

Advancing Testbench Efficiency and Automation Based on Machine Learning Approach

Rajender Kumar¹, Sayan Mukherjee²

¹Assistant Professor, Electronics and Communication Engineering Department, NIT Kurukshetra, India.

²M. Tech Student, School of VLSI Design and Embedded Systems, NIT Kurukshetra, India.

Using random testbench variables is one of the most important techniques employed by the DV engineers during the Design Verification (DV). Before releasing the Design Under Test (DUT), 100% functional coverage must be ensured. In this dissertation report, methodology is proposed to increase the randomization efficiency by Machine Learning algorithm generated in C++ and integrated in SV/UVM Testbench by the use of DPI-C Call. Also the process to check the FIFO event at the EoS is also automated which reduced the manual effort required. Using ML algorithm in SV/UVM Testbench, an improvement of 50% over conventional methods is also observed. Also, the timing improvement of ML based algorithm over conventional search algorithms, a 50 times improvement was observed. The repetition of random value in multiple seed regression is also improved by 33% for lower number of seed and 0% of higher seeds. The automation of to check FIFO events reduced the manual effort requirement significantly.

Keywords: DUT, FPGA, VLSI, ML, SoC, KNN, System Verilog.

1. Introduction

Traditional coverage closure methods face scalability challenges, delaying product development. Traditional coverage closure methods face scalability challenges, delaying product development. Leveraging machine learning, the proposed approach optimizes testbench configurations, accelerates simulations, and reduces verification time, aiming to revolutionize coverage closure processes in digital system design. Automation enhances the design verification process, such as verifying the FIFO is empty at the end of a simulation. Ensuring all values are extracted from the Design Under Test (DUT) and match expected golden data is crucial for confirming functionality. This step validates the FIFO and the system's integrity. Automating this verification reduces tediousness, allowing engineers to trust the design's reliability and identify potential issues early, streamlining the overall verification process.

RTL design verification is integral to the VLSI industry, ensuring every design is verified before integration into silicon at Taiwan Semiconductor Manufacturing Company (TSMC).

Pre-Silicon Verification is essential for identifying design bugs in ASICs and FPGAs. Using System Verilog testbenches with constraint-based randomization, designs are rigorously tested. Randomization enhances thoroughness and efficiency, generating diverse input stimuli and uncovering elusive bugs. The verification environment includes components such as stimulus generators, monitors, checkers, and coverage mechanisms, emulating real-world scenarios to validate hardware functionality. Advanced features like constrained random testing improve modularity and reusability.

Machine Learning (ML) optimizes verification processes, predicting system behavior and automating tasks. A supervised ML model integrated into the SV/UVM testbench has shown significant improvement in coverage closure turnaround time across various EDA tools.

Automation is crucial in bug-catching, efficiently managing complexity. Automated tools execute numerous test cases, ensuring consistency and rapid detection. A Python script generates System Verilog files from FIFO signals, integrating seamlessly with EDA tools, and enhancing verification efficiency by reducing manual labor.

2. Literature Review

In this part of the report, we discuss the methodologies currently used in Design and Verification (DV) for calculating the Turnaround Time (TAT) of coverage closure. Companies like Synopsys have developed their own approaches to improve coverage closure speed using machine learning (ML) techniques. We will also highlight the drawbacks of these approaches and explain how our ML approach addresses these issues. Specifically, we have used the K-nearest neighbor algorithm in our supervised machine learning approach.

Approximate nearest neighbor (ANN) search is vital in computer vision and deep learning, benefiting from GPU-based implementations like PQT, FAISS, and SONG. This paper introduces GGNN, a novel GPU-friendly search structure leveraging nearest neighbor graphs and information propagation to accelerate index construction and querying. Empirical results demonstrate GGNN's superiority in build-time, accuracy, and search speed over current state-of-the-art systems. [4]

By integrating the proposed ML approach with the K-nearest neighbor algorithm, TAT of coverage closure is optimized, addressing the challenges faced by existing methodologies and improving the efficiency and accuracy of the verification process.

The author in the [1] describes digital circuits for space applications are prone to radiation-induced failures. A formal verification method using the VeriHard tool ensures fault tolerance techniques like TMR, DwC, and Hamming encoding prevent fault propagation and maintain functional equivalence. Experimental results demonstrate the method's high performance and effectiveness in industrial cases.

ML approach used by other EDA vendors

In the ML approach by EDA vendor (VSO.ai) tool have used AI/ML approach to automate tasks and improve the TAT of coverage closure. The commonly known problem was addressed that many of the random tests are the same and have little to no impact on the coverage closure metrics. In the VLSI industry manual regression is not a good solution

as because of its fragile nature but multiple seeds regression is a necessity. Synopsys have integrated the VSO.ai to their VCS Simulator in the following way.[5]

Step 1: After running regression in the randomization starts and Vdb files are generated for each test.

Step 2: Check if the coverage is 100%. If yes, then stop the process or else move to step 3.

Step 3: Merge the Vdb files created and the pass it to the ML model developed by them.

Step 4: The ML model will update the constraint solver of the VCS tool and run the randomization again with the new constraints of the variable.

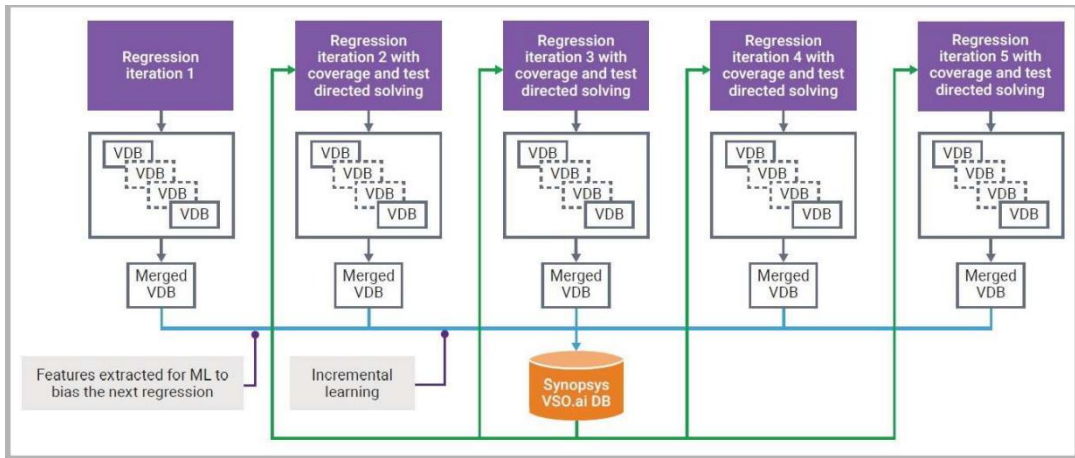


FIGURE 2.1: ML approach by EDA vendor

Key features and constraints of this approach:

It is greatly dependent on the Synopsys VCS tool and cannot be used with other EDA tools and simulators. The tool requires license and is costly. So, any small organization or any individual person cannot use this solution because of affordability issue.

To update the constraint solver, post processing of regression data is required and cannot be done on the fly (during the process of randomization). Any change in the RTL will render the merged Vdb useless and the same process must be repeated. This results in waste of time and resources.

This solution cannot be integrated with the existing setup and changing current setup to a new in the industry is a challenge in itself.

Related Work:

Writing a test bench is challenging task and requires precise knowledge of the design architecture and executing it correctly is one the most important aspect. There are

TABLE 2.1: Data from existing methodology

Functional Coverage	76.2%
TB development time	<= 75%
Reduction in resource utilization	91.6%
Development Efficiency	50 times

many areas where test bench requires automation and improved performance because it is greatly dependent on the traditional method of writing as mentioned in the Language reference manual of the System Verilog language. The tasks like checking for functional coverage and debugging requires a lot of manual work which does not add to the quality of work and is tedious in nature. Moreover, the use of manual work for a process like debugging leaves a chance that a bug may persist. There is a paper which discusses a methodology named TestQuBE (Testbench Quality Benchmark Enhancement) which targets the total TB development time, the total resources used, checking for functional coverage and the development efficiency. This proposed an improvement of at least 75% of each the above metrics.[11] The total improvement found is:

Another paper [2] discusses about the connectivity gap between high-level synthesis (HLS) design and a low-level RTL design. Now RTL Design is an important aspect of FPGA integration. The code must be synthesizable to be implemented in the FPGA. For this HLS is often used to generate the synthesizable code. The FIFO generated by the HLS is not always accurate and contains stale data in them (stale data is the presence of data inside FIFO at the end of simulation). This presence of stale data is not desired and considered as a bug inside the design which needs to be fixed at the earliest. The paper further discusses about the deadlock issue of the FIFO which arises only when the depth of the FIFO is less than the latency difference among the different modules of the design. Each module may have different latency and during interaction of the module this deadlock problem may arise.

Automated approach to FIFO

As discussed previously, the automation of testbenches has become imperative, with the adoption of automated testbenches rising rapidly. One of the significant challenges encountered is the time-consuming nature of debugging FIFOs, particularly due to the possibility of transitions occurring at any positive clock edge. To mitigate this issue, the team has devised a solution involving the verification of FIFO events at the End of Simulation (EOS).

This solution entails writing coverpoints for each FIFO and examining the empty and full conditions. Any discrepancies identified in these conditions are flagged as errors, pinpointing the exact FIFO containing stale data. Additionally, employing coverpoints for each FIFO and verifying coverage ensures that all possible conditions have been traversed, and there is no stale data present at EOS.

The entire process has been automated through a Python script, significantly reducing the time investment required. The forthcoming report will provide a detailed exposition of the targeted problem and the solution implemented. Moreover, it will outline the improvements

realized through this automated approach.

3. Theory

Introduction

In the modern world of VLSI design, functional coverage is an important metric. Therefore functional coverage efficiency must be reduced by reducing the number of random tests in multiple seed regression. Also design contains multiple FIFO which is required because the multiple components in SoC runs on different clock speed. To store excess data FIFO is required. Now at the beginning and end of simulation FIFO must be empty and FIFO must be full during simulation. Here in this section we will discuss some theories related to FIFO and why they are required and also about the KNN algorithm used to improve functional coverage.

K-Nearest Neighbour algorithm

K-nearest neighbor (kNN) search is essential in data mining. This paper introduces random projection forests (rpForests), combining multiple kNN-sensitive trees via random projections for high accuracy and low computational complexity. Easily parallelizable, rpForests demonstrates remarkable performance and scalability, with theoretical insights on ensemble effectiveness.[2]

Concept of K-Nearest Neighbor algorithm

K-Nearest Neighbors (KNN) is a foundational classification algorithm in the realm of Machine Learning, widely applied in various domains such as pattern recognition, data mining, and intrusion detection. It falls under the category of supervised learning and proves to be highly versatile in real-world scenarios. What sets KNN apart is its non-parametric nature, meaning it doesn't impose assumptions about the underlying data distribution. This stands in contrast to other algorithms like Gaussian Mixture Models (GMM), which rely on specific data distribution assumptions. The flexibility of KNN makes it particularly valuable when dealing with diverse and complex datasets. In practical terms, KNN utilizes prior data, often referred to as training data, to categorize coordinates into groups based on specific attributes. This method excels in scenarios where the relationships within the data are not easily characterized by predefined mathematical models. The simplicity and effectiveness of the K-Nearest Neighbors (KNN) algorithm make it a go-to choice in scenarios where data relationships are intricate and traditional assumptions may not hold. Its strength lies in its ability to classify new data points by examining the attributes of their closest neighbors in the existing dataset. This intuitive approach aligns well with real-world applications, where understanding the context and relationships within the data is crucial. KNN's versatility makes it suitable for scenarios where data distribution is not explicitly known, allowing it to adapt to a wide range of use cases and making it a valuable tool in the machine learning toolkit.[2]

Consider, for example, a dataset with two features, where KNN can effectively group data points based on their proximity to each other in the feature space, illustrating the algorithm's simplicity and applicability.

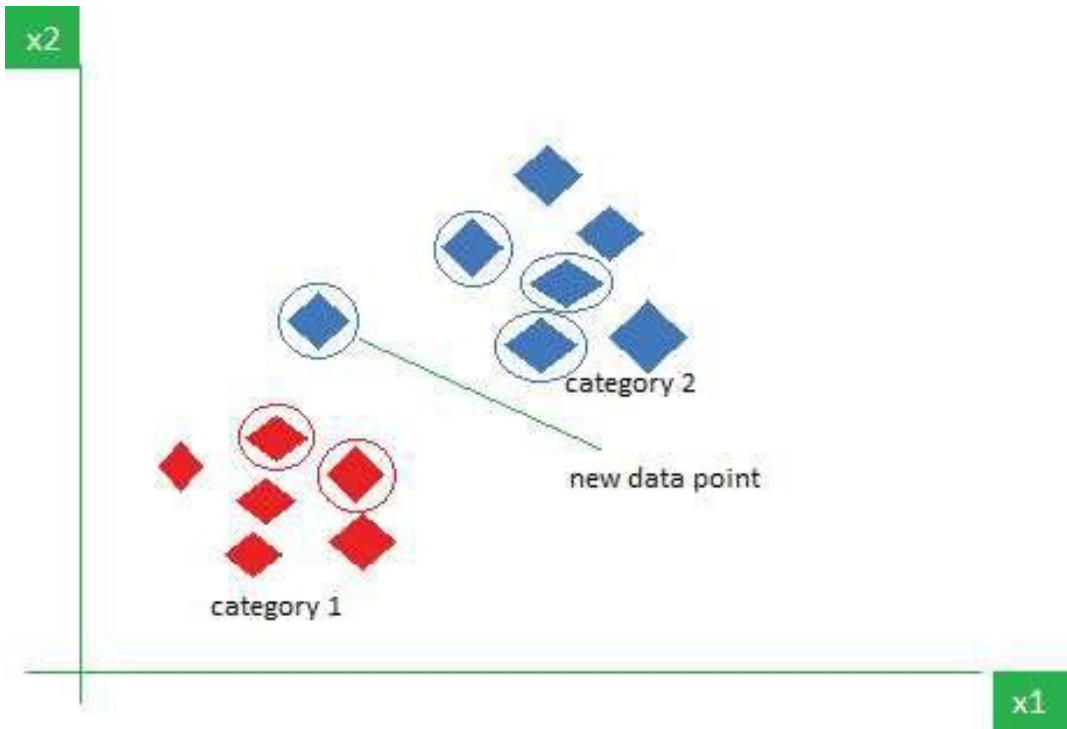


FIGURE 3.1: KNN Algorithm working visualization

Deciphering the KNN Algorithm

Understanding how the K-Nearest Neighbors (KNN) algorithm operates involves grasping its intuitive approach to categorizing data points based on the majority class of their closest neighbors. This intuitive framework is particularly valuable for making sense of complex patterns within datasets. By plotting data points on a graph, discernible clusters or groups often emerge. When confronted with an un-classified point, assignment to a group is determined by observing the classifications of its nearest neighbors. For instance, a point in proximity to a cluster labeled as 'Reel' has an increased likelihood of being classified as 'Reel' itself. Intuitively, this method suggests that the first point (2.5, 7) should be categorized as 'Green,' while the second point (5.5, 4.5) is more fittingly classified as 'Red.'

Essentiality of the KNN Algorithm

The K-Nearest Neighbors (K-NN) algorithm stands out as a versatile and widely applied machine learning technique, prized for its simplicity and straightforward implementation. Unlike some algorithms, K-NN operates without presumptions about the underlying data distribution, accommodating both numerical and categorical data types. Its adaptability makes it a preferred choice for diverse datasets in tasks ranging from classification to regression. As a non-parametric method, K-NN relies on the similarity of data points within a dataset, demonstrating reduced sensitivity to outliers in comparison to alternative algorithms. In practical terms, the K-NN algorithm identifies the K nearest neighbors to a given data point based on a chosen distance metric, such as Euclidean distance. The classification or value of

the data point is then determined by the majority vote or average of the K neighbors. This intrinsic adaptability empowers the algorithm to navigate diverse patterns within the data, making predictions grounded in the local structure of the dataset.

Distance calculation in KNN Algorithm

Method	Formula
Euclidean Distance	$\text{distance}(x, X_i) = \sqrt{\sum_{j=1}^d (x_j - X_{ij})^2}$
Manhattan Distance	Manhattan distance = $\sum_{i=1}^n x_{1i} - x_{2i} $
Minkowski Distance	Minkowski distance = $(\sum_{i=1}^n x_{1i} - x_{2i} ^p)^{1/p}$

FIGURE 3.2: KNN Algorithm working visualization

Process to choose the value of k In KNN Algorithm

Determining the appropriate value for k is a pivotal aspect of the K-Nearest Neighbors (KNN) algorithm, as it defines the number of neighbors considered during the classification process. The choice of k should be tailored to the characteristics of the input data. In instances where the data exhibits more outliers or noise, opting for a higher k value is advisable. To mitigate ties in classification, it is generally recommended to select an odd value for k. Employing cross-validation methods proves beneficial in the iterative process of finding the optimal k value that aligns with the nuances of the specific dataset under consideration.[2]

Details of KNN Algorithm

The K-Nearest Neighbors (KNN) algorithm functions based on the concept of similarity, predicting the label or value of a new data point by considering the labels or values of its K nearest neighbors in the training dataset. To generate predictions, the algorithm computes the distance between each new data point in the test dataset and all the data points in the training dataset. While Euclidean distance is commonly used, other metrics like Manhattan distance or Minkowski distance may be employed depending on the problem and data. Once distances are calculated, the algorithm identifies the K nearest neighbors by sorting the distances and selecting the K data points with the shortest distances. Upon determining the K nearest neighbors, the algorithm makes predictions based on their associated labels or values. In classification tasks, the majority class among the K neighbors becomes the predicted label, while in regression tasks, the average or weighted average of the values of the K neighbors serves as the predicted value. Consider a training dataset X with n data points, each represented by a d-dimensional feature vector X_i , and corresponding labels or values Y. For a new data point x, the algorithm calculates the distance between x and each data point X_i in X using a distance metric such as Euclidean distance:

The algorithm selects K data points from X with the shortest distances to x. For classification tasks, the algorithm assigns the label y most frequent among the K nearest

neighbors to x . For regression tasks, the algorithm calculates the average or weighted average of the values y of the K nearest neighbors and assigns it as the predicted value for x .

Applications of the KNN Algorithm

Data Preprocessing: During the initial stages of dealing with a Machine Learning problem, exploratory data analysis (EDA) may reveal missing values. The KNN Imputer method, which is effective for sophisticated imputation, can be employed for data preprocessing.

Pattern Recognition: KNN algorithms exhibit high accuracy when trained using datasets like MNIST, making them suitable for pattern recognition tasks.

Recommendation Engines: The KNN algorithm assigns new query points to pre-existing groups, a crucial aspect in recommender systems where users are grouped based on preferences.

Advantages of the KNN Algorithm

1. **Easy Implementation:** The algorithm is easy to implement as it has a relatively low complexity.
2. **Adapts Easily:** KNN adapts easily to new examples, adjusting itself and contributing to future predictions.
3. **Few Hyperparameters:** The only parameters needed for KNN training are the value of k and the choice of the distance metric.

Disadvantages of the KNN Algorithm

1. **Does Not Scale:** KNN is computationally demanding and requires significant data storage, making it time-consuming and resource intensive.
2. **Curse of Dimensionality:** The algorithm struggles with classifying data accurately in high-dimensional spaces, known as the curse of dimensionality.
3. **Prone to Overfitting:** Due to the curse of dimensionality, KNN is susceptible to overfitting, prompting the use of feature selection and dimensionality reduction techniques to address this issue.

Verification Environment

Component	Description
Testbench Module	The top-level module encapsulating the testbench functionality.
DUT (Design Under Test)	The module or design being verified by the testbench.
Stimulus Generator	Creates input stimuli for the DUT, generating test scenarios.
Monitor	Observes and records the DUT's output during simulation.
Scoreboard	Compares the actual outputs with expected outputs for correctness.
Coverage Collector	Tracks functional coverage metrics to assess test completeness.
Assertions	Expresses design properties or constraints for automatic verification.
Clock and Reset Drivers	Generate clock and reset signals for synchronous designs.
Environment Configuration	Manages testbench configurations, setups, and global settings.
Virtual Interface	Represents a virtual connection between the testbench and the DUT.

FIGURE 3.3: KNN Algorithm working visualization

First in First Out (FIFO)

First-In-First-Out (FIFO) is a fundamental data structure used in computer science and digital design to manage the sequential flow of data. Its principle is straight-forward-items or elements are processed in the order they are received, akin to standing in a queue where the first person in line is the first to be served. Beyond its simplicity, FIFO finds diverse applications, ranging from memory management to communication protocols. At its core, a basic FIFO structure involves two primary operations: enqueue (adding an element to the end) and dequeue (removing an element from the front). This ensures that the sequence in which elements enter the structure is maintained throughout processing. We use water tank to store the excess fluid that is not required right now and use it later. So it kind of delinks the consumption of water by the city to the residents. In the exact same manner FIFO delinks the data that is being produced and the consumer of data by holding the excess amount of production in the FIFO.

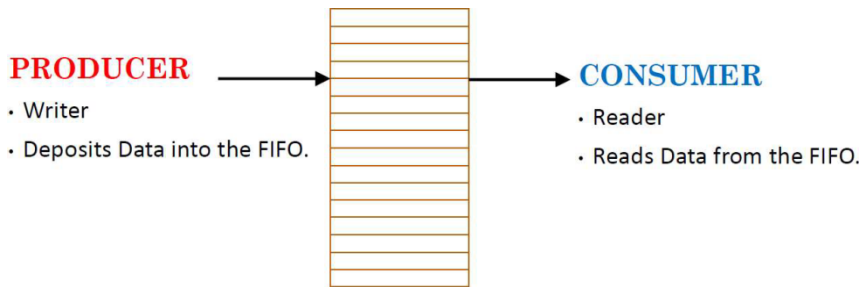


FIGURE 3.4: Pictorial representation to show how FIFO delinks the data

The main task of FIFO is to match the bit-width of the data produced at one end and the data read at another end. Let's say for example, there is one producer which can produce a 32 bit of data, but the reader of the data can only take 8 bits at a time. This will result in the overflow of data at the receiver's end. There needs to be someone in the middle who can store the data efficiently before reaching it to the receiver. This is where FIFO comes into picture to store the excess data.

The FIFO utilizes two pointers to indicate the positions for writing and reading:

Write Pointer (WP): This is the location where the data producer will store the data while writing the data.

Read Pointer (RP): This is the location where data is read by the consumer.

At initial state the FIFO will be empty and the condition for the FIFO to be empty is that both RP and WP should point to 0th location of the FIFO. As and when data is written onto FIFO, WP moves up until it reaches the top and when data is read then the RP starts moving up. When the $WP=RP$, FIFO is considered to be full. To prevent overflow situations, systems employing FIFOs often check for this condition before attempting to write new data. Proper handling mechanisms, such as blocking further writes or triggering specific actions, need to be in place when the write and read pointers indicate that the FIFO is full. Managing these pointers effectively ensures the integrity and reliability of data handling within the FIFO data structure.

Synchronous FIFO

When there are digital circuit elements working on the same clock domain, synchronous FIFO is used. It is nothing but a digital circuit which is used to send the data inside the same clock domain. When the rate of sending the data is faster as compared to the analyzing speed of the receiver end, FIFO works as a buffer between the two. The read and the write operation of the digital elements of the circuit is controlled when the clocks are synchronized in the same clock domain. This is the reason FIFO is termed as "synchronous" in this case. In the world of digital systems, think of the synchronous First-In-First-Out (FIFO) structure as the conductor or- chestrating the smooth flow of data. It's like a choreographer ensuring that every piece of data enters and exits the system in a coordinated way. What makes the synchronous FIFO unique is its connection to a clock signal—a sort of rhythmic beat that guides the movement of data. Picture a dance where the steps perfectly match the music. Similarly, in a synchronous FIFO, data transactions sync up with the ticks of the clock signal, creating a structured

approach to digital communication. The heartbeat of the synchronous FIFO is this clock, setting the pace for how data moves in and out of the system. This synchronized rhythm is especially helpful in situations where precise timing matters, like in microprocessor systems or communication interfaces. The clock-driven nature of synchronous FIFOs adds predictability, making them great for applications where the timing of data transactions is crucial.

Asynchronous FIFO

In the world of digital design and communication, asynchronous FIFO (First-In-First-Out) plays a crucial role in facilitating data transfer between different parts of a system. Think of it as a virtual queue that manages the flow of information, ensuring that data is processed in the order it was received. Unlike synchronous FIFO, which relies on a common clock signal to synchronize data transfer, asynchronous FIFO operates without a shared clock. Instead, it employs handshaking signals such as read and write pointers, along with flags like almost full and almost empty, to coordinate the data exchange. One of the significant advantages of using an asynchronous FIFO is its ability to handle data transfers between components running at different clock frequencies. This flexibility makes it a valuable component in systems where various modules may operate independently with their own timing constraints. The asynchronous nature of the FIFO introduces challenges, notably the potential for metastability issues. Metastability occurs when a signal hovers between logic levels during setup or hold time, leading to unpredictable behavior. To mitigate this risk, designers often incorporate synchronization techniques, such as double or triple flip-flop synchronizers, to ensure stable and reliable operation. Efficient data buffering is another key aspect of asynchronous FIFO design. The depth of the FIFO, determined by the number of storage elements, influences the capacity to store and transfer data. Striking a balance between sufficient buffering and minimizing latency is crucial for optimal system performance. In real-world applications, asynchronous FIFOs find use in scenarios where data communication between components involves varying delays or when clock domains differ. Their versatility and adaptability make them indispensable for enhancing the robustness and efficiency of digital systems, ensuring smooth data flow even in complex, heterogeneous environments.

Timer circuit

Timer circuits in an ASIC design is a hardware component which overlooks the timing aspect of the circuit. Commonly used timer is a watchdog timer. A watchdog timer is a hardware element which is the part of a microcontroller or microprocessor. It monitors the proper execution of a program and reset the circuit in case of any anomaly or system lock-up. Any operation of the design needs time for its execution and after which the watchdog timer will reset the operation even if that task is not completed fully. It is like a watchman who overlooks the system and in case the system is stuck, it will reset it for the smooth functioning. When the watchdog timer detects a potential issue, it triggers a system reset or takes predefined corrective actions to bring the system back to a known and stable state. This capability is particularly valuable in embedded systems, where reliability is critical, and unattended operation is common. The use of watchdog timers helps prevent scenarios where a system might hang or become unresponsive due to software bugs, code errors, or external factors. It adds a layer of fault tolerance to the system, enhancing its robustness in various applications,

including industrial control systems, automotive electronics, and other mission-critical environments.

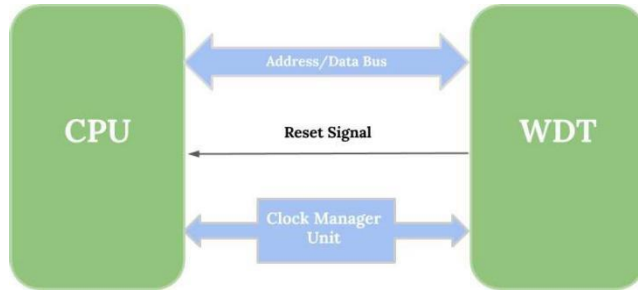


FIGURE 3.5: Watchdog Timer

Stimulation of ASIC design for the verification environment

During the design verification process a stimulus is generated in the SV/UVM testbench which drive the ASIC design. The DV engineer decides the type of stimulus which needs to be generated to drive the ASIC design. The following figure shows how the ASIC design is stimulated by the SV/UVM testbench and the generated output from the design is captured in the testbench.

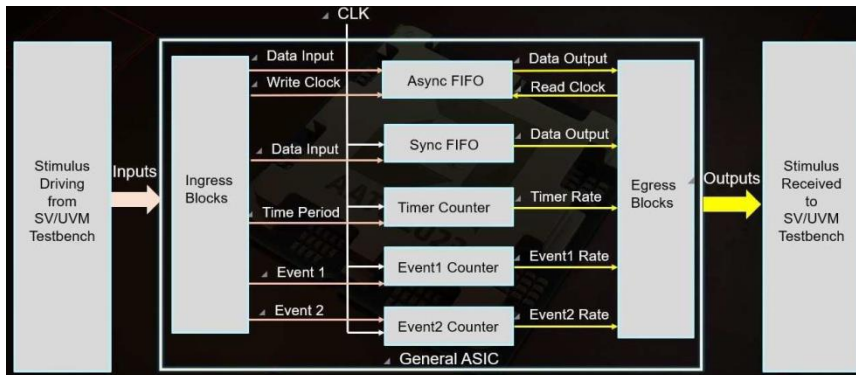


FIGURE 3.6: Stimulus to ASIC design from SV/UVM Testbench Environment

Problem Statement

Verifying FIFOs' empty and full transitions in simulations is complex due to their dynamic nature, occurring asynchronously at any clock edge. This challenge is amplified in ASIC designs with multiple FIFOs, each serving specific roles. Engineers rely on advanced simulation and verification tools to manage these complexities. They create diverse scenarios to stress test FIFOs, ensuring robustness in handling rapid transitions and unique conditions. Success in this verification is crucial for ensuring the reliability of digital systems. It demands a combination of tools, comprehensive testing, and deep understanding of timing aspects. This meticulous approach validates FIFOs' capability to handle dynamic challenges, contributing to overall system success.

In the verification process, the engineer is aware of the output data that will come from the

design of that architecture. Verification engineer writes a testbench code and matches his output with the output coming from the design under test. If both these data matches, then the design is said to be without any bugs. Now the rate at which the data is generated by the testbench environment might be slower or faster than the rate at which data is generated from the design. Now to address this problem, FIFO are used. The data is stored in the FIFO (in testbench side) is compared with the data stored in another FIFO (in the design side). Consider the following figure for pictorial representation.

The existence of stale data possesses a threat to the verification process. It is important for the design verification engineer to satisfy that the FIFOs are functioning correctly and at the end of simulation they go back to their default state. In stale data is present in the design then the FIFO will not be empty at the end of the design and DV engineer needs to eyeball which FIFO is not working correctly among

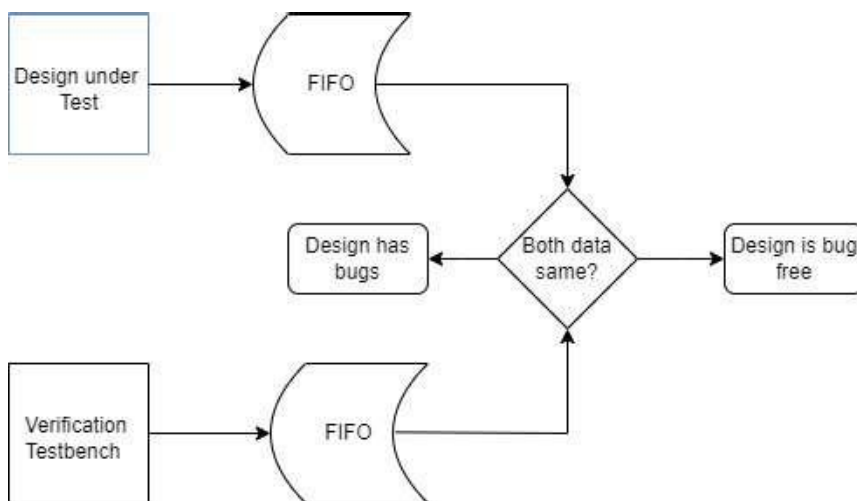


FIGURE 3.7: Verification Process Pictorial representation the thousands of FIFOs.

This process needs automation so that the manual work of the engineer is reduced, and they can focus on the more important part of the design. To reduce this manual work, we have developed a python script which can identify these FIFOs among the thousands of FIFOs within a matter of seconds by identifying the coverpoint SV file

4. Proposed Architecture

Overview

In this chapter, there are mainly two sections. In the first section the main drawback of the commonly used randomization process is discussed in form of flow chart and then an improved architecture is proposed using Supervised Machine Learning. In the second section, the automation flow to check FIFO events at the End of Simulation is discussed which reduced manual effort required.

Supervised Machine Learning for Improved Coverage Closure

In the following two sub-sections, commonly used randomization process is explained and after that the proposed architecture is introduced.

Common Randomization Process

The following figure shows how randomization is performed during functional verification of the Design Under Test (DUT) in the SV/UVM environment. The variables in the testbench may have different values which must be covered to ensure the functionality of the design. For that reason randomization becomes essential, which will cover all the possible values of that variable in uniform distribution during regression testing.

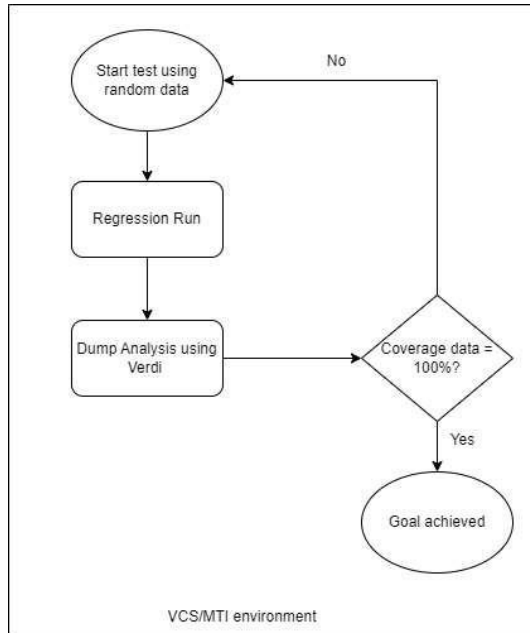


FIGURE 4.1: Regular Randomization SV testbench environment

Take for example, a 4:1 Mux which has 4 input lines and 2 select lines. Now for each combination the values of select line, that input line will be selected. Now each possible value of the select line must be covered to ensure the functionality of the design. The table below show the possible value of the select line and what input line will be selected for the same.

TABLE 4.1: Truth Table of 4:1 Mux

Sel 1	Sel 0	Output
0	0	Input line 0
0	1	Input line 1
1	0	Input line 2
1	1	Input line 3

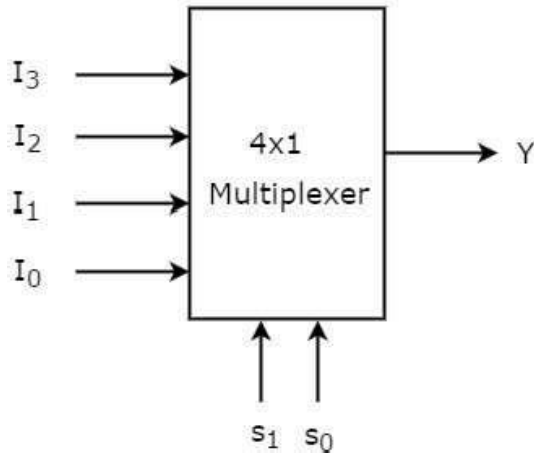


FIGURE 4.2: 4:1 mux with 2 select lines

So, in the common approach the keyword "rand" is used in front of the select variable and that variable will be randomized. Now any simulator let's say its VCS/MTI/- Vivado uses the same approach throughout. It is the inbuilt functionality of that tool.

To achieve 100% coverage, it becomes mandatory to run multiple seed regression of some random test. In each individual test some valid possible values of the variables are generated and bins of each value is hit for that variable. If all the bins are hits, 100% coverage is achieved. Ideally to cover for example 250 valid values, 250 test must be run and coverage of 100% can be achieved but it is not the real scenario.

Drawback of this approach is that less Turnaround Time of coverage closure cannot be ensured as in the process of randomization there is repetition of values. So, running 250 test does not mean that 250 unique bins will be hit. Some of the bins are hit multiple times while some are not hit at all. This results in waste of time and as well as resources. For addressing this issue, an improved architecture is proposed.

Proposed ML based randomization approach

The following figure shows that the ML model as a wrapper to the VCS/MTI environment. The random data which was generated by the EDA Tool is now taken from the ML algorithm and the random number is generated on the during simulation. The ML model will only return the combination of the random values of the variables when they are unique and thus eliminating the possibility of hitting the bin for same value multiple times during the multiple seed regression.

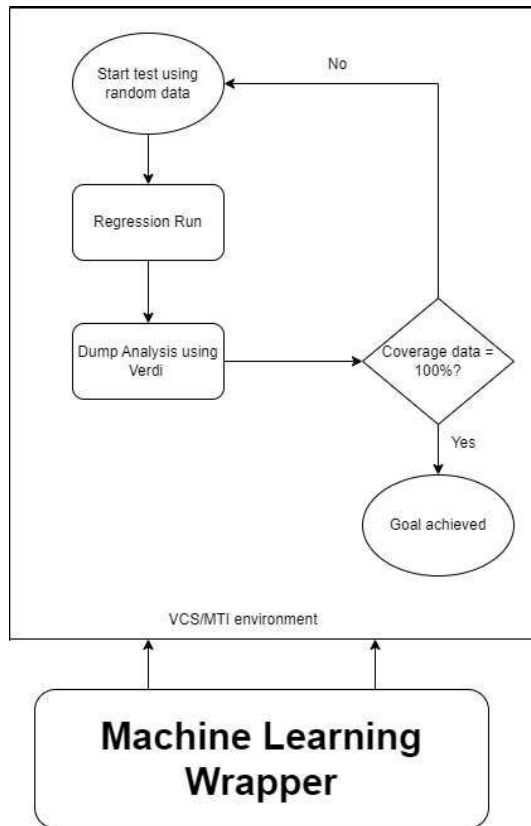


FIGURE 4.3: Proposed approach using ML

The ML algorithm is developed using System Verilog and the random data will now be generated by the code instead of the simulator. There are independent as well as dependent variables in the testbench. During a random test, the simulator produces random values for those independent variables and the dependent variable gets updated according to the values of the independent variables. So, a wrapper is added to the simulator and relieving it from its duty of generating random variables.

The advantage of this method is that only unique sequence of values are returned to the tb and a bin is hit only once and not multiple times. Thus, by running 250 different seeds of regression almost 95% of the valid bins were hit and thus improving the turnaround time of the coverage closure significantly. Also, all the random values are generated at "O ns" time so no pre or post processing of the test data is not required.

Automated approach to check FIFO events at EoS

In the following two subsections the method to check for EoS of FIFO events in form of writing assertion and checking for stale data is defined which is a manual process that takes time to debug. Eyeballing FIFO which contains stale data is done by writing assertion checks for during simulation and end of simulation condition for each FIFO. The next subsection introduces the automates approach for FIFO assertion checks at EoS and also defines where to add the automated generated codes.

Current method for checking FIFO events

The following flow diagram describes how the events for FIFOs are checked for debug at the end of simulation for the ASIC design. Among the thousands of FIFOs in the design, there may be some FIFO which has not be set to the default state at EOS. The process for checking it is shown below.

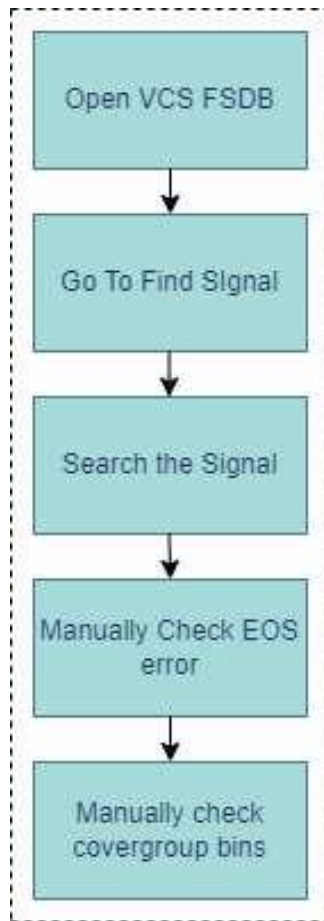


FIGURE 4.4: Common Approach to check for FIFO events

Proposed approach to check FIFO events

The below mentioned flow shows how the automation is achieved.

In the second box different System Verilog files are created by using the help of python script and automating the entire flow. The main objective is to check whether the FIFOs have become empty at the end of simulation and check each of the transition of FIFO during simulation also.

In the first step, a VCS FSDB is opened to identify all the signals present in the design. The process involves searching for the FIFO full and empty signals separately

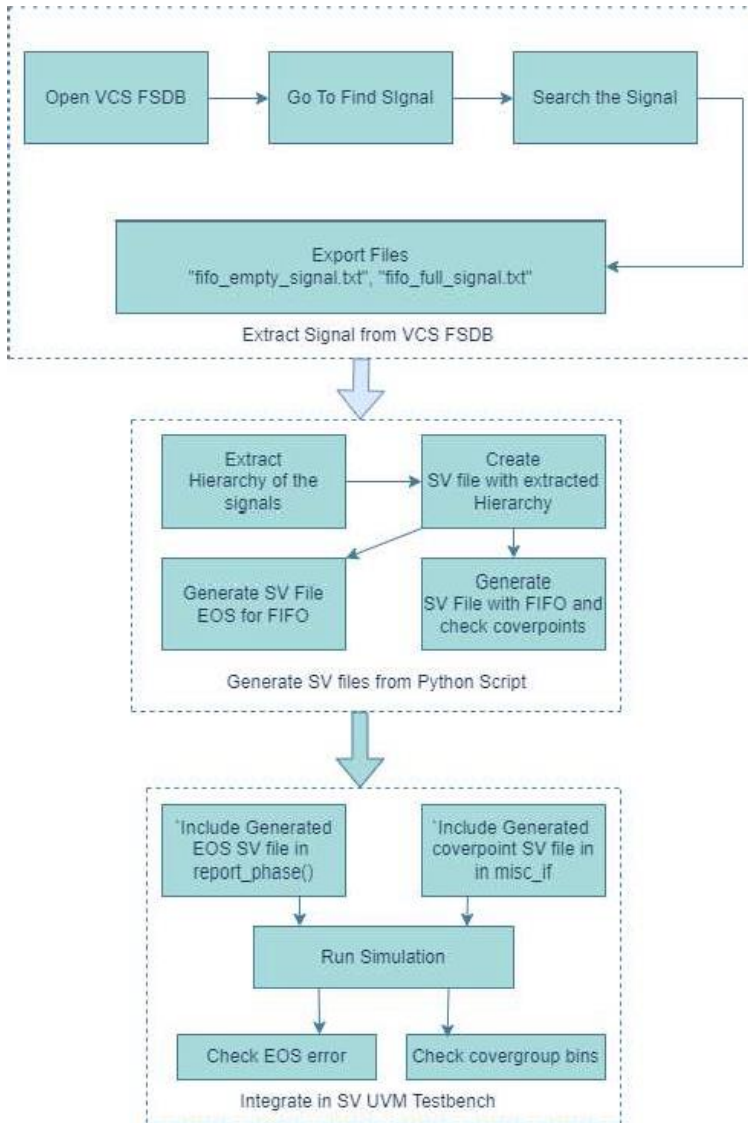


FIGURE 4.-5: Proposed approach for automation

and creating two distinct text files for each. These text files serve as input to a Python script. This marks the completion of the signal extraction from the VCS FSDB. The text files are named "fifo_empty_signal.txt" and "fifo_full_signal.txt" for storing the empty FIFOs and full FIFOs, respectively.

This shows the hierarchy of the pc_sysmem_wr_req_fifo_almost_full[7:0] signal and from which classes it was derived from. We then generate the SV files from the extracted hierarchy of the signal. The SV files are as follows:

fifo_check_input.sv - All the FIFO as input signal.

fifo_check_coverpoint.sv - This file contains the coverage script for each of the FIFO.

Coverage script is to check the transition coverage of fifo full and empty from 0->0, 0->1, 1->0 and 1->1.

EOS fifo check.sv - This file contains the end of simulation check for each fifo and in case the condition is something it is not supposed to be, it throws an error message, and we can check which fifo is as per expectation.

Finally, the EOS _ fifo _ check.sv file is added to the report phase, which will report any errors arising from the non-ideal condition of the FIFO. Additionally, the other two files, fifo _ check _ input.sv and fifo _ check _ coverpoint.sv, are included in the misc_if.sv file, which checks the coverage of each FIFO.

This current approach reduces the time taken to debug FIFO-related problems. Ideally, the empty signal should be equal to 1 and the full signal should be equal to 0 at the end of the simulation. If this condition is not met, it indicates an error in that FIFO. In the existing approach, a DV engineer needs to review each FIFO and check for errors manually. With the new approach, everything is automated, significantly reducing the need for manual work.

5. Experimental Setup

Overview

In this chapter the experimental setup to deploy the Machine Learning algorithm is discussed along with the automation of EoS assertion checks for the FIFO events.

Design Environment

The experimental setup is constructed around a standard digital design and verification environment using System Verilog. The primary objective is to enhance the coverage closure efficiency using supervised machine learning techniques. The tools and resources utilized in the experiment are as follows:

Hardware Description Language (HDL): SystemVerilog is employed to model the design and verification components.

Simulation Tool: Synopsys VCS is used for simulation, providing comprehensive support for SystemVerilog.

Waveform Database: FSDB (Fast Signal Database) files generated by VCS store the signal activities during simulation.

Debug Tool: Synopsys Verdi used for debugging purpose and to check improvement in Functional Coverage.

Text Editor: The editor used to write code is GVIM editor and Visual Studio Code for writing python code.

Signal Extraction Process

Opening FSDB File: The VCS-generated FSDB file is opened to access all signals present in the design.

Signal Identification:

FIFO Signals: The script searches for FIFO full and empty signals separately.

File Generation: Two text files, named "fifo_empty_signal.txt" and "fifo_full_signal.txt", are created to store the empty and full FIFO signals respectively. These files will serve as inputs to a Python script for further processing.

6. Result and Discussion

Overview

In this chapter a comparison report is given to justify the use of Machine Learning based search algorithm which is used to search for a combination of user defined numbers inside an already existing data set (training data). The timing improvement is shown in form of table.

Motivation to use the ML approach

There are some advanced search methods which can search a particular sequence of values in a given set of data. The common search and advanced search methods are as follows:

1. Binary search
2. Interpolation search
3. Saddleback Search
4. Linear search

We eliminated binary, interpolation and saddleback search on the premises that the given set of data must be sorted in order for this search methods to work. So the comparison was made for linear and ML approach. The number of training data was set constant to 4000 and the number of variables were varied from 100,200 to finally 400 variables.

The following table highlights the timing performance of the following method:

	Linear Approach (Time in ms)	ML Approach (Time in ms)
Training Data = 4000*100 Test Data = 1*100	285.34	5.6
Training Data = 4000*200 Test Data = 1*200	1119.58	15.2
Training Data = 4000*400 Test Data = 1*400	4613.94	24.2

FIGURE 6.1: Timing improvement in ML approach

In this method ML approach showed almost 50 times improvement over the linear approach

for 100 variables and the improvement were higher for the increase in the number of variables. VCS and MTI coverage result

Test was done to find out the coverage result of 6 independent variables of 3-bit size which will create 729 unique combinations. When all the 729 combinations are hit then the coverage report will show 100% report. In the following figure we ran tests on VCS and MTI simulator to check the coverage report and repetition of values.

We ran 700 seed of regression, and the result is as follows:

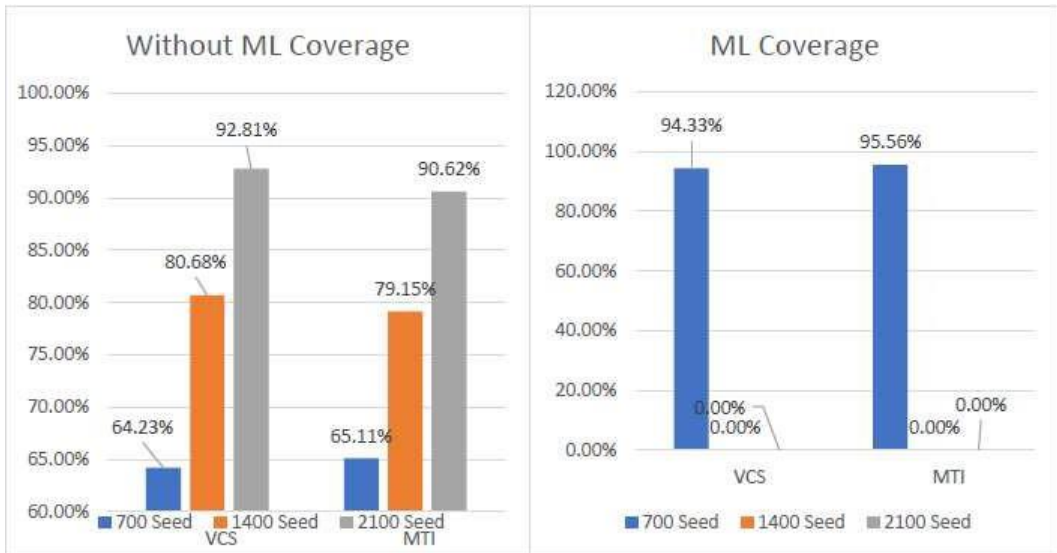


FIGURE 6.2: VCS and MTI Data repetition with and without ML approach

The main reason for decreased coverage result was because of repetition of the same sequence of values. For 700 seed both VCS and MTI both showed close to 40% repetition while using the ML approach showed maximum of 2% for VCS and 1.2% for MTI repetition of values. This directly correlated to the improved coverage report using the Machine learning approach.

Coverage Improvement in the existing testbench setup

We conducted 20 random tests by integrating the ML model to the existing setup of the testbench. The result showed almost 1.5 times improved performance over the current setup.



FIGURE 6.3: Functional Coverage Improvement result

Timing Optimization by Automation

The current approach of automation was added to the current setup, and we noted the timing improvement for that. Following figure shows the timing improvement of our approach to manual approach. We performed this with different number of

TABLE 6.1: The comparison of time taken for manual vs automated approach

Number of FIFO	Manual approach (minutes)	Automated approach (minutes)
10	10	1
40	28	1.5
100	72	2.5

FIFO like 10,40,200 and 400. The improvement was little for a smaller number of FIFOs but as the number of FIFO in the design increased, the change in the time was significant.

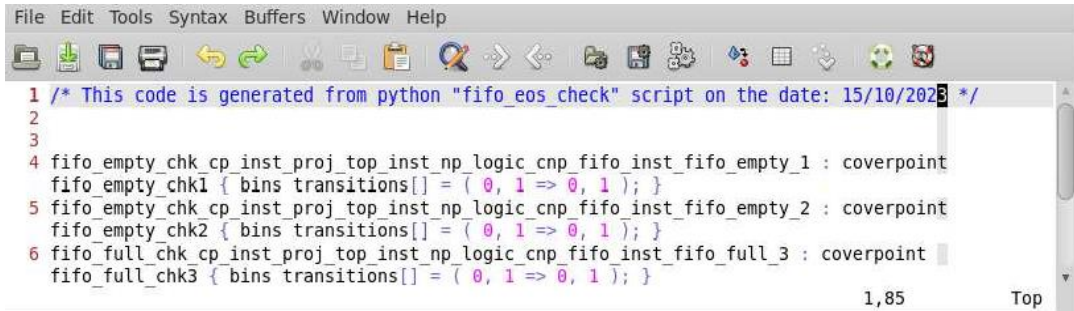
Although the timing improvement for a smaller number of FIFO was quite significant but the improvement seen in case of more number of FIFO is seen to be increasing exponentially. The automation done using python script made the process easy and time efficient. The need for reviewing each fifo to check for error is reduced completely and is more efficient as it reduced the human involvement in the process.

Integration with current testbench setup

In the current ongoing project, there are more than 150 fifo and the coverage for each of the fifo must be checked and all the at the end of simulation fifo empty and full condition must also be ensured. As mentioned earlier only process to do is manually and coverage for each fifo must also be written separately after identifying the fifos of the design. Now from our approach we automated the entire flow and got the following files as output from the python script:

1. First SV code is for fifo end of simulation check for fifo events from full to empty or empty to full. This file is named as "fifo_check_input.sv."

2. Second SV code is for the fifo signal. This file is named as "fifo_check_input.sv."

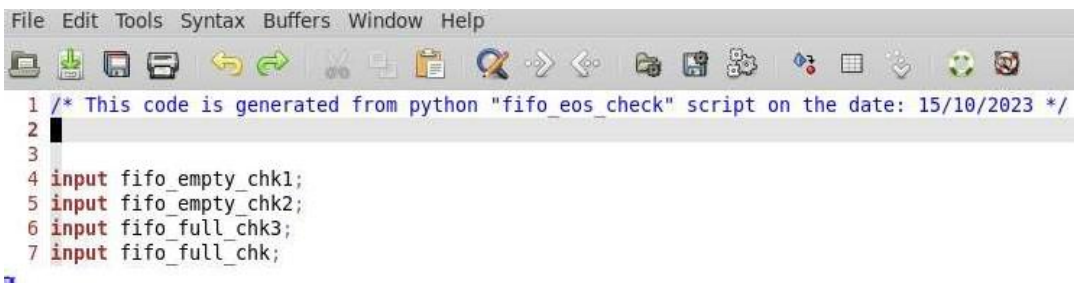


```

1 /* This code is generated from python "fifo_eos_check" script on the date: 15/10/2023 */
2
3
4 fifo_empty_chk_cp_inst_proj_top_inst_np_logic_cnp_fifo_inst_fifo_empty_1 : coverpoint
  fifo_empty_chk1 { bins transitions[] = ( 0, 1 => 0, 1 ); }
5 fifo_empty_chk_cp_inst_proj_top_inst_np_logic_cnp_fifo_inst_fifo_empty_2 : coverpoint
  fifo_empty_chk2 { bins transitions[] = ( 0, 1 => 0, 1 ); }
6 fifo_full_chk_cp_inst_proj_top_inst_np_logic_cnp_fifo_inst_fifo_full_3 : coverpoint
  fifo_full_chk3 { bins transitions[] = ( 0, 1 => 0, 1 ); }

```

FIGURE 6.4: FIFO EOS check SY code



```

1 /* This code is generated from python "fifo_eos_check" script on the date: 15/10/2023 */
2
3
4 input fifo_empty_chk1;
5 input fifo_empty_chk2;
6 input fifo_full_chk3;
7 input fifo_full_chk;

```

FIGURE 6.5: FIFO signals of the code

1. These above two codes are then added to the misc if file to check for the coverage of the fifo. We check whether the fifo has transitioned through all the possible states in the end of simulation. The files that are added are "fifo_check_input.sv" and "fifo_check_input.sv." This is added by using 'include' and then adding the respective file names. This is given below.

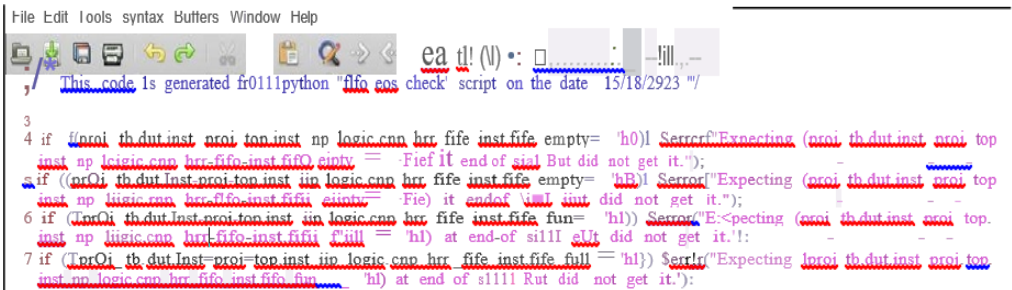
2. Another SV file generated from the python script is to check the state of fifo and if there is any error in it, then that error will be printed. This is added to the report phase.

This helped in catching bugs in the two RTL bug in the Project Block Level. This has reduced the manual work which was previously required for catching these bugs.

```

10
11   'include "counter check logic.sv"
12   'include "fifo check logic.sv"
13
14   // =====
15   // Clocking block for testbench
16   // =====
17   clocking cb@(posedge elk);
18       input elk;
19       'include "counter check input.sv"
20       'include "fifo check injut.sv"
21   endclocking
22   // =====S
23   // Coverage Group in interface
24   // =====
25   covergroup fifo empty full cov @(posedge elk);
26       'include "fifo check coverpoint.sv"
27   endgroup: fifo empty full cov
28
29   covergroup counter cov @(posedge elk);
30       'include "counter check coverpoint.sv"
31   endgroup : counter cov
32   // =====
33   // Instance of covergroup
34   // =====
35   fifo empty full cov fifo empty full cov inst= new();
36   counter cov          counter cov inst = new();
37
38 endinterface
39 endif

```



7. Conclusion

The Python script fully automated the process of checking FIFO events and transition coverage at the end of the simulation, significantly reducing manual effort and increasing coverage checking performance by over 9x. This automation enabled earlier bug detection and improved coverage checking, especially as the number of variables increased. Integration with the testbench enhanced the speed of writing coverage and checking FIFO events, efficiently handling over 150 FIFOs and identifying two RTL bugs, reducing the coverage writing time to minutes.

The report evaluated the performance of a machine learning (ML) model under various dependency conditions between constraint random variables. The ML approach demonstrated a 50x timing improvement over the linear approach, with an ML model test on 8 independent variables of 3-bit size showing a 33% improvement in turnaround time (TAT) for coverage closure. The ML approach reduced value repetition from 45% to 2%, and its integration into the testbench setup and 20 random tests resulted in an improved coverage report.

8. Future Scope

There are many areas in the existing testbench setup of SV/UVM that require a lot of manual effort and which are slow in finding bugs because of increased computation time. Using Machine Learning in general in Design Verification is a new avenue that hasn't been explored much which in turn has lot of potential reduce the overall timing parameter and increase the overall efficiency. The discussed Machine learning model along with the automation code can be part of a EDA Tool like Vivado which will reduce the effort to add the setup in the existing testbench and can be run as a part of the tool.

References

1. L. Entrena et al., "Formal Verification of Fault-Tolerant Hardware Designs, 11 in IEEE Access, vol. 11, pp. 116127-116140, March 2021
2. D. Yan, Y. Wang, J. Wang, H. Wang and Z. Li, "K-Nearest Neighbor Search by Random Projection Forests, 11 in IEEE Transactions on Big Data, vol. 7, no. 1, pp. 147-157, May 2022
3. S. Zhang, X. Li, M. Zong, X. Zhu, and R. Wang, "Efficient kNN classification with different numbers of nearest neighbors," IEEE Trans. Neural Netw. Learn. Syst., vol. 29, no. 5, pp. 1774-1785, May 2018.
4. F. Groh, L. Ruppert, P. Wieschollek and H.P. A. Lensch, "GGNN: Graph-Based GPU Nearest Neighbor Search, 11 in IEEE Transactions on Big Data, vol. 9, no. 1, pp. 267-279, 1 Feb. 2023
5. Synopsys VCS @ "Reducing Simulation Regression Turnaround Time with Dynamic Performance Optimization" White Paper @2023 Synopsys.
6. A. Puhm and P. Rossler, "Considerations on teaching digital ASIC design," 2014 IEEE/ ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA), Senigallia, Italy, 2014, pp. 1-6.
7. W. -P. Lee and C. -Y. Wang, "Reusable and flexible verification methodology from architecture to RTL design," VLSI Design, Automation and Test(VLSI-DAT), Hsinchu, Taiwan, 2015, pp. 1-4.
8. P. A. Beerel and M. Pedram, "Opportunities for Machine Learning in Electronic Design Automation, 11 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 2018, pp. 1-5.
9. E. Fallon, "Machine Learning in EDA: Opportunities and Challenges," 2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD), Reykjavik, Iceland, 2020, pp. 103-103.
10. System Verilog Language Reference Manual
11. A. Kulkarni, A. Singh, S. A. Waje, S. S. Kashide and S. B. Choi, "TestQuBE: A Testbench Enhancement Methodology for Universal Serial Interfaces in Complex SoCs," 2021 IEEE 34th International System-on-Chip Conference (SOCC), Las Vegas, NV, USA, 2021, pp. 106-111.
12. Y. -K. Choi, Y. Chi, J. Wang and J. Cong, "FLASH: Fast, Parallel, and Accurate Simulator for HLS," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 39, no. 12, pp. 4828-4841.
13. Testbench Automation and Constraint Tutorial. [Online]. Available: Testbench Automation and Constraints Tutorial (doulos.com)
14. Synopsys. (2019) VCS Functional Verification Solution. [Online]. <https://www.synopsys.com/verification/simulation/vcs.html>