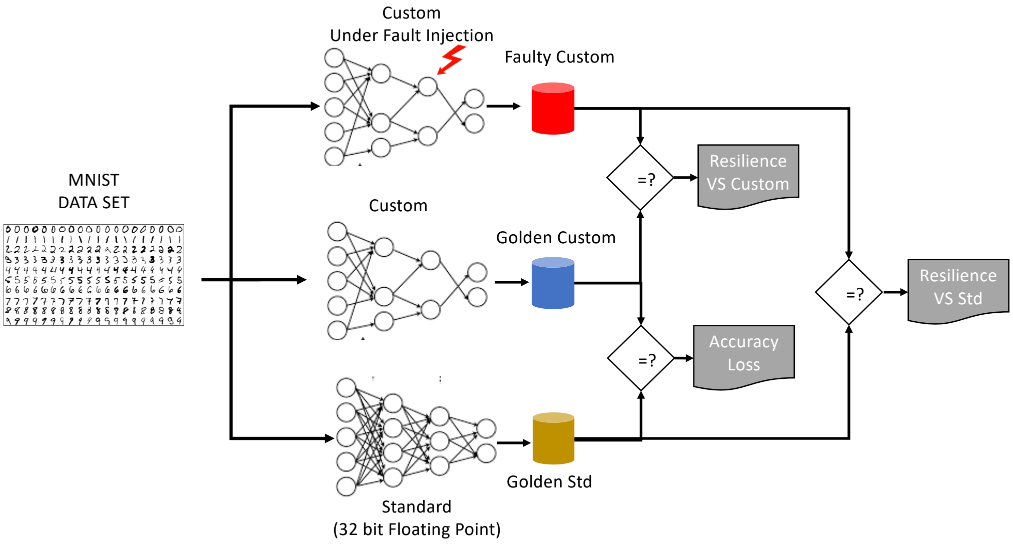


Fig. 12.11: Fault Injection Scenario.

The final purpose of customizing a DNN, i.e., by approximating it, aims at replacing the Standard DNN in edge/resource limited devices. Thus, it is crucial to asses the faults impact on the Custom DNN with respect to the standard DNN. On the other hand, it is also possible to apply the custom-data-type to DNN before training. In this case, the reference DNN to compare the faults impact to be assessed is the Custom DNN itself.

To classify the faults impact, we will define the following application-dependent outcome:

- **Masked:** no difference is observed from the faulty DNN and the golden one.
- **Observed:** a difference is observed from the faulty DNN and the golden one. Depending on how much the results diverge, we further classify these as:
  - **Good:** the confidence score of the top ranked is higher with respect to the golden DNN. In other words, the faulty DNN provides a better inference than the golden one;
  - **Accept:** the confidence score of the top ranked element is reduced by less than 5% with respect to the golden DNN;
  - **Warning:** the confidence score of the top ranked element is reduced by more than 5% with respect to the golden DNN;
  - **Data Corruption:** the top-1 prediction is different. In other words, the faulty DNN makes a wrong inference.

It can be easy seen that the above classification is slightly different than the one of 12.5.1. This is mainly because of the need of considering the final application outputs, and the fact that in some cases the faults impact lead to an improvement of the output quality. The latter point is specific to the case of DNNs. Indeed, a DNN can be considered as an approximation of a given function (in our case a classifier). The approximation is implemented by determining the parameters (i.e., weights) values through the training. Since the training cannot be exhaustive, it is sill possible that a different weights values' distribution provides better results. As mentioned before, in the provided experiments, we target the DNN weigths, so, the faults impact can be seen as a modification of weights values that may improve the output accuracy. For the sake of clarity, we can also classify faults as proposed in 12.5.1 for further comparison:

- Acceptable Faults: Masked $\bigcup$ Observed $\bigcup$ Accept
- Critical Faults: Warning $\bigcup$ Data Corruption

It is important to specify that "Warning" can or cannot be considered as Critical Faults depending on a user threshold.

12.6.1.1 Experiments

For running the experiments, the MNIST database [17], a well-known dataset used to evaluate the accuracy of new emerging models, has been selected. The dataset includes 60,000 images for training and 10,000 for test/validation of the model, encoded in 28 × 28 grayscale pixels. Since we are focusing on the inference phase and on the response of the network in a faulty scenario, a set of pre-trained weights has been adopted. The set comes from the *darknet* website, and defines all the weights as 32-bit floating-point.

To carefully select the custom data type representation, we first analyze the LeNet-5 weight values distribution. All the values are in the range -0.6 to 0.6 with the most of them around zero. From this analysis, we simply deduce that the data type does not need a high dynamic range while a high precision is preferred. We thus select the custom data types reported in Table 12.4.

Table 12.4: LeNet-5 Data Type Accuracy Loss [%]

Scenario	**Data type**	**Bit-width**	**Bit encoding**	**[%] Accuracy Loss**
FP32	floating-point	32	1 sign, 8 exponent, 23 fractional	Ref.
FP16	floating-point	16	1 sign, 5 exponent, 10 fractional	0%
FP8	floating-point	8	1 sign, 4 exponent, 3 fractional	0.02%
FxP32	fixed-point	32	1 integer, 31 fractional	0%
FxP16	fixed-point	16	1 integer, 15 fractional	0%
FxP8	fixed-point	8	1 integer, 7 fractional	0.04%

Two data types are used: the fixed and floating point with different bit width. Moreover, we computed the accuracy loss of the network resulting from the adoption of custom data types weights. As highlighted in Table 12.4, five different scenarios have been analyzed. The second column of the table reports the data type used in each campaign, while the third column reports the bit-width of the weights. The fourth column shows the amount of bits allocated to encode the different part of the number, i.e., sign, exponent, and fractional part in the case of floating point representations, and integer and fractional part in the case of fixed point. To compute the accuracy of the network in the different scenarios, the inference of all the images belonging to the validation set of the MNIST database (10,000 images) have been run on LeNet-5, without injecting any faults, i.e., in a golden scenario. The results show that only when reducing the bit-width to 8 bits the network exhibits an appreciable level of accuracy loss. In details, for the network with weights encoded by using 8-bit floating-point variables (FP8) the accuracy loss was 0.02%, while it was 0.04% when the weights were encoded by using 8-bit fixed-point variables (FxP8).

We evaluated the faults impact by using as reference the Standard 32 bit floating point DNN (see Figure 12.11), and considering the whole set of workloads (10,000 images). This is useful in the case where a designer wants to approximate the DNN (i.e., change its data type and/or bit-width) after that it has been trained.

To discuss the results, we refer to the classification presented in 12.6.1. In particular, we want to evaluate the safety of the different DNN versions, when subject to faults. Therefore, we consider faults in the *Accept*, *Warning*, and *Data Corruption* classes as events reducing the DNN safety; these classes include the faults defined before as *Critical Faults*. The sum of these contributions is referred to as ***Safety***

Decrease. Conversely, we consider the faults in the *Masked* and *Good* classes as events either leaving the safety of the DNN unaltered or improving it. The sum of these contributions is referred to as ***Safety Increase***.

Results are shown in Table 12.5 where each row corresponds to one of the DNN variants (FP32-FP16-FP8-FxP32-FxP16-FxP8 defined in Table 12.4). Each column corresponds to a faulty behavior class as described in 12.6.1.

Table 12.5: LeNet-5 Fault Injection outcomes w.r.t. Golden Std.

CNN	**Observed**				**Masked**	**Safety**		**Gain w.r.t. FP32**	
	Data Corruption	Warning	Accept	Good		Decrease	Increase	Safety*	Memory
FP32	1.32%	0.06%	29.96%	28.98%	39.68%	31.34%	68.66%	-	-
FP16	2.61%	0.12%	53.28%	41.27%	2.71%	56.01%	43.98%	-24.67%	2X
FP8	3.41%	0.91%	64.13%	31.53%	0.02%	68.45%	31.55%	-37.11%	4X
FxP32	0.03%	0.04%	25.93%	25.54%	48.47%	26.00%	74.01%	+5.34%	0
FxP16	0.05%	0.08%	49.98%	46.94%	2.96%	50.11%	49.90%	-18.77%	2X
FxP8	0.45%	0.60%	46.74%	52.18%	0.03%	47.79%	52.21%	-16.45%	4X

* Safety increasing effect difference between a given scenario and FP32

We can firstly note a different resilience to faults depending on the data type. More in detail, **the safety decreasing effect is lower for fixed point than for floating point**, comparing the same bit-width. As representative of this fact, let us discuss scenarios with FP32 and FxP32 (32-bit DNNs): the *safety increasing (decreasing)* effect varies from 69% (31%) of the floating-point version (scenario FP32) to 74% (26%) of the fixed-point version (scenario FxP8). This corresponds to a difference of 5%. The average difference between floating- and fixed-point versions with respect to safety increasing/decreasing effect over the three variants (32- 16- 8- bits) is 10.64%. This can be seen by comparing the scenarios FP32 with FxP32, FP16 with FxP16, and FP8 with FxP8, in terms of the average safety increase/decrease effect variation (columns 7 and 8). In general terms, the safety decreasing effect is critical only in few cases. In fact, the percentage of Data Corruption is always lower than 3.42% for all variants. In particular, fixed-point variants have a very small percentage of critical faults, always lower than 0.46%. Moreover, the increasing contribution of *Good* faults to the safety turns out to be significant, especially for 16- and 8-bit versions. As an example, in the scenario FxP8, we observed a safety increasing effect for 52.21% of the cases, with a 52.18% of *Good* faults.

Furthermore, also the bit width plays an important role for the reliability: **the lower the bit width the lower the resilience**. Therefore, a designer who wants to use a more efficient version of the DNN (reduced memory footprint) has to be aware that it would also be less resilient than the original DNN (FP32). However, it is worth also remarking that using fixed-point data representation, instead of the floating-point counterpart, provides the better results in terms of trade-off between resilience and efficiency. This is reported in the last two columns of 12.5 where the difference in *Safety* and *Memory* footprint is repported considering the FP32 DNN as the reference. For instance, we may compare scenarios FP8 and FxP8 (8-bit DNNs, 1 bit for integer and 7 bit for fractional): we observed a safety loss compared with the FP32 DNN of 37% in the floating-point version (FP8) and only of 16% in the fixed-point version (FxP8). Therefore, choosing the DNN in the scenario FxP8 allows the designer to compact the memory footprint by a 4x factor while reducing the safety only by 16%. By looking more closely, the occurrence of critical faults in scenario FxP8 even decreases from 1.32% of FP32 to 0.45%, while in the case of the floating-point scenario (FP8) it increases to 3.41%. Additionally, for scenario FxP32 (32-bit fixed-point DNN), it has been observed that the DNN achieves 5% improved safety with respect to the FP32 scenario for the same memory footprint. Thus, simply changing the DNN data-type to fixed-point improves its resiliency.

12.6.1.2 Discussion

The above results have been obtained resorting to fault injection campaigns that are highly time consuming because of the huge number of faults that have to be considered. To reduce the number of faults, we consider the layers on the LeNet-5 that perform arithmetic computations involving the trained weights, i.e., the two Convolutional and the two Fully Connected layers. Indeed, we consider the resilience of the DNN against faults affecting the memory, where the weights are stored.

Table 12.6 provides details about the configuration as well as the fault list of each layer. The first two rows (labeled *“Layer"* and *“Detail"*) of the table present the target layers; the third one (*“Connections"*) specifies the amount of their connection weights. The number of possible faults is computed as the multiplication between the connections number (*“Connections"*) and the weight size (*“Bit-width"*).

Table 12.6: LeNet-5 Fault List for Injection Campaigns

	Layer	**L0**	**L2**	**L4**	**L6**
	Detail	Convolutional	Convolutional	Fully Connected	Fully Connected
	Connections	2,400	51,200	3,211,264	10,240
Scenarios FP32, FxP32	Bit-width	32	32	32	32
	#Faults	76,800	1,638,400	102,760,448	327,680
	#Injections	13,678	16,474	16,638	15,837
Scenarios FP16, FxP16	Bit-width	16	16	16	16
	#Faults	38,400	819,200	51,380,224	163,840
	#Injections	11,610	16,310	16,636	15,107
Scenarios FP8, FxP8	Bit-width	8	8	8	8
	#Faults	19,200	409,600	25,690,112	8,1920
	#Injections	8,915	15,991	16,630	13,831

As the rows *“#Faults"* point out, the overall number of possible faults is very high and this reflects in a non-manageable fault injection campaign execution time. Thus, in order to reduce the fault injection execution time, we can randomly select a subset of faults. To obtain statistically significant results with an error margin of 1% and a confidence level of 99%, an average of 15.6k fault injections have to be considered for 32-bit scenarios (FP32 and FxP32), 15k for 16-bit scenarios (FP16 and FxP16), and 13.8k for 8-bit scenarios (FP8 and FxP8). The precise numbers are given in the rows of Table 12.6 labeled *#Injections"* and they have been computed by using the approach presented in [18]. In details, we resorted to the following formula:

$$fault\_injections = \frac{N}{1 + e^2 \times \frac{N-1}{t^2 \times 0.25}} \tag{12.10}$$

where N is the total number of fault locations (i.e., row *#Faults* of Table 12.6), e is the desired error margin (1%), and t depends on the desired confidence level (t=2.58 corresponds to 99% confidence level [18]). Equation 12.10 has an horizontal asymptotic value ($N \rightarrow \infty$) equal to 16,641, thus limiting the number of fault injections necessary to achieve an evaluation with an error margin of 1% and a confidence level of 99%. Moreover, it is worth underlining that the injections are performed by randomly selecting the bit to inject among all the bits of all the connection weights.

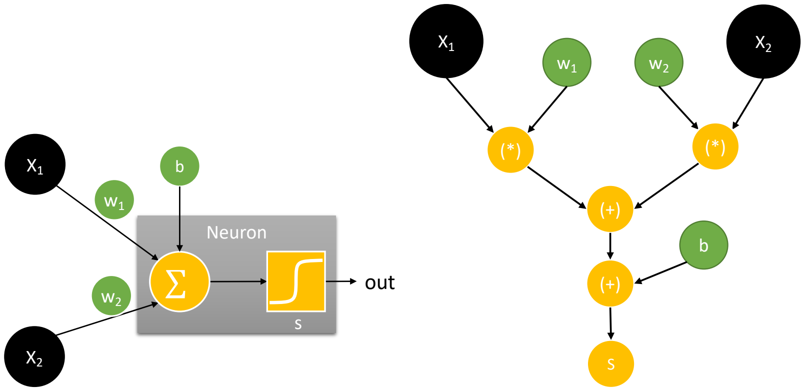
Despite the fact that we used Equation 12.10, it is quite clear that for bigger DNNs the faults number can literally explode. In the next subsection we thus propose a novel method to avoid the need of carrying out fault injection campaign on the whole DNN.

12.6.2 Probabilistic Approach

In the previous section, we showed how we could investigate the fault effect within the system under analysis using fault injection techniques and quantify the deviation with respect to the expected results. In the literature, several other fault injection approaches exist [25]. The common drawback of existing techniques is the high cost of fault injection campaigns in terms of time. We conclude this chapter by presenting a stochastic approach to predict the impact of faults on a DNN's accuracy. The proposed approach builds a Bayesian model of the neural network and, by analyzing the network using the Bayesian inference theory, estimates the neural network's error distribution.

The idea is to first model the DNN topology as a Bayesian Network (BN). In BN, the nodes represent random variables, each defined over a set of states, while edges model the conditional relationships. Figure 12.12b shows a simple NN neuron having two inputs and processing the results using a sigmoid activation function. The neuron can be modeled based on the data flow, having inputs, weights, and biases as data sources, which feed a set of multiplications and sums to produce a final result that is the input of the activation function. Figure 12.12 reports the transformation into a BN model, having the data and the operators play as nodes and the edges modeling the flow between nodes of the network.

Figure 12.12 includes three color classes of nodes: the black ones represent the input of the node, the green ones represent weights and biases, and the yellow ones are the intermediate nodes representing the mathematical operation required



(a) Neuron with sigmoid activation function (b) Neuron's Bayesian network model

Fig. 12.12: Two input neuron Bayesian Network example

within the neuron. It is easy to notice how the graph in Figure 12.12 has all yellow nodes depicting the multiplications, sums, and sigmoid function response, with the edges showing the data flow. The color distinction is necessary to handle the neuron behavior in Bayesian Theory terms properly. In fact, every single node must be a random variable to be compliant with the Bayesian Theory. To define the states representing the behavior of every type of node in the neuron model, we propose a classification approach to the tricky point of DNN faults impact assessment because not all deviations of the output lead to an inference error. The classification flows what already published in [30, 34, 43]. We classify the data associated with the node, i.e., the value of an input or weight or the result of multiplication, with the possibility of representing three different error's states:

- *Masked* (M), when the value is error-free regardless of the HW fault;
- *Acceptable* (A), when the introduced error remains under a user-defined threshold (α);
- *Critical* (C), when the introduced error rises above the threshold, making the value not acceptable.

The use of a user-define threshold allows to support a flexible evaluation, which can be adapted to the specific problem. The data error ($\tilde{y}$), easily evaluable by difference with the faulty free one, reflect the classification using Equation 12.11:

$$Class(\tilde{y}) = \begin{cases} M \text{ (Masked)} & if\ \epsilon_y = 0 \\ A \text{ (Acceptable)} & if\ \epsilon_y \leq \alpha \\ C \text{ (Critical)} & if \epsilon_y > \alpha \end{cases} \tag{12.11}$$

Resorting to the classification in Equation 12.11, the model can include the necessary set of Conditional Probability Tables (CPTs) that should describe each node's probabilistic behavior, i.e., express every node's probability belonging to each of these classes, eventually conditioned to the inputs states. Figure 12.13 shows how the classification translates in terms of information. The reported two tables demonstrate the two types of CPT associated with a node of the BN. The first one (the CPT associated with X_2) displays how black and green nodes of the network are probabilistically characterized: three probabilities define the distribution of all possible faulty values among the three classes. This table is easily computed following an enumeration approach.

Since ϵ_y depends on the actual value, we already know that in a DNN this value is fixed for weights and biases or is distribution-based for inputs. Therefore, for every single value, we can compute the probability of having $\tilde{y}$ in M, A, or C, from all possible faults ($f \in F$), as in Equation 12.12.

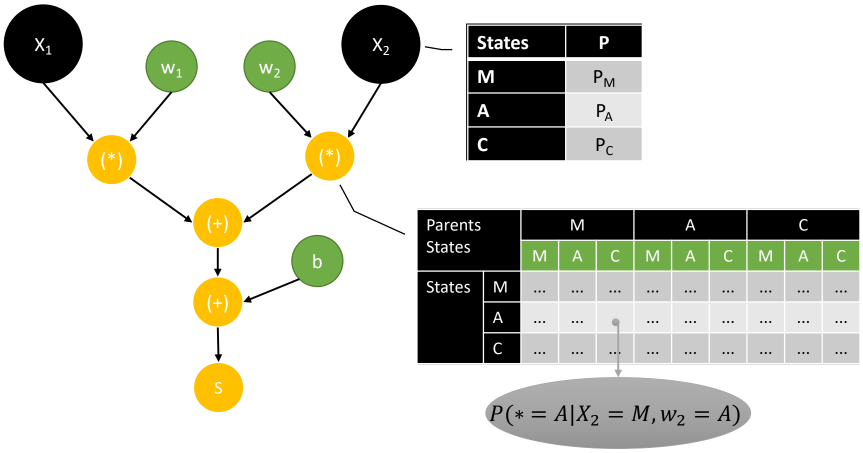
$$\begin{aligned} P(\tilde{y}\ is\ M) &= P(\epsilon_y|_f\ = 0) = \frac{\sum_{\forall F \rightarrow \epsilon_y|_f\ )=0} 1}{\#F} \\ P(\tilde{y}\ is\ A) &= \ P(\epsilon_y|_f\ ) \leq \alpha) = \frac{\sum_{\forall k \rightarrow \epsilon_y|_f\ \leq \alpha} 1}{\#F} \\ P(\tilde{y}\ is\ C) &= \ P(\epsilon_y|_f\ ) > \alpha) = \frac{\sum_{\forall k \rightarrow \epsilon_y|_f\ > \alpha} 1}{\#F} \end{aligned} \tag{12.12}$$

This evaluation perfectly stands for all fixed values, while for distributed values, the evaluation has to spawn over all possible values in the distribution (or a statistically significant subset). The second CPT in Figure 12.13 refers to the operators, i.e., operations and functions, case (the CPT associated with the multiplication $(*)$). When the random variable depends on other random variables, i.e., its parents, those probabilities are defined for each possible combination of states of its parents[30]. In the BN model, all operations and functions (yellow nodes) are always nodes that depend on the states of their input to produce the output. In our example, we have two operations, i.e., sum and multiplication, and one function, i.e., the sigmoid ($S(x)$ in Equation 12.13).

$$S(x) = \frac{1}{1 + e^{-x}} \tag{12.13}$$

Since the input error can be either masked or amplified, depending on how the operation handles the inputs, the operations and function characterization determine how this propagation occurs. In general, the manipulation done by an operator may change the way the results of the application distribute among the three accuracy classes (M, A, or C). It is also crucial to understand that those probabilities are independent of the probability expressed by the input states. In the sense that they reflect the probability of the node being in a particular state knowing that its parents express a specific combination of states (as in Figure 12.13 the $P(* = A)$ expresses the probability of the output of the multiplication to be an acceptable value knowing that X_2 value belongs to Masked and w_2 value to Acceptable).

All the yellow nodes' CPTs have been evaluated through simulations, one per type. In detail, we characterize the operator's output when we feed as input values corresponding to the three error classes. In order to produce those CPTs, we devise a computational task that characterizes a library of **faulty** operators quantifying the error introduced by the occurrence of a fault. The most important message here is that this operator characterization has to be done **only one time for each operator** for each α threshold that we might need to evaluate. The consequence is that even if the target DNN has thousands of neurons, we need to work on a single operator.



States	P
M	P_M
A	P_A
C	P_C

Parents States		M			A			C		
		M	A	C	M	A	C	M	A	C
States	M	...	...	...	...	...	...	...	...	...
	A	...	...		...	...	...	...	...	...
	C	...	...	...	...	...	...	...	...	...

$P(* = A|X_2 = M, w_2 = A)$

Fig. 12.13: Bayesian Conditional Probability Tables meaning example

After the model is built, it is time to use the Bayesian theory to evaluate it. The Bayesian inference allows the analysis of the model to predict the approximation at the output of the DNN [5]. We compute the posterior probability of the leaves of the network to be in one of the three error classes defined in equation 12.11 to have the necessary prediction. We used a publicly available BN library and engine [3] to implement the Bayesian inference.

To demonstrate the modeling's prediction capability when applied to DNNs, we compared the iterative exploration against the BN model. The comparison covers different α values. This process requires generating the CPT tables of all operators for each α, and then use them on the BN modeled only once (by replacing the CPTs). The experimental setup works under the hypothesis of having the faults appearing into the weights and biases memory; thus, the weights and biases distribution have been computed and assigned to the proper nodes of the CPT. Table 12.7 shows the three classifications for all three α used.

Reported results confirm the reliability of the BN prediction, showing a maximum percentage point error of 5.42pp. The variability in the absolute error reflects the data dependency of the multiplication. Nonetheless, since the outcome comes from a non-linear function such as the sigmoid, it is interesting to notice that we could adequately model it using the BN. Moreover, having the three alpha expressing a considerable variation in the quality tolerance, the absolute errors confirm the BN prediction's precision with respect to a whole fault injection campaign.

Table 12.7: Prediction Average Error

	Prediction Classes (avg. error)		
α	**M**	**A**	**C**
1	0.0477	0.0477	0.0000
0.1	0.0675	0.0030	0.0705
0.0001	0.0845	0.0049	0.0893

12.7 Conclusions

This chapter presented an overview of different approaches to handle the design, verification, testing and in-field operation of approximate computing systems. The presented solutions are not exhaustive and new publications and approaches will appear while the field becomes mature. The chapter leverages on the experience of the authors to overview the major challenges that still represent a barrier to transform this interesting research field into real solutions ready to the market.

Acknowledgements *This work was partly supported by the Czech Science Foundation project 21-13001S.*

References

1. V.D. Agrawal, S.C. Seth, and IEEE Computer Society. *Tutorial test generation for VLSI chips*. Computer Society Press, 1988.
2. Lorena Anghel, Mounir Benabdenbi, Alberto Bosio, Marcello Traiola, and Elena Ioana Vatajelu. Test and reliability in approximate computing. *Journal of Electronic Testing*, 34(4):375–387, Aug 2018.
3. BayesFusion, LLC. SMILE Engine.
4. A. Bosio, S. D. Carlo, P. Girard, E. Sanchez, A. Savino, L. Sekanina, M. Traiola, Z. Vasicek, and A. Virazel. Design, verification, test and in-field implications of approximate computing systems. In *2020 IEEE European Test Symposium (ETS)*, pages 1–10, 2020.
5. George EP Box and George C Tiao. *Bayesian inference in statistical analysis*, volume 40. John Wiley & Sons, 2011.
6. Milan Ceska, Jiri Matyas, Vojtech Mrazek, Lukas Sekanina, Zenek Vasicek, and Tomas Vojnar. Adaptive verifiability-driven strategy for evolutionary approximation of arithmetic circuits. *Applied Soft Computing*, 95:1–17, 2020.
7. Arun Chandrasekharan, Mathias Soeken, Daniel Große, and Rolf Drechsler. Precise error determination of approximated components in sequential circuits with model checking. In *Proc. of DAC'16*, pages 1–6. ACM, 2016.
8. V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–9, May 2013.
9. Vinay K Chippa, Srimat T Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, page 113. ACM, 2013.

10. Richard D. Eldred. Test routines based on symbolic logical statements. *J. ACM*, 6(1):33–37, January 1959.
11. Robert A. Frohwerk. Signature analysis: a new digital field service method. 1977.
12. J. Han and M. Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*, pages 1–6, May 2013.
13. P. Kulkarni, P. Gupta, and M. Ercegovac. Trading accuracy for power with an underdesigned multiplier architecture. In *2011 24th Internatioal Conference on VLSI Design*, pages 346–351, Jan 2011.
14. M L. Bushnell and V D. Agarwal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. 01 2000.
15. M L. Bushnell and V D. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. 01 2000.
16. Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.
17. Yann LeCun et al. The mnist database., 2020.
18. R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 502–506, April 2009.
19. J. Liang, J. Han, and F. Lombardi. New metrics for the reliability of approximate and probabilistic adders. *IEEE Transactions on Computers*, 62(9):1760–1771, Sept 2013.
20. J. Miao, A. Gerstlauer, and M. Orshansky. Approximate logic synthesis under general error magnitude and frequency constraints. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 779–786, Nov 2013.
21. J. Miao, A. Gerstlauer, and M. Orshansky. Multi-level approximate logic synthesis under general error constraints. In *2014 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 504–510, Nov 2014.
22. Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, March 2016.
23. Sparsh Mittal. A survey of techniques for approximate computing. *ACM Comput. Surv.*, 48(4):62:1–62:33, March 2016.
24. A. Momeni, J. Han, P. Montuschi, and F. Lombardi. Design and analysis of approximate compressors for multiplication. *IEEE Transactions on Computers*, 64(4):984–994, April 2015.
25. Giorgio Di Natale, Dimitris Gizopoulos, Stefano Di Carlo, Alberto Bosio, and Ramon Canal, editors. *Cross-Layer Reliability of Computing Systems*. Institution of Engineering and Technology, October 2020.
26. [Online]. Tetramax. https://www.synopsys.com/.
27. [Online]. Libfixmath library. https://github.com/Petteri-Aimonen/libfixmath, 2020.
28. A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan. Aslan: Synthesis of approximate sequential circuits. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2014.
29. Annachiara Ruospo, Alberto Bosio, Alessandro Ianne, and Ernesto Sanchez. Evaluating convolutional neural networks reliability depending on their data representation. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 672–679, 2020.
30. Alessandro Savino, Marcello Traiola, Stefano Di Carlo, and Alberto Bosio. Efficient neural network approximation via bayesian reasoning. In *2021 24th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 45–50, 2021.
31. Lukas Sekanina, Zdenek Vasicek, and Vojtech Mrazek. *Automated Search-Based Functional Approximation for Digital Circuits*, pages 175–203. Springer International Publishing, 2019.
32. D. Shin and S. K. Gupta. Approximate logic synthesis for error tolerant applications. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 957–960, March 2010.
33. D. Shin and S. K. Gupta. A new circuit simplification method for error tolerant applications. In *Design, Automation Test in Europe (DATE)*, pages 1–6, March 2011.

34. M. Traiola, A. Savino, M. Barbareschi, S. D. Carlo, and A. Bosio. Predicting the impact of functional approximation: from component- to application-level. In *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pages 61–64, 2018.
35. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Towards digital circuit approximation by exploiting fault simulation. In *2017 IEEE East-West Design Test Symposium (EWDTS)*, pages 1–7, Sep. 2017.
36. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Investigation of mean-error metrics for testing approximate integrated circuits. In *2018 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, Oct 2018.
37. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. On the comparison of different atpg approaches for approximate integrated circuits. In *2018 IEEE 21st International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 85–90, April 2018.
38. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Testing approximate digital circuits: Challenges and opportunities. In *2018 IEEE 19th Latin-American Test Symposium (LATS)*, pages 1–6, March 2018.
39. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Testing integrated circuits for approximate computing applications. In *4rd Workshop On Approximate Computing (WAPCO)*, pages 1–7, Jan. 2018.
40. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. A test pattern generation technique for approximate circuits based on an ilp-formulated pattern selection procedure. *IEEE Transactions on Nanotechnology*, pages 1–1, 2019.
41. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Maximizing yield for approximate integrated circuits. In *2020 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2020.
42. M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. A survey of testing techniques for approximate integrated circuits. *Proceedings of the IEEE*, 108(12):2178–2194, 2020.
43. Marcello Traiola, Alessandro Savino, and Stefano Di Carlo. Probabilistic estimation of the application-level impact of precision scaling in approximate computing applications. *Microelectronics Reliability*, 102:113309, 2019.
44. Zdenek Vasicek. Formal methods for exact analysis of approximate circuits. *IEEE Access*, 7(1):177309–177331, 2019.
45. Zdenek Vasicek and Vojtech Mrazek. Trading between quality and non-functional properties of median filter in embedded systems. *Genetic Programming and Evolvable Machines*, 18(1):45–82, 2017.
46. S. Venkataramani, K. Roy, and A. Raghunathan. Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1367–1372, March 2013.
47. S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. Salsa: Systematic logic synthesis of approximate circuits. In *DAC Design Automation Conference 2012*, pages 796–801, June 2012.
48. S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. Salsa: Systematic logic synthesis of approximate circuits. In *DAC Design Automation Conference 2012*, pages 796–801, June 2012.
49. I. Wali, M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Can we approximate the test of integrated circuits? In *3rd Workshop On Approximate Computing (WAPCO)*, pages 1–7, Jan. 2017.
50. I. Wali, M. Traiola, A. Virazel, P. Girard, M. Barbareschi, and A. Bosio. Towards approximation during test of integrated circuits. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, pages 28–33, April 2017.
51. Laung-Terng Wang, Cheng-Wen Wu, and Xiaoqing Wen. *VLSI Test Principles and Architectures 1st Edition*. Elsevier, 2006.

52. Y. Wu and W. Qian. An efficient method for multi-level approximate logic synthesis under error rate constraint. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.
53. Q. Xu, T. Mytkowicz, and N. S. Kim. Approximate computing: A survey. *IEEE Design Test*, 33(1):8–22, Feb 2016.