# Optimizing Constraint Selection in a Design Verification Environment for Efficient Coverage Closure

Vanessa Cooper
*Nova Southeastern University*, vcook77@gmail.com

# Optimizing Constraint Selection in a Design Verification Environment for Efficient Coverage Closure

by

Vanessa R. Cooper

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

College of Computing and Engineering
Nova Southeastern University
2023

**We hereby certify that this dissertation, submitted by Vanessa Cooper conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.**

Sumitra Mukherjee

**Sumitra Mukherjee, Ph.D.**
**Chairperson of Dissertation Committee**

**4/ 5 /23**
**Date**

[illegible]

**Michael Laszlo, Ph.D.**
**Dissertation Committee Member**

**4/ 5 /23**
**Date**

[illegible]

**Francisco J. Mitropoulos, Ph.D.**
**Dissertation Committee Member**

**4/ 5 /23**
**Date**

**Approved:**

Meline Kevorkian

**Meline Kevorkian, Ed.D.**
**Dean, College of Computing and Engineering**

**4/ 5 /23**
**Date**

**College of Computing and Engineering**
**Nova Southeastern University**

**2023**

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

# Optimizing Constraint Selection in a Design Verification Environment for Efficient Coverage Closure

by
Vanessa R. Cooper
April 2023

Hardware Design Verification, commonly called verification, is the process of functionally verifying a design that was written in a Register Transfer Language (RTL) based on the specification. The most common methodology in practice today uses the Universal Verification Methodology (UVM). In addition to being a methodology, UVM is also a collection of object-oriented base classes written in SystemVerilog. A UVM testbench is created that serves as a harness for the Design Under Test (DUT). Stimulus, with constrained random inputs, is written that exercises the design. Constrained Random Verification (CRV) is used to find functional errors in the design and to meet coverage goals.

One form of coverage is functional coverage. Functional coverage is defined and developed by the verification engineer to verify that certain scenarios are covered during simulation. Coverage is created as a collection of coverpoints, and each coverpoint has one or more bins to indicate what is or is not hit. As a simple example, one coverpoint could be the write indicator for a First-In First-Out (FIFO) queue. This coverpoint would have two bins: one if the write indicator was high and another if it were low. For a large System-on-Chip (SOC) design, the number of bins to be covered could number in the tens of thousands. Reaching 100% coverage on such a design would require a large amount of compute space and potentially thousands of simulations running daily. The time it takes to analyze the coverage results and develop new stimulus to cover missing cases can take weeks to months, depending on the complexity of the design. To improve the efficiency and time it takes to reach coverage closure, this dissertation study evaluates the use of machine learning techniques. More specifically, this study uses a combination of Supervised Learning, Reinforcement Learning, and Bayesian Optimization to select constraints to reach coverage closure more efficiently.

Using two different RTL models, the results of this study show that using Machine Learning models reduces time to coverage closure. Using a combination of Bayesian Optimization with Reinforcement Learning, constraints were optimized so that the number of simulations required to reach 100% coverage was much less than using constrained random constraints. This research highlights the effectiveness of using Machine Learning in the Hardware Design Verification flow.

# Acknowledgements

I am profoundly thankful to my advisor, Dr. Sumitra Mukherjee. My first class at NSU was his Artificial Intelligence class. He advised all of his students to keep their topics focused and reminded us that you can do all the research you want after you finish your disseration. That sage advice as carried me through this process. I would also like to express my appreciation to the rest of the dissertation committee, Dr. Michael Laszlo and Dr. Francisco Mitropoulos, for their input and feedback. Special thanks to my Academic Advisor, Laura Macias, who was always so helpful and gracious when I had questions.

I was able to take on this challenge thanks to the support of my company, Verilab Inc. Special thanks go to Jason Sprott, CTO, for allowing me to bounce ideas off of him and his great feedback. Also, a heartfelt thank you to Dr. Tommy Kelly, CEO. Tommy's guidance, friendship, and mentorship has allowed me to grow as a person, engineer, and researcher.

I would be remiss if I did not thank my biggest cheerleaders. Thank you to my loving husband, Doug. I am truly blessed beyond measure for your constant support and encouragement. Thank you to my son, Lucas. One day, when you are working on your dissertation, I will cheer you on as enthusiastically as you have done for me.

Finally, as a woman of faith, I thank God for this opportunity. Not everyone has the chance to pursue and accomplish a life-long goal. I will be forever grateful.

## Table of Contents

## List of Tables

## List of Figures

# List of Equations

# Chapter 1

# Introduction

## Problem Statement

As Application Specific Integrated Circuit designs continue to grow in complexity, the hardware design verification effort continues to be a bottleneck (Vanaraj et al., 2020). The study by H. D. Foster (2015) shows that the majority of the effort spent in the design flow is in the verification process. This dissertation investigates how machine learning techniques can be used to address this bottleneck problem so that verification goals can be reached faster and more efficiently.

Design Verification is the discipline of functionally verifying the hardware design of an Application Specific Integrated Circuit (ASIC) or Intellectual Property (IP). The design is typically written in a Register Transfer Language (RTL) such as Verilog or VHDL. A verification environment or testbench is constructed to stimulate the RTL and verify the results according to the specification. The testbench and stimulus can be written in SystemVerilog and many companies leverage the Universal Verification Methodology (UVM) for more complex environments. A UVM based environment consists of several components used to stimulate and check the Design Under Test (DUT)(Cooper, 2013). Figure 1 illustrates a simple UVM testbench and its components. The subcomponent of interest is the Sequences. At a high level, the sequences contain data items that aggregate to form a packet or transaction that is sent to the DUT. The data items or knobs are randomized with constraints to meet the verification goals of stimulating various use cases in the design. This method of verification is known as constrained random verification (CRV). As

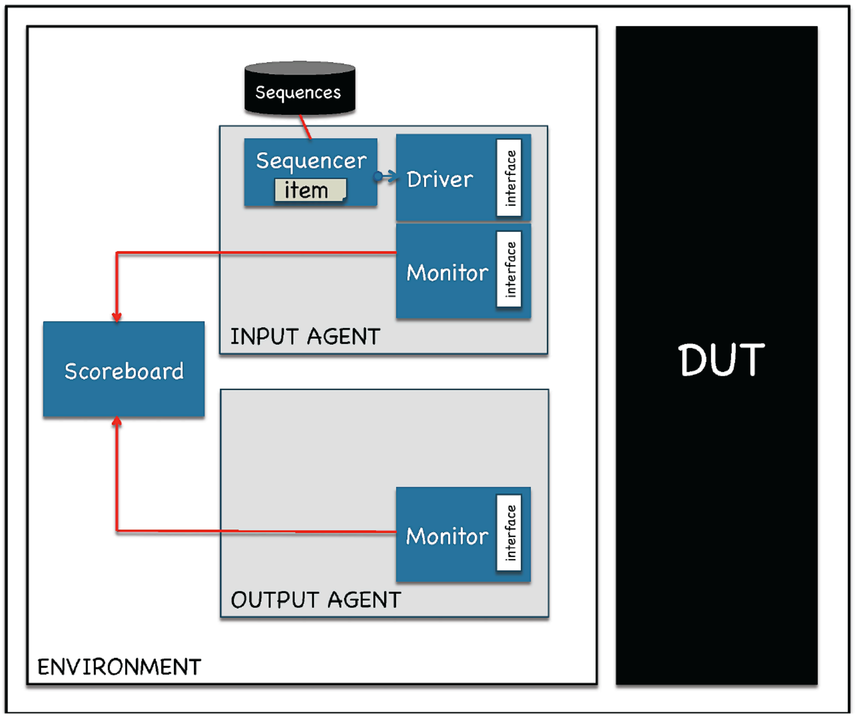


Figure 1. UVM Testbench (Cooper, 2013)

designs have become increasingly complex, the complexity of constrained random verification has drastically increased as well (Roy et al., 2021; Vanaraj et al., 2020).

One of the verification goals that must be met is called Coverage Closure. There are two types of coverage: code and functional. Code coverage is observing that you hit a certain line of code, toggled a bit, or exercised all conditional branches. However, functional coverage is a better measure of the robustness of the stimulus. Functional coverage indicates if a certain use case or scenario has occurred. For example, did the stimulus hit the case where a First-In First-Out (FIFO) queue is full or empty? Did back-to-back transactions on a protocol bus happen? With a large System On Chip (SOC) design, the coverage space could take weeks or months to close using standard constrained random verification. The challenge becomes how to optimize the constraints in such a way as to significantly reduce the time to closure.

## Dissertation Goal

The goal of this study is to combine Supervised Learning, Reinforcement Learning, and Bayesian Optimization into a hardware design verification flow to increase the speed and efficiency of coverage closure. Hughes et al. (2019) proposed a method of optimizing constraint selection using a combination of Supervised and Reinforcement Learning. In comparison, Roy et al. (2021) proposed using Bayesian Optimization to optimize constraints. Both studies show improvement in coverage closure compared to constrained random verification alone. The hypothesis is that by combining these two approaches, the time to coverage closure will be significantly reduced. Success would be defined as meeting a coverage closure goal in fewer iterations of the simulation.

The first optimization method proposed by Hughes et al. (2019) uses a combination of Supervised Learning in the form of a Deep Neural Network (DNN) and Reinforcement Learning with a Deep-Q Network (DQN). Initially, the constrained random values, or random knobs as indicated in Figure 2, are fed into the simulation to produce an output. That output is then used to train the neural network. Once the network is trained, it can be used to select new inputs to maximize the output that most effectively achieves coverage closure.

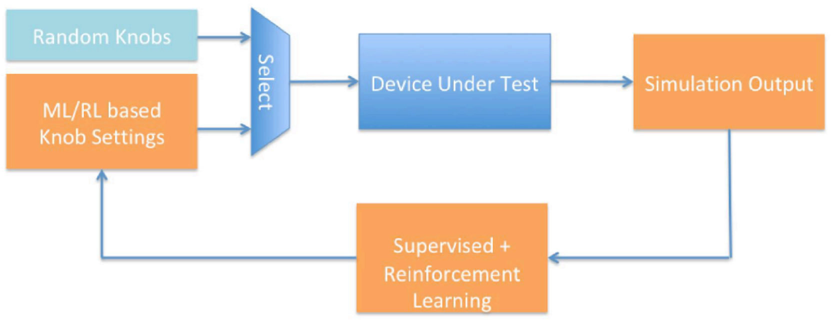

Random Knobs
ML/RL based Knob Settings
Select
Device Under Test
Simulation Output
Supervised + Reinforcement Learning

Figure 2. Simulation Flow with Supervised and Reinforcement Learning (Hughes et al., 2019)

This setup also includes a Reinforcement Learning component. Reinforcement

Learning (RL) involves an agent trying to learn the best actions in a heuristic manner to achieve a reward. In a small state space, the RL algorithm of Q-Learning would be sufficient (Hughes et al., 2019). In that scenario, the rewards could be stored in a lookup table. However, given the complexity of the environment, a lookup table is no longer sufficient and requires a neural network called a Deep Q-Network (DQN). In both cases, the networks act as a function approximator to achieve the desired outcome (Goodfellow et al., 2016). The best set of input knobs from both setups is then used in simulation.

In comparison, instead of using deep neural networks, Roy et al. (2021) proposes using Bayesian Optimization. Bayesian Optimization is a machine learning algorithm that finds the global optimum $x$ of the objective function $f(x) : \mathbb{X} \rightarrow \mathbb{R}$ (Li & Kanoulas, 2018). For the verification goal of coverage closure, the coverage hit counts would be the function $f(x)$ and the randomized constraints would be $x$ (Roy et al., 2021). Bayesian Optimization (BayesOpt) is an iterative process. Roy et al. (2021) outlines the iteration in the following steps:

1. Evaluate the objective function $f(x)$ for an initial random set of $x$ values $[x_0 : x_{n-1}]$
2. Train a surrogate model with data $[(x_0, f(x_0)) : (x_{n-1}, f(x_{n-1}))]$ collected from evaluation
3. Determine the next set of $x$ to evaluate using an acquisition function on the surrogate model
4. Evaluate the objective function for the new $x$ values: $f(x)$ for $x$ in $[x_n : x_{2n-1}]$
5. Start the next iteration at Step 2 with the additional data collected from evaluation

An acquisition function is one that finds a trade-off between exploration and exploitation (Li & Kanoulas, 2018). Exploitation is sampling where the surrogate model predicts a high value of $f(x)$ and exploration is sampling where the prediction uncertainty is high (Roy et al., 2021). The surrogate model is used by the acquisition function to find the next samples for the iteration. According to Li and Kanoulas (2018), the Gaussian Process (GP) is the most widely used surrogate model in BayesOpt. In addition to GP, Roy et al. (2021) also used Extra-trees (ET) and Gradient-boosted regression trees (GBRT). With regards to coverage, results of the experiment showed a 2x-6x coverage count increase with the optimized constraints for 20% of the coverpoints. The methodology used by Hughes et al. (2019) also showed a significant improvement in certain coverage goals using a ML approach instead of normal constrained random verification.

## Relevance and Significance

Using Machine Learning in a hardware design verification flow is an area that has seen a resurgence given the developments in technology (Gogri et al., 2020). Gogri et al. (2020) proposed a flow that used a machine learning model for each planned cover point. F. Wang et al. (2018) proposed using the relationship between tests and assertions in the verification flow in conjunction with training an Artificial Neural Network (ANN). To train the model, the outputs were labeled positively if a test triggered a certain assertion (F. Wang et al., 2018). A verification flow created by C. Wang et al. (2022) used supervised and reinforcement learning, but they added the novel approach of adding a Transformer into the flow. A Transformer is a Natural Language Processing (NLP) technique used in deep learning models that applies different weights to parts of the input data (C. Wang et al., 2022). As opposed to using Bayesian optimization to choose constraints, the Transformer creates a subset of constraints from all the constraints. Work done by Chauhan (2022) also uses natural language processing; however in this case, it is used to translate text from the specification into assertions. Chauhan and Ahmad (2021) has also researched using machine learning for regression automation. Like this proposed study, prior research has focused on various segments of hardware design verification. Developing efficient methods using artificial intelligence to optimize portions of the flow will eventually lead to an end to end solution.

## Barriers and Issues

Typically, when simulations are run for hardware design verification, it is done on a compute farm. Utilizing a compute farm allows simulations to be run in parallel across the various number of servers in the farm. For the experimentation carried out in this research study, access to a compute farm is not available. All simulations done were run on one server. Although the time to completion of all necessary simulations was longer, the results were not affected.

## List of Acronyms

| | |
|---|---|
| **ANN** | Artificial Neural Network |
| **API** | Application Programming Interface |
| **ASIC** | Application Specific Integrated Circuit |
| **CRV** | Constrained Random Verification |
| **DNN** | Deep Neural Network |
| **DQN** | Deep-Q Network |
| **DUT** | Design Under Test |
| **EDA** | Engineering Design Automation |
| **FIFO** | First In First Out |
| **IP** | Intellectual Property |
| **LSTM** | Long Short-Term Memory |
| **ML** | Machine Learning |
| **MLP** | Multilayer Perceptron |
| **NLP** | Natural Language Processing |
| **OS** | Operating System |
| **RNN** | Recurrent Neural Network |
| **RL** | Reinforcement Learning |
| **RTL** | Register Transfer Language |
| **SOC** | System On Chip |
| **UVM** | Universal Verification Methodology |
| **VPLAN** | Verification Plan |

Table 1. Acronyms

## Summary

Hardware design verification is an integral component in ASIC design. As chips become more and more complex, the complexity of the verification effort increases as well. One element of the verification process is the closure of functional coverage. This study utilizes Supervised Learning, Reinforcement Learning, and Bayesian Optimization in an effort to close coverage more efficiently.

# Chapter 2

# Review Of the Literature

## Overview

The purpose of hardware design verification is to verify that a design is correct based upon its specification (Sethulekshmi et al., 2016). As this study explores using machine learning techniques to improve an element of the verification flow, the review covers current research in Hardware Design Verification as well as Bayesian Optimization, Supervised Learning, and Reinforcement Learning.

## Hardware Design Verification

Industry stalwart, H. D. Foster (2015), led a study to determine the cost of verification in the overall design process. The study showed that there has been a steady increase in projects where the cost of verification was 80% of the project. Given this high cost, verification has become the bottleneck in the design flow (H. D. Foster, 2015). One contributor to this cost is the life cycle of a product. Whereas in years past, the life cycle of an ASIC from inception to production would take years, that cycle has been reduced to months (Chen et al., 2017). As such, despite being more complex, the stages of pre-silicon verification must fit within the expedited life cycle. As shown in Figure 3, the three major components of pre-silicon verification are verification planning, architecture design, and plan execution.

Planning
Architectural Design
Execution

Figure 3. Pre-Silicon Verification Stages

Verification planning starts early and consists of understanding the IP or blocks of IP under test (Chen et al., 2017). The verification plan contains the features that will be tested and how they will be tested, checked, and covered. The next phase is the architectural design of the verification components. This includes the design of agents, checkers, monitors, and scoreboards outlined in the verification plan (Chen et al., 2017). These components include those shown in Figure 1. The most intensive and costly stage is the execution of the plan. Components designed during the previous stage have to be coded, including the sequences for the stimulus (Chen et al., 2017). Any failing tests need to be debugged, and the RTL fixed if there is a design flaw (bug). Functional coverage has to be implemented and closed 100% (Vanaraj et al., 2020). Regressions (test suites) need to be passing cleanly for a pre-determined amount of time to help ensure there are no bug escapes.

One strategy to help reduce the verification bottleneck has been the adoption of the Universal Verification Methodology (H. D. Foster, 2015). The use of UVM helps with reuse across projects. It also helps reduce the learning curve of new employees since it has been so widely adopted. However, the adoption of SystemVerilog and UVM has reached a saturation point while designs continue to become more complex (H. D. Foster, 2015). Researchers and industry verification engineers have continued to deploy other strategies to optimize the verification effort. Vanaraj et al. (2020) proposed a framework to calculate the valid input stimuli space based on reaching 100% coverage closure. Chatterjee et al. (2021) proposed a solution to automate finding scenarios that over-constrain the stimulus.

By relaxing constraints, simulation time is better used to close functional coverage. In the past few years, verification engineers and Engineering Design Automation (EDA) companies have been investing in machine learning techniques to optimize verification in various ways such as sequence generation (Gad et al., 2021; Gogri et al., 2020), Bayesian Optimization methods (Li & Kanoulas, 2018; Wu et al., 2019), Natural Language Processing (Chauhan, 2022), and regression automation (Chauhan & Ahmad, 2021).

## Bayesian Optimization

Bayesian optimization is a mathematical way to find the maximum of a cost function in the form $f(x)$ (Brochu et al., 2010). These optimization problems have the following form, where $A$ is a set of points in $\mathbb{R}^d$:

$$max_{x \in A \subset \mathbb{R}^d} f(x) \tag{2.1}$$

Bayesian optimization was derived from Bayes' Theorem which computes the probability of an event based on prior and historical knowledge. For Bayesian optimization, the mathematical formula is as follows, where $E$ is the data and $M$ is the model (Wu et al., 2019):

$$P(M|E) = \frac{P(E|M)P(M)}{P(E)} \tag{2.2}$$

Since the objective function is unknown, Bayesian optimization creates a probability model of the function. Using the sample points, $A$, a surrogate model is created to approximate the actual objective function. An acquisition function is then built to select the next hyperparameter to use to obtain the maxima of the approximated objective function. This process of choosing the next hyperparameter and adding it to the surrogate model continues until the maxima is found (Bergstra et al., 2011).

## Supervised Learning and Deep Neural Networks

Supervised Machine Learning algorithms are those that are trained using labeled data (Goodfellow et al., 2016). As an example, a model can be trained to predict credit card fraud if given labeled data that include both fraudulent and valid credit card transactions (Lee, 2019). This type of prediction is called Classification and is the most common type of ML algorithm (Goodfellow et al., 2016). A classification algorithm is one that, after being trained, predicts which $k$ category an input belongs to (Goodfellow et al., 2016). Using the credit card example, based on the inputs the transaction can be classified as valid or fraudulent. When the feature set of the input data becomes more complex, shallow learning algorithms may not be sufficient. In that case, Deep Learning algorithms are needed.

Deep Learning is a concept that relies upon Artificial Neural Networks (ANN) (Janiesch et al., 2021). Janiesch et al. (2021) defines ANNs as mathematical representations of artificial neurons that closely represent biological neurons. Deep Neural Networks (DNNs) are a subset of ANNs that consists of more hidden layers and more advanced neurons that may have convolutions or multiple activation functions on a neuron (Janiesch et al., 2021). Figure 4 shows a generic ANN with inputs, a hidden layers, and the subsequent outputs.

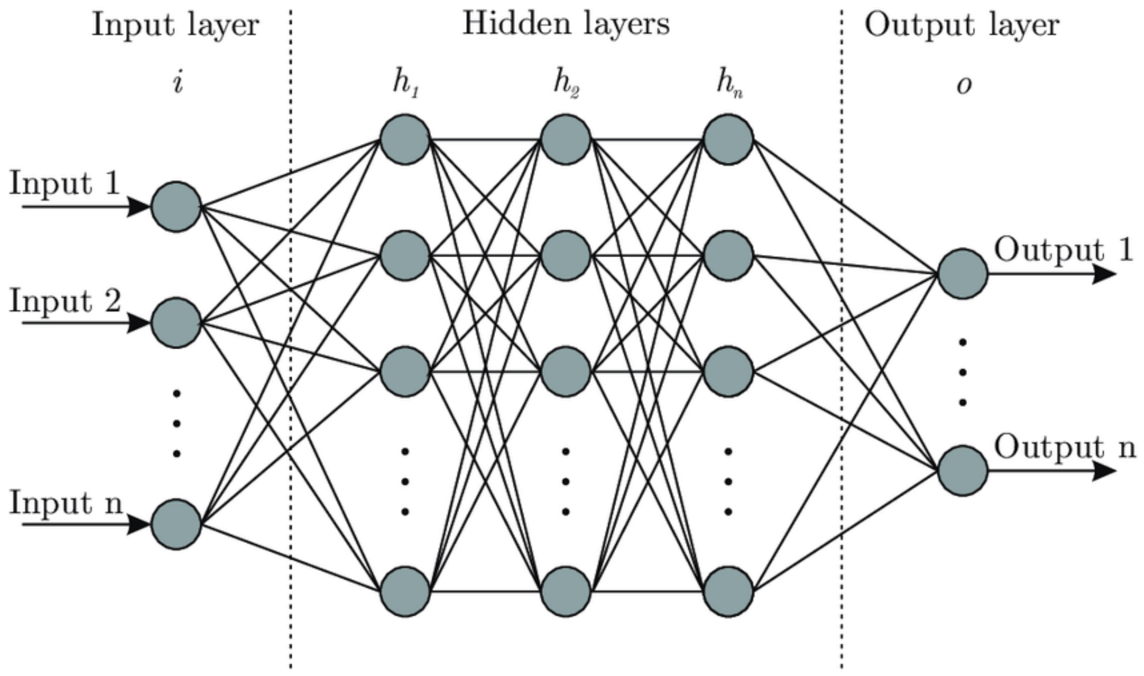


Figure 4. Artificial Neural Network

## Reinforcement Learning and the Deep-Q Network

Reinforcement Learning (RL) is a branch of unsupervised learning in which an agent interacts with the world in which it lives, with the goal of receiving rewards based on successes (Russell & Norvig, 2021). The agent acts using a trial and error approach to solving a problem; it is rewarded or penalized for the action (Hughes et al., 2019). It also receives an updated state of its environment based on the action as Figure 5 shows. For example, if action $a$ is a constraint fed into the environment in Figure 5 and a coverage point is hit, then a reward would be fed back to the agent. Similarly, if coverage is not hit, then a penalty would be sent back. In either case, the agent is updated with the new state.

The most common RL algoritm is Q-Learning in which the agent learns a Q-function, $Q(s, a)$, indicating the sum of rewards going forward from state $s$ if $a$, the action, is taken (Russell & Norvig, 2021). The goal is to maximize the reward using the following Bellman Equation where $s$ is a given state, $a$ is an action, $t$ is time, $t'$ is the time of the

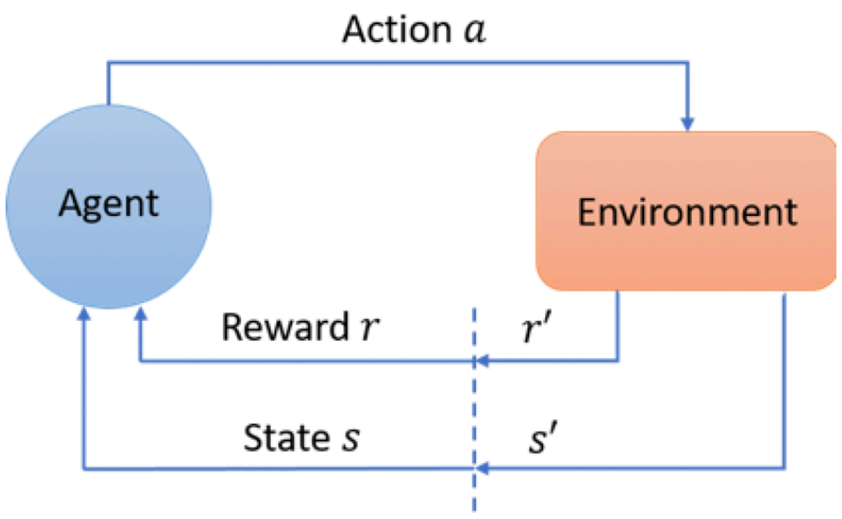


Figure 5. Reinforcement Learning (Hughes et al., 2019)

previous state, $r$ is the reward, and $\gamma$ is a discount factor (Hughes et al., 2019):

$$Q(s_{t'}, a_t) = E[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_{t'}, a_t] \qquad (2.3)$$

When the solution space is relatively small, the reward values can be stored in a Q-table. However, for a larger space which is typical for verification, a Q-table is not feasible and a Deep Neural Network (DNN) called a Deep Q-Network (DQN) is required (Hughes et al., 2019). A Deep Q-Network can include a DNN or a Convolutional Neural Network (CNN) with the differentiator being the use of Q-Learning. The DQN becomes a Q-value function approximator and learns the values for each action at a state (Hughes et al., 2019).

# Chapter 3

# Methodology

## Overview

The proposed study evaluates the merits of combining the works of Hughes et al. (2019) utilizing supervised and reinforcement learning with the Bayesian optimization deployed by Roy et al. (2021). The steps performed to conduct the study are outlined as follows:

1. Build a UVM testbench for a simple RTL design that can initially be used as a proof of concept. The design chosen is an Asynchronous FIFO (Pretet, 2020).
2. Generate coverage data using a standard constrained random approach that has been optimized by BayesOpt.
3. Add the DNN/DQN into the UVM simulation flow and generate coverage results.
4. Add the DNN/DQN into the UVM simulation flow filtered through BayesOpt and generate coverage results.
5. Once all environments are complete for the proof of concept, repeat with a more complex design such as a RISC-V core used by (OpenHW Group, 2021) or an Abstract Model.

Coverage results are the number of iterations of the simulation required to hit a certain coverage point. For example, if the coverage point is that a FIFO is full, how many iterations did it take to reach that condition.

Figure 6 illustrates this flow.

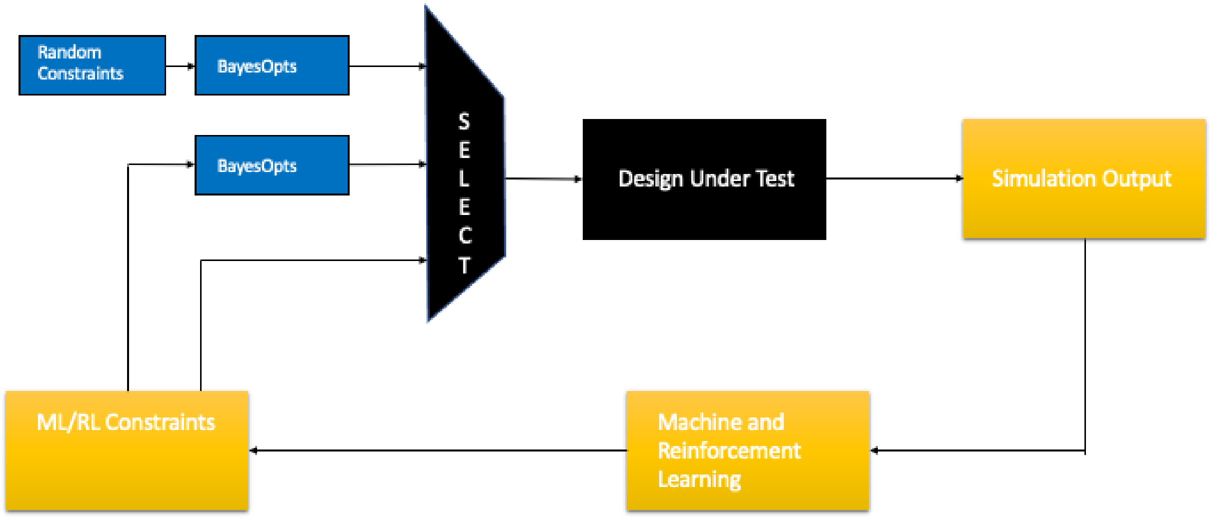


Figure 6. Proposed Simulation Flow

## Proof Of Concept

The experiment begins with a FIFO as a proof of concept to build the UVM environments and develop the ML implementations as simply as possible. A FIFO is a first in, first out block that is used to transfer data typically across different clock domains in a system. The Asynchronous FIFO that is used is Open Source and available on Githbub (Pretet, 2020). The random knobs (input features) in this scenario include the following: clock frequencies (24MHz, 48MHz, 96MHZ), FIFO depth (128 words, 512 words, 1024 words), data (16 bit, 32 bit, 64 bit), read, and write. The coverage (output) hit includes:

1. Same clock frequency on both sides of the FIFO
2. Different clock frequency on each side
3. Read and write enables both high

4. Read and write enable both low
5. Read enable high with write enable low
6. Read enable low with write enable high
7. Data ranges (minimum, maximum, ranges in between)
8. FIFO full
9. FIFO empty

The projected coverage results from each implementation, given the same number of simulation iterations, are similar given the simplicity.

## Abstract Model

To determine if the new hybrid ML optimization method is more successful, a more complex design is needed. The architecture used by (Hughes et al., 2019) was RISC-V. RISC-V is an open instruction set architecture (ISA) that is based on reduced instruction set computer (RISC) guidelines (Zaruba & Benini, 2019). The core and implementation can also be found on GitHub OpenHW Group (2021). However, given limitations in the environment that do not support the RISC-V compiler, an abstract model was created. The abstract model has a coverage space large enough to give a degree of complexity. Using an abstract model also has the flexibility of scaling the complexity.

Following the same steps as the proof of concept, a UVM environment is created for the abstract model. The model builds upon the FIFO in the proof of concept but adds a layer of complexity. The schematic for the model is shown in Figure 7. The FIFO is configurable for six different depths, four modes, and three data transformations.

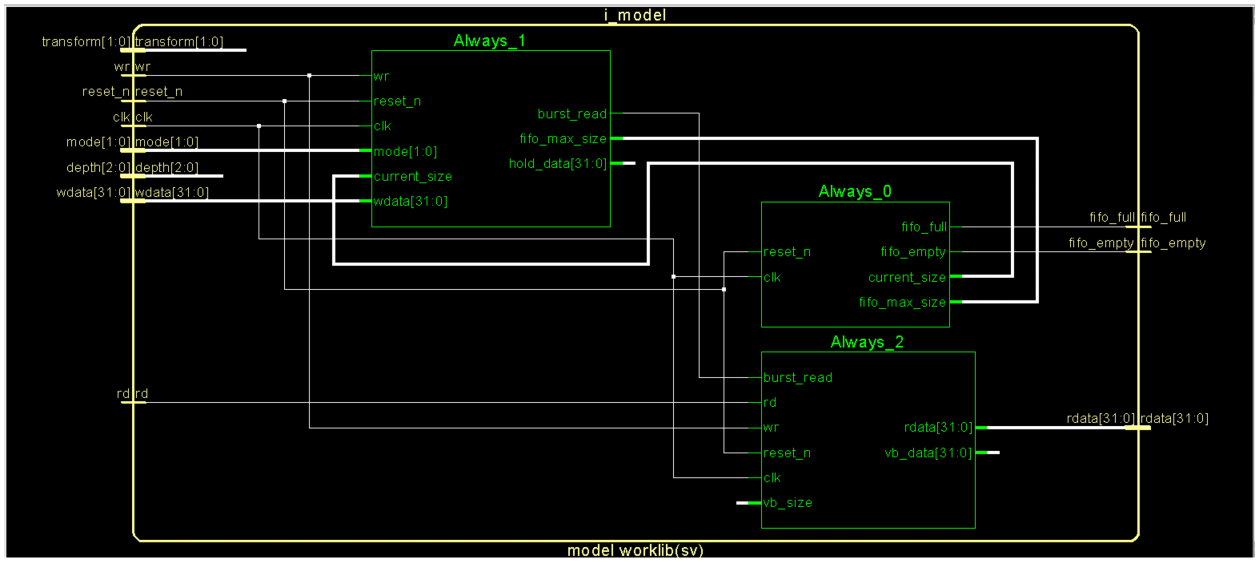


Figure 7. Abstract Model

Before the implementation of the simulation flow, the random knobs and functional coverage are defined, and the standard constrained random simulation run as a control in the experiment. This flow consists of the DNN/DQN used by Hughes et al. (2019). It also includes the BayesOpt implementation proposed by Roy et al. (2021). To be consistent with the previous work, the Scikit-Optimize library is also used (Kumar et al., 2020). Simulations are run evaluating GBRT, ET, and GP as surrogate models. The three acquisition functions used by Roy et al. (2021) are also evaluated. The number of coverage points hit per each iteration of the simulation are compared with the control outcome and the results of Hughes et al. (2019) and Roy et al. (2021). The final results show whether the combined or hybrid method produces results that improve upon those of the cited researchers.

## The Verification Environment

One of the first steps in any verification process is creating a verification plan based on the design specification. As a proof of concept, this experiment starts with the specification of the FIFO which is found in Appendix B. A typical verification plan includes what features are tested, how they are tested (stimulus), how they are checked (verified), and how they are covered (functional coverage). Since the focus of this study is coverage efficiency, the verification plan for the FIFO includes the stimulus and coverage and is found in Appendix A. The stimulus is required because it details the random knobs that are used to hit the coverage points.

After the verification plan is completed, the testbench is developed. Figure 8 shows another example of a UVM testbench with all of its various components.

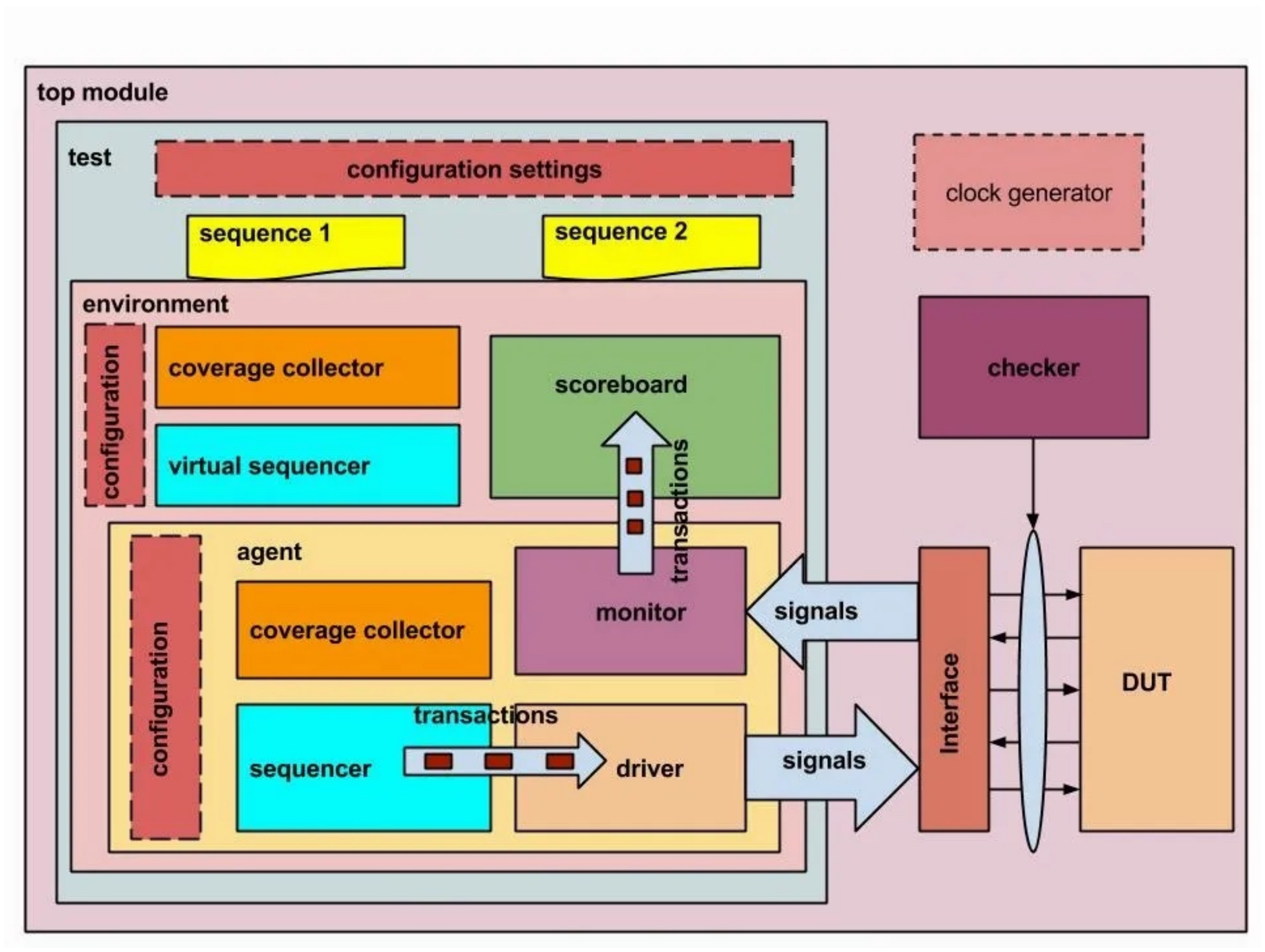


Figure 8. Full UVM Testbench (learnuvmverification.com)

The sequences contain constrained random variables (knobs) that are used in the transactions sent by the driver over the virtual interface to the DUT. Figure 8 also shows a coverage collector. This object subscribes to the transactions being sent, and the implemented coverage is updated. The data from the generated coverage report is used to inform and train the ML models. Once the full proof of concept environment is completed, the effort of verification planning and testbench development needs to be repeated for the abstract model. The respective specification and verification plans are found in Appendices B and A.

## BayesOpt Implementation

To implement the Bayesian Optimization component of the environment, the BayesOpt implementation provided in the Scikit-Optimize library as proposed by Roy et al. (2021) is used. Roy et al. (2021) utilized both an offline and an online flow. Given the hybrid nature of this experiment, only the offline flow is utilized, as illustrated in Figure 9.

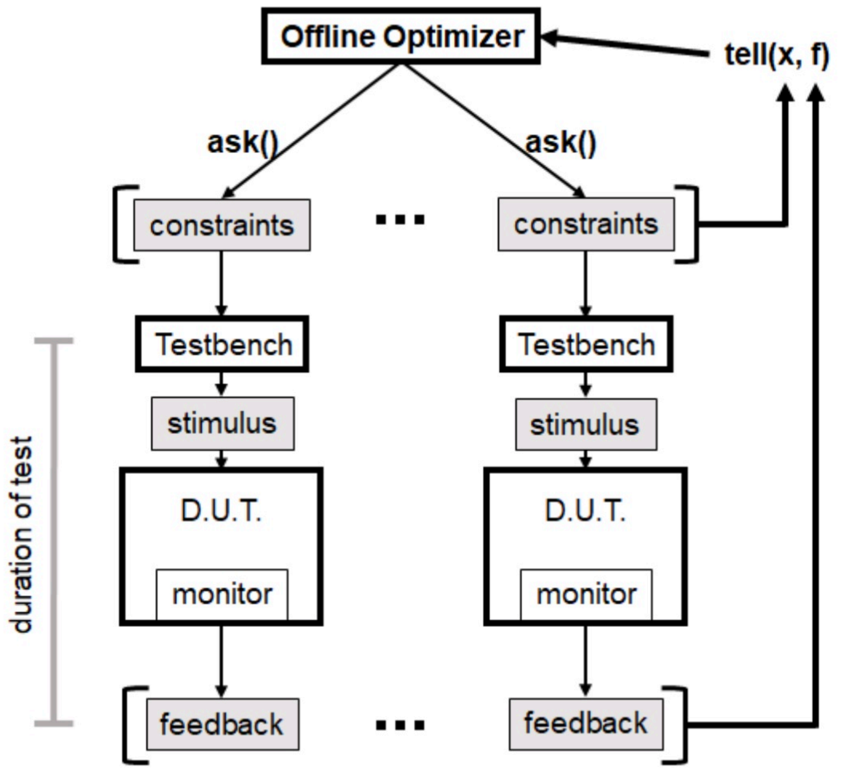


Figure 9. BayesOpt Flow (Roy et al., 2021)

## Machine Learning Models

As illustrated in Figure 6, once a simulation is done, the output is used as input in the ML models. Although the work by Hughes et al. (2019) states that a DNN is used, the work does not explicitly state the algorithm used. The first step after collecting data from several simulation runs is exploring the various models to determine which model or models work best. Given the level of complexity between the FIFO and the Abstract Model, the models needed for each environment are different. The models that are trained come from the scikit-learn classifier library.

## Reinforcement Learning

The final component of the environment is the Reinforcement Learning model. Similarly to the supervised learning models, the simulation output serves as input to the RL model. The model needs to find the optimal policy and maximize the reward return of its goal. In this case, the goal is to reach 100% functional coverage.

## Summary

To begin implementation, the first DUT is the Asynchronous FIFO. Step 1 is to create the Verification Plan (VPLAN) for the FIFO based on its specification found in Appendix B. The VPLAN can be found in Appendix A. The VPLAN outlines the constrained random knobs needed for the stimulus and the functional coverage plan. Once the plan is complete, a simple UVM testbench is created to drive the stimulus and collect coverage. Three selection paths through the DUT are illustrated in Figure 6. First, once the testbench is developed, the possible constrained random knobs are chosen using the BayesOpt flow described in Figure 9. The data captured in this flow is compared against the other two paths. The data of interest is the coverage closure rate based on the number of simulation runs. Also, it is noted if any functional coverage points are never hit.

Still utilizing the FIFO testbench, the second path also takes advantage of Bayesian Optimization. However, in this path, the stimulus is no longer just the constrained random values from the simulation tool constraint solver. The stimulus is a product of the Machine learning and Reinforcement learning models based on simulation output. Finally, the third path eliminates the Bayesian Optimization from the flow. Once this proof of concept experiment is complete, it is replicated on a more complex design using the Abstract Model. The more complex DUT emulates the real-world scenario of closing functional coverage across many simulation runs.

# Chapter 4

# Results

This chapter details the results of the experiments previously described. The first step, the Proof of Concept, was to create a UVM environment for a simple asynchronous FIFO as the DUT. Once the UVM environment was implemented, the next step was to create and prove the viability of the models: Bayesian Optimization, Supervised Learning, and Reinforcement Learning as shown in Figure 6. The Proof of Concept served as a development basis for the more complex Abstract Model environment.

## FIFO: Bayesian Optimization

The coverage model developed for the FIFO focused on the frequencies of the write and read clocks, whether or not a write or read was enabled, if the FIFO is full or empty, the size of the data (dsize), and the depth of the FIFO (asize). The clock frequencies were limited to 24MHz, 48MHz, and 98MHz. Based on the design, the data width and depth sizes were also constrained. The implemenation of the coverage model is found in Appendix A.

The constrained values for the coverpoints are as follows:

- write_freq $\in$ {0, 1, 2} (FREQ_24MHz, FREQ_48MHz, FREQ_96MHz)
- read_freq $\in$ {0, 1, 2} (FREQ_24MHz, FREQ_48MHz, FREQ_96MHz)
- write_enable $\in$ {0 , 1}
- read_enable $\in$ {0 , 1}

- full ∈ {0, 1}

- empty ∈ {0, 1}

- dsize ∈ {8, 16, 32, 64}
- asize ∈ {4, 8, 16, 32, 64, 128, 512, 1024}

As a baseline, three trial simulation sets, T1, T2, and T3, were conducted until 100% coverage was reached. Each trial simulation set consisted of individual runs and the cumulative coverage was captured after each run. Figure 10 shows that it took an average of 23 individual runs to reach full coverage.

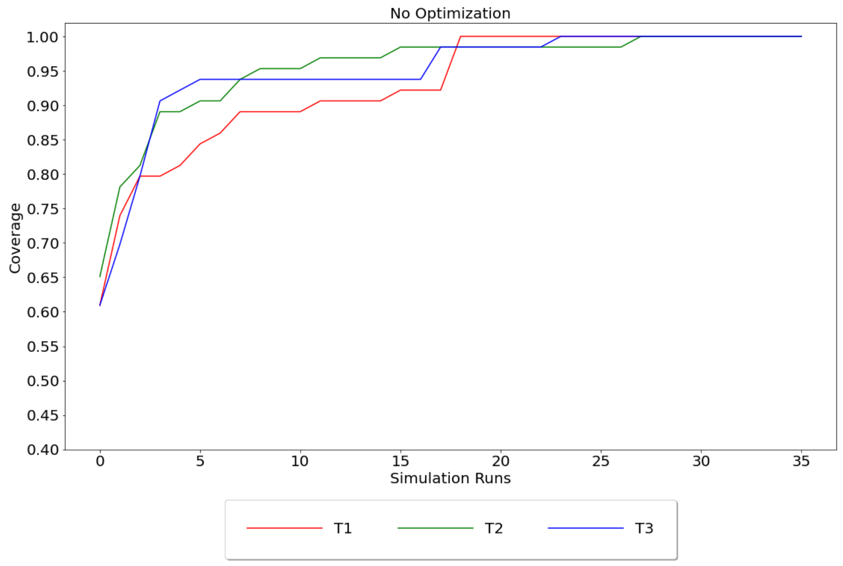
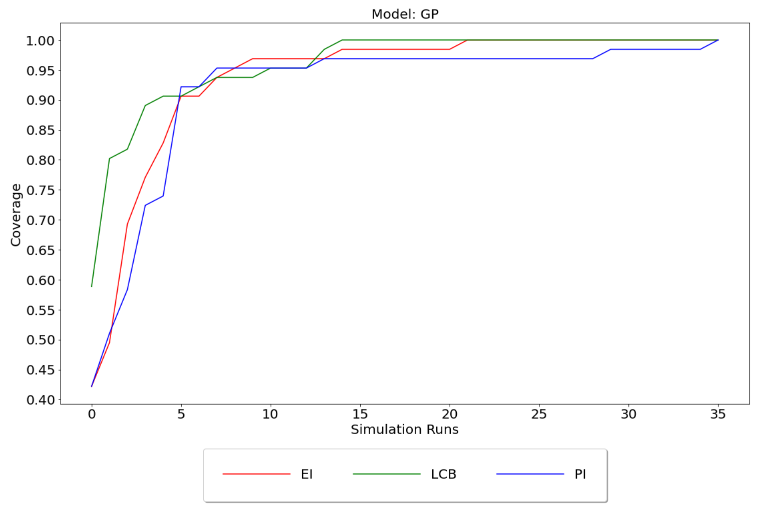


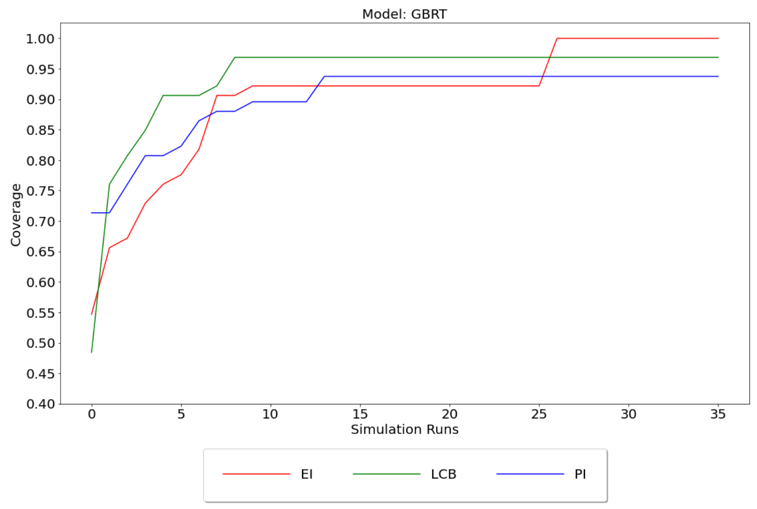
Figure 10. FIFO Simulation Results without Optimization

Using the work of Roy et al. (2021) as a baseline, an offline optimizer as described in Figure 9 was implemented. The surrogate models used were Gaussian Processes (GP), Gradient Boosted Regression Trees (GBRT), and Extra Trees (ET). Each surrogate model was run with three acquisition functions: Expected Improvement (EI), Lower Confidence Bound (LCB), and Probability of Improvement (PI). For this simple design, the constraints of the acquisition functions that balance the trade-off between exploration and exploitation were left as default (Carlen & Nahrstaedt, 2020).

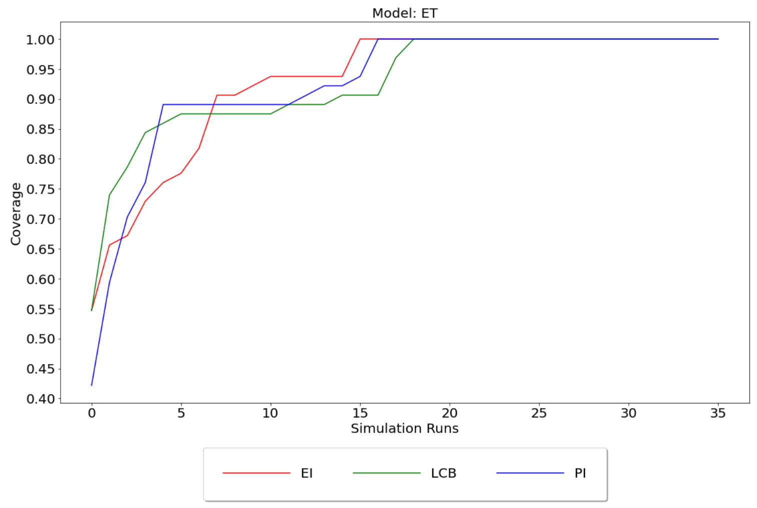
As observed in Figure 11, the GBRT model demonstrated the worst performance. Using the LCB and PI, acquisition functions, 100% coverage was never reached. The GP and ET models were both able to converge to 100% coverage, with ET reaching the goal with an average of 21 simulation runs. Since the Extra Trees model, performed best, it was used in the hybrid flow with the Reinforcement Learning model.



(a)



(b)



(c)

Figure 11. FIFO Simulation Results with Bayesian Optimization

## FIFO: Supervised Learning and Reinforcement Learning

The next step in the Proof of Concept was identifying a Supervised Machine Learning model. The first step was to train and test various Multilayer Perceptrons to identify one with a high degree of accuracy. The Scikit-Learn models that were trained were Multinomial NB, Gaussian NB, Decision Tree Classifier, Random Forest Classifier, Extra Trees Classifier, SGD Classifier, Ridge Classifier, Linear SVC, K Neighbors Classifier, SVC, Logistic Regression, and MLP Classifier. Of the twelve models evaluated, the Decision Tree Classifer was the most accurate at 70%. Although better than the other models, the accuracy was still low to make valid predictions. This result was not surprising. The features the model trained on were the six verification constraints. The binary output of whether or not coverage improved was based on a cumulative coverage score. For the model to be effectively accurate, it needed to have a concept of memory of the previous coverage. For this case, a variant of a Recurrent Neural Network (RNN) would be better suited (Pfeifer et al., 2020). Because each simulation run can be considered a time step and the coverage result depends on the previous values, a Long Short-Term Memory (LSTM) was chosen and incorporated into the Reinforcement Model.

For the RL model, the action space in the environment consisted of all possible permutations of the six feature constraints. Four possible actions were defined, as shown in Table 2. Using an epsilon greedy policy, one of the four actions was chosen in each step of an episode. Once epsilon decayed past a specified minimum, the action was selected based on target values from the historical data produced by the LSTM model.

| Action | Description |
|---|---|
| 0 | Choose a random constraint permutation |
| 1 | Choose a constraint to target the coverage case where the FIFO is full |
| 2 | Choose a constraint that targets covering the ASIZE feature |
| 3 | Choose a constraint that targets covering the DSIZE feature |

Table 2. FIFO: Reinforcement Learning Action Set

For the Proof of Concept toy problem, only one episode was performed. Each step produced a set of constraints and waited for the resulting coverage value. The reward was calculated as the current value of the cumulative coverage less the previous value. The episode was considered done once the cumulative coverage reached 1.0. As shown in Figure 12, three simulation sets, T1, T2, and T3, were performed, with one set resulting in coverage closure in 24 individual simulation runs. The results show for the Proof of Concept FIFO case, the ML/RL results are no better than the constrained random verification.

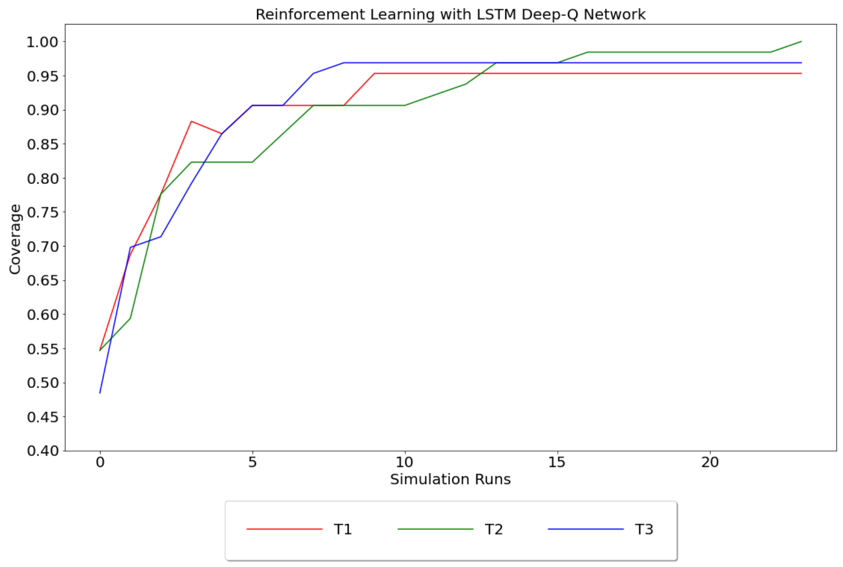


Figure 12. FIFO Simulation Results Reinforcement Learning using LSTM

## FIFO: Hybrid Approach

The final path is to use the Extra Trees (ET) model from BayesOpt in conjunction with the ML/RL model. Figure 6 showed the proposed flow of the constraints from the RL model being fed into the BayesOpt model. Given the nature of the RL model to produce constraints based on the policy and the reward, the proposed flow did not work. An amended flow was developed by which the initial batch of constraints was produced by the BayesOpt model. In other words, the position of the model changed in the flow, as illustrated in Figure 13. With a batch size of 8, the last 8 constraints from the BayesOpt ET model were used, resulting in a cumulative coverage of 95.31%. After the initial batch, the RL model produced constraints based on the calculated target and policy. Coverage closure, 100% was reached with an additional six simulation runs. With the models in place for the Proof of Concept, the next step was to utilize them in a more complex space with the Abstract Model.

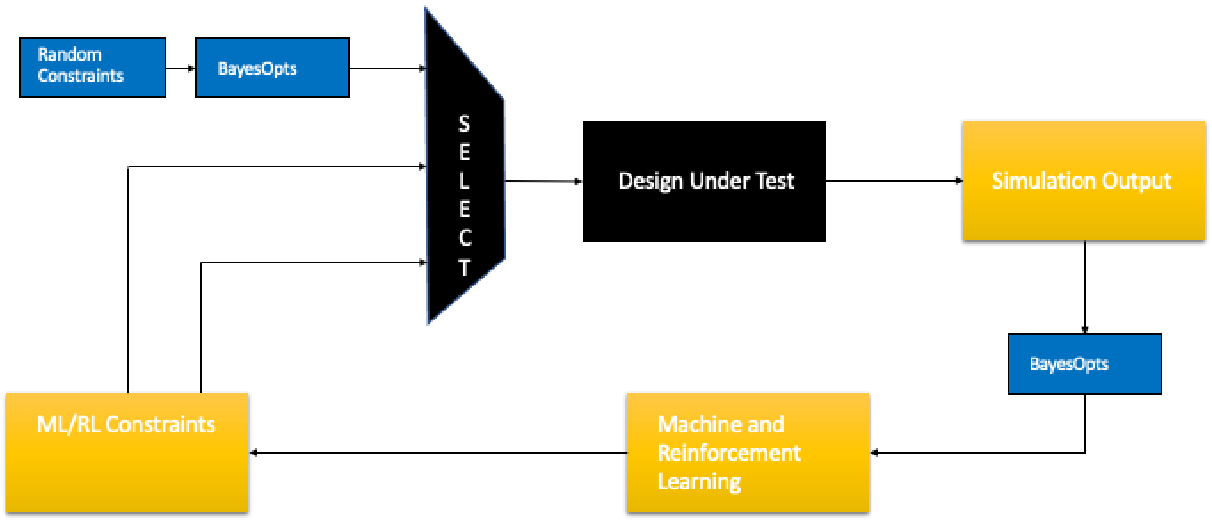


Figure 13. Amended Flow Simulation

## Abstract Model: Bayesian Optimization

For the Abstract Model implementation, the coverage model captured whether or not the FIFO was full, if all four modes were hit as well as the three possible data transformations, and the depth. To add to the complexity of the model, the depth could be scaled to a much larger value making coverage harder to achieve. Also adding to the coverage complexity is the cross coverage between the modes and transformations, and also cross coverage between the depth and if the FIFO was full. For example, cross_depth_full listed below covers whether the FIFO was full at all possible depths. The specification for the Abstract Model that details the functionality of the various modes and data transformations is found in Appendix B.

The constraints for each coverpoint for Abstract Model verification are as follows:

- fifo_full $\in \{0, 1\}$
- mode $\in \{0, 1, 2, 3\}$
- depth $\in \{8, 16, 32, 64, 128, 256\}$
- transform $\in \{0 , 1, 2\}$
- cross_mode_transform (all permutations of mode and transform)
- cross_depth_full (all permutations of depth and fifo_full)

The constraints (random knobs) needed for the simulation were write, read, mode, transform, and depth. As a baseline, groups of simulations were run to determine how many were required to achieve 100% coverage closure. Figure 14 shows that with the three trials of fifty simulation runs, two achieved full coverage closure while the remaining one was close at 98.61%.

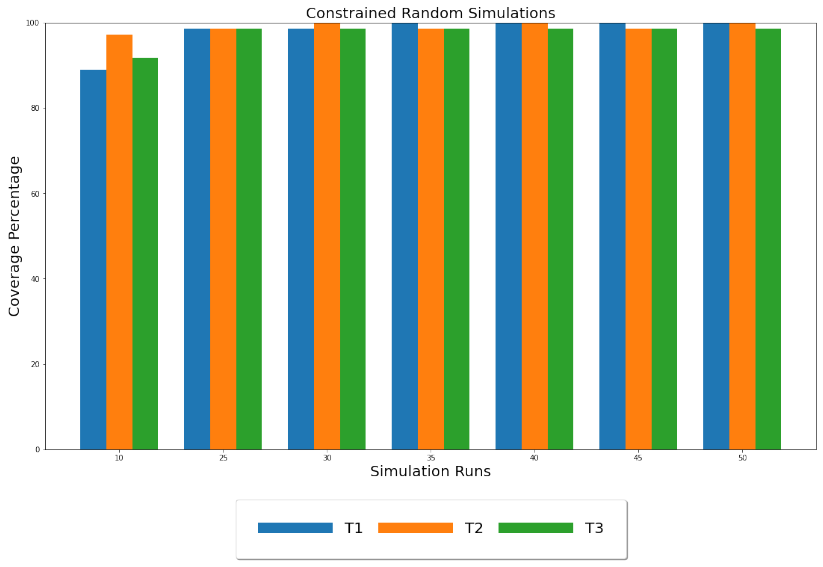
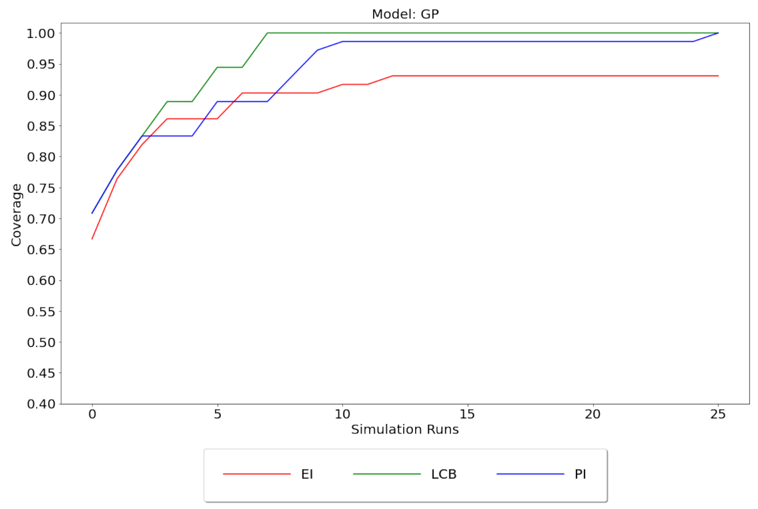
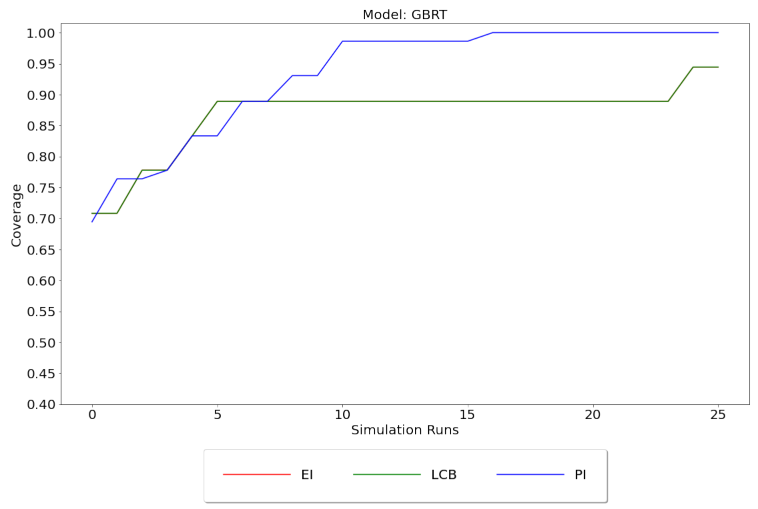


Figure 14. Abstract Model Baseline

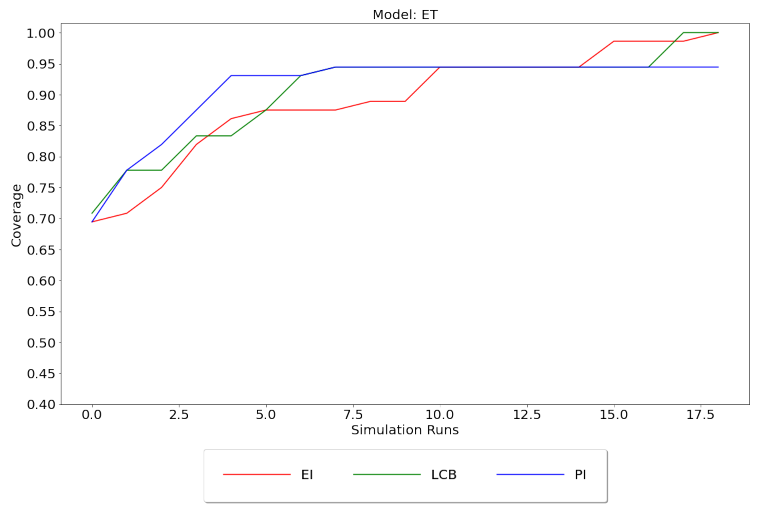
After the control simulations were completed, the optimized constraints were generated using Bayesian Optimization. To replicate the Proof of Concept experiment done with the FIFO, the Gaussian Processes (GP), Gradient Boosted Regression Trees (GBRT), and Extra Trees (ER) surrogate models were used. The same three acquisition functions were used as well. As with the FIFO, Figure 15, shows that the Extra Trees surrogate model performed best with the Expected Improvement (EI) and Lower Confidence Bound (LCB) acquisition functions. Both achieved 100% coverage in under 20 simulation runs.



(a)



(b)



(c)

Figure 15. Abstract Model Simulation Results with Bayesian Optimization

## Abstract Model: Hybrid Approach

The Reinforcement Learning Model with the LSTM Deep-Q Network created was used as a foundation for the Abstract Model. The action space consisted of all possible combinations of write, read, mode, and transform. Since each simulation run used the same depth along with a batch of constraints, the depth was chosen by the model separately. The initial batch of constraints used was a set generated from the ET/LCB combination. The remaining constraints were chosen based on the target values produced by the LSTM model. The updated actions taken by the policy are shown in Table 3.

| Action | Description |
|---|---|
| 0 | Choose a random constraint set permutation |
| 1 | Choose a constraint set to target the coverage case where the FIFO is full |
| 2 | Choose a constraint set that targets covering the Depth feature |
| 3 | Choose a constraint set that was generated using Bayesian Optimization |

Table 3. Abstract Model: Reinforcement Learning Action Set

Using this hybrid approach of combining Reinforcement Learning with Bayesian Optimization improved coverage closure results. Figure 16 shows that the first two simulation trails achieved closure in 13 and 12 runs, respectively. The final trial did not reach full coverage closure in this same scope due to the missing depth size of 8.

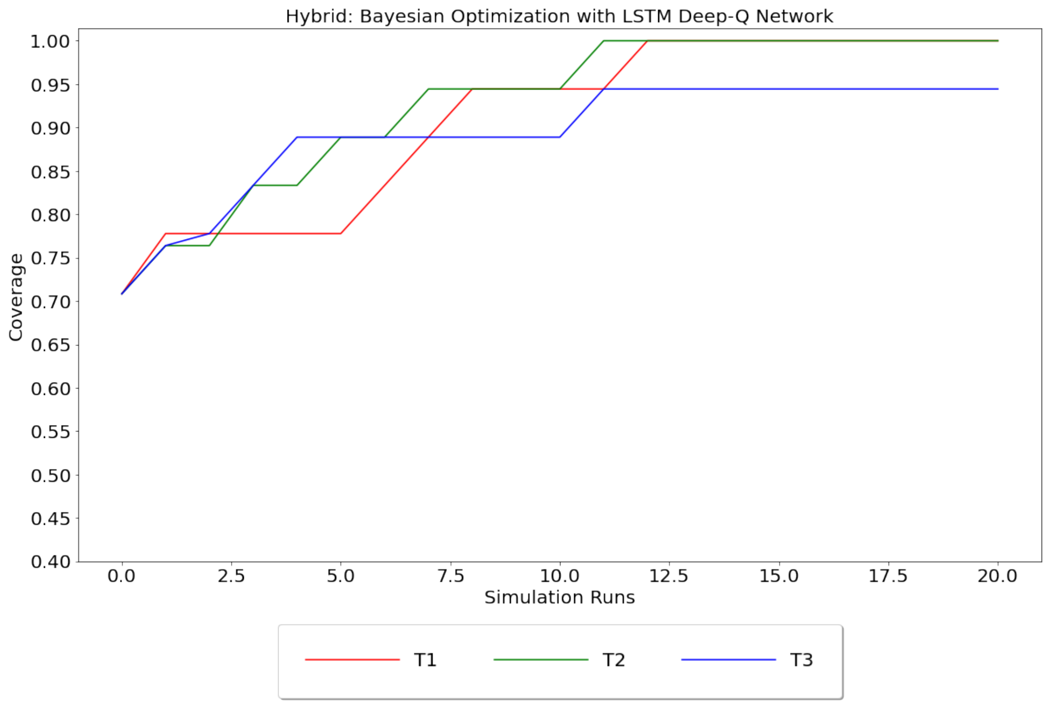


Figure 16. Abstract Model Hybrid Results

## Summary

Two different RTL designs were used: a simple FIFO and a more complex Abstract Model that built upon the FIFO. A UVM environment that was suitable to drive stimulus and collect coverage was developed for each. Once the verification environments were complete and the coverage spaces identified, control or baseline simulations were run to determine how many simulations were needed to reach 100% coverage, also known as coverage closure. Simulations were then run using Bayesian Optimization, Deep Neural Networks, and Reinforcement Learning. Results showed that for the FIFO, adding Machine Learning showed no significant improvement over constrained random. However, with the Abstract Model which was designed to be a more complex case, the improvement was measurable. For the baseline with just constrained random verification to using the hybrid Bayesian Optimization with Reinforcement Learning model full coverage was reached 74% faster.

# Chapter 5

# Conclusions, Implications, Recommendations, and Summary

## Conclusions

Constrained Random Verification is one of the most commonly used methodologies in Design Verification (Aboelmaged et al., 2021). Researchers have been exploring implementing Machine Learning techniques in the various aspects of verification. The most common explorations are test generation and optimizing constraints (Vangara et al., 2021). The goal of this study was to determine the effectiveness of using Bayesian Optimization in conjunction with Reinforcement Learning to optimize constraints. This study was derived from work done by Roy et al. (2021) who used Bayesian Optimization in both online and offline verification flows, and Hughes et al. (2019) who used a combination of Supervised Machine Learning and Reinforcement Learning.

Starting with the Proof of Concept case, the FIFO, three different flows were examined: Bayesian Optimization, Supervised Learning with a DNN, and Reinforcement Learning. The results showed that the most effective cases were using Bayesian Optimization and Reinforcement Learning with a Deep-Q LSTM model. The flow was also amended to use Bayesian Optimization constraints as the initial batch of constraints in the RL model. Using this amended flow, simulations were run using the Abstract Model. The baseline simulation sets using only constrained random data took approximately fifty individual simulation runs to reliably close coverage. By comparison, the hybrid model, only 12 to 13 individual runs per simulation set were required. The difference in the amount represents a 74% reduction of simulations required to close coverage. The hybrid combination of Bayesian Optimization with the LSTM Reinforcement model was demonstrably effective in optimizing the constraints.

## Implications and Limitations

Given that the design verification flow is a bottleneck in getting a chip to market, having an effective methodology to reduce the time is crucial (H. Foster, 2023). Although Engineering Design Automation (EDA) companies are implementing various AI techniques into their tools, changing or upgrading simulation tools is not always feasible. As a verification team, being able to utilize existing libraries such as scikit-learn will allow the team to increase their efficiency while also managing new costs. However, investing in having the correct infrastructure is required. In additional to the normal EDA tools and framework, engineers will need access to the libraries, and APIs will need to be created to access coverage data from the simulator.

One limitation that was encountered during the research was the compatibility of the OS with the latest Tensorflow library. Another limitation was the access to the needed compiler for a RISC-V processor case study. Limitations such as these would need to be addressed in the infrastructure for the verification team. If these and other limitations are addressed, then verification engineers will have less obstacles in adopting these ML models into their flow.

## Recommendations and Future Work

This study has illustrated an effective use case of Artificial Intelligence in the Hardware Design Verification space. To continue this work further, the next step would be automation to make the work easily reproducible and scalable. Both the Bayesian Optimization models and RL models were done using Google Colab. That data was then entered into the Linux-based simulation environment. The next goal would be to create a cohesive flow in the Linux-based environment. A wrapper would be created so that the engineer could execute the flow from ML to simulation and back with minimal manual effort. This automation would also allow further study of the models and adjusting of the hyperparameters to continue to achieve better results.

Another area of further study is to explore whether or not this hybrid approach is effective for a full System on Chip (SOC) design as opposed to block implementations used in this research.

## Summary

The intention of this research study was to utilize Artificial Intelligence in a Hardware Design Verification Environment to optimize the stimulus constraints to improve the rate at which a verification engineer can close coverage. Two different designs were used to test the effectiveness of Bayesian Optimization, Supervised Machine Learning, and Reinforcement Learning. A hybrid flow was created such that an initial batch of constraints was generated using the Extra Trees surrogate model. That initial batch was then used in the Reinforcement Learning model with a LSTM model as the Deep-Q Neural Network. This hybrid approach improved the coverage closure results by 74% using the Abstract Model design. Based on these results, the study has shown that implementing this hybrid model would improve coverage closure with similar designs. Further study is warranted to determine effectiveness full scale SOC designs as well as improve the automation of the flow.

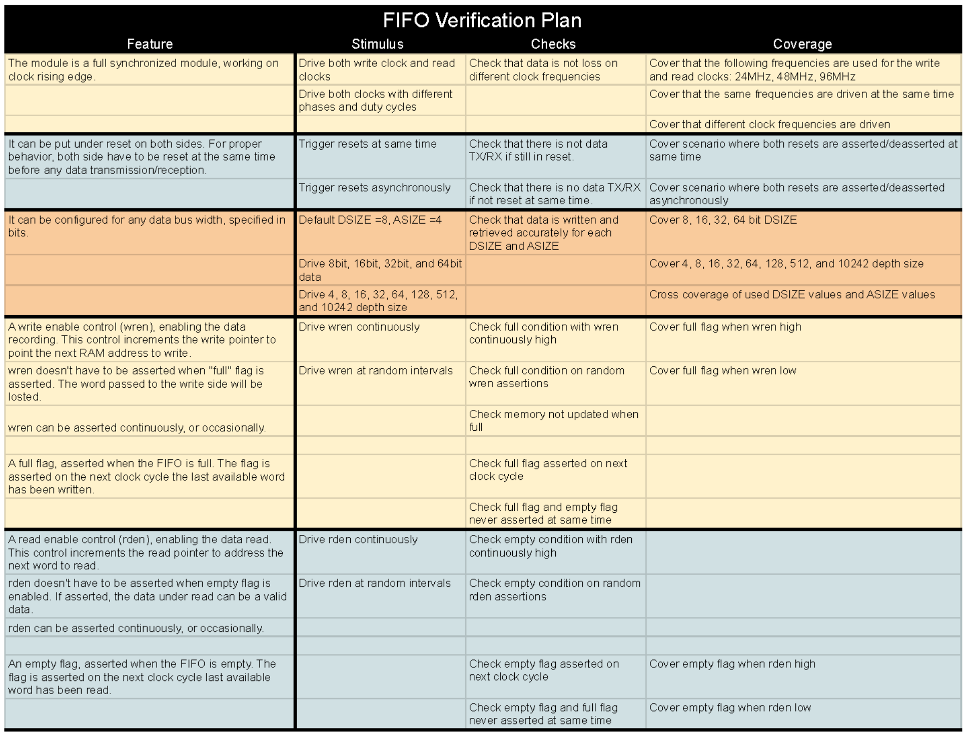
# Appendices

# Appendix A

# Verification Plans

## A.1 Verification Plan: FIFO

This Verification Plan (VPLAN) is an illustration of a simple plan that outlines the features that are to be tested, the stimulus to test them, how they will be checked, and the coverage collected on those features.



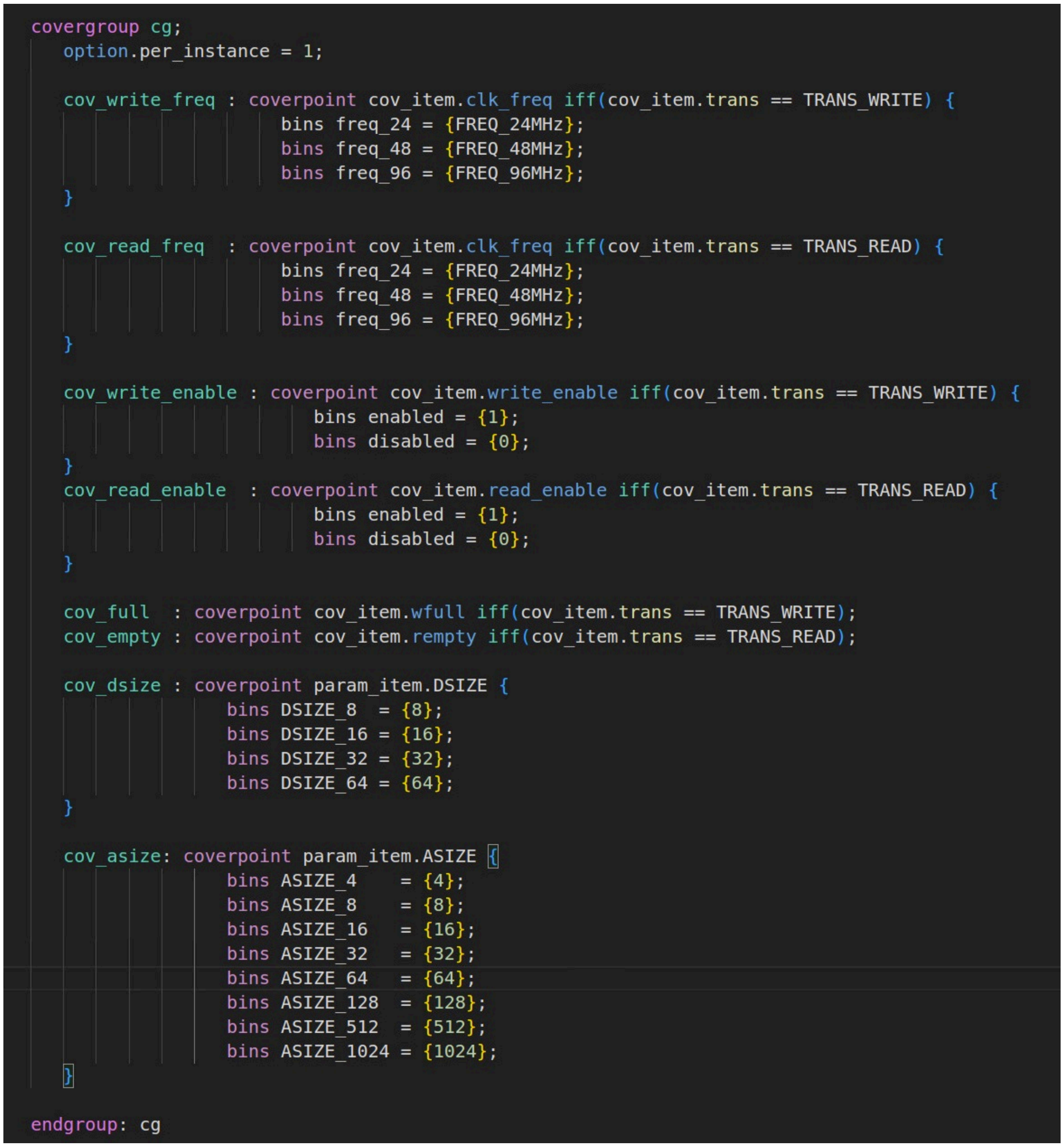
FIFO Verification Plan

| Feature | Stimulus | Checks | Coverage |
|---|---|---|---|
| The module is a full synchronized module, working on clock rising edge. | Drive both write clock and read clocks | Check that data is not loss on different clock frequencies | Cover that the following frequencies are used for the write and read clocks: 24MHz, 48MHz, 96MHz |
| | Drive both clocks with different phases and duty cycles | | Cover that the same frequencies are driven at the same time |
| | | | Cover that different clock frequencies are driven |
| It can be put under reset on both sides. For proper behavior, both side have to be reset at the same time before any data transmission/reception. | Trigger resets at same time | Check that there is not data TX/RX if still in reset. | Cover scenario where both resets are asserted/deasserted at same time |
| | Trigger resets asynchronously | Check that there is no data TX/RX if not reset at same time. | Cover scenario where both resets are asserted/deasserted asynchronously |
| It can be configured for any data bus width, specified in bits. | Default DSIZE =8, ASIZE =4 | Check that data is written and retrieved accurately for each DSIZE and ASIZE | Cover 8, 16, 32, 64 bit DSIZE |
| | Drive 8bit, 16bit, 32bit, and 64bit data | | Cover 4, 8, 16, 32, 64, 128, 512, and 10242 depth size |
| | Drive 4, 8, 16, 32, 64, 128, 512, and 10242 depth size | | Cross coverage of used DSIZE values and ASIZE values |
| A write enable control (wren), enabling the data recording. This control increments the write pointer to point the next RAM address to write. | Drive wren continuously | Check full condition with wren continuously high | Cover full flag when wren high |
| wren doesn't have to be asserted when "full" flag is asserted. The word passed to the write side will be losted. | Drive wren at random intervals | Check full condition on random wren assertions | Cover full flag when wren low |
| wren can be asserted continuously, or occasionally. | | Check memory not updated when full | |
| | | | |
| A full flag, asserted when the FIFO is full. The flag is asserted on the next clock cycle the last available word has been written. | | Check full flag asserted on next clock cycle | |
| | | Check full flag and empty flag never asserted at same time | |
| A read enable control (rden), enabling the data read. This control increments the read pointer to address the next word to read. | Drive rden continuously | Check empty condition with rden continuously high | |
| rden doesn't have to be asserted when empty flag is enabled. If asserted, the data under read can be a valid data. | Drive rden at random intervals | Check empty condition on random rden assertions | |
| rden can be asserted continuously, or occasionally. | | | |
| | | | |
| An empty flag, asserted when the FIFO is empty. The flag is asserted on the next clock cycle last available word has been read. | | Check empty flag asserted on next clock cycle | Cover empty flag when rden high |
| | | Check empty flag and full flag never asserted at same time | Cover empty flag when rden low |

Figure 17. FIFO Verification Plan

## A.2 Coverage Model: FIFO

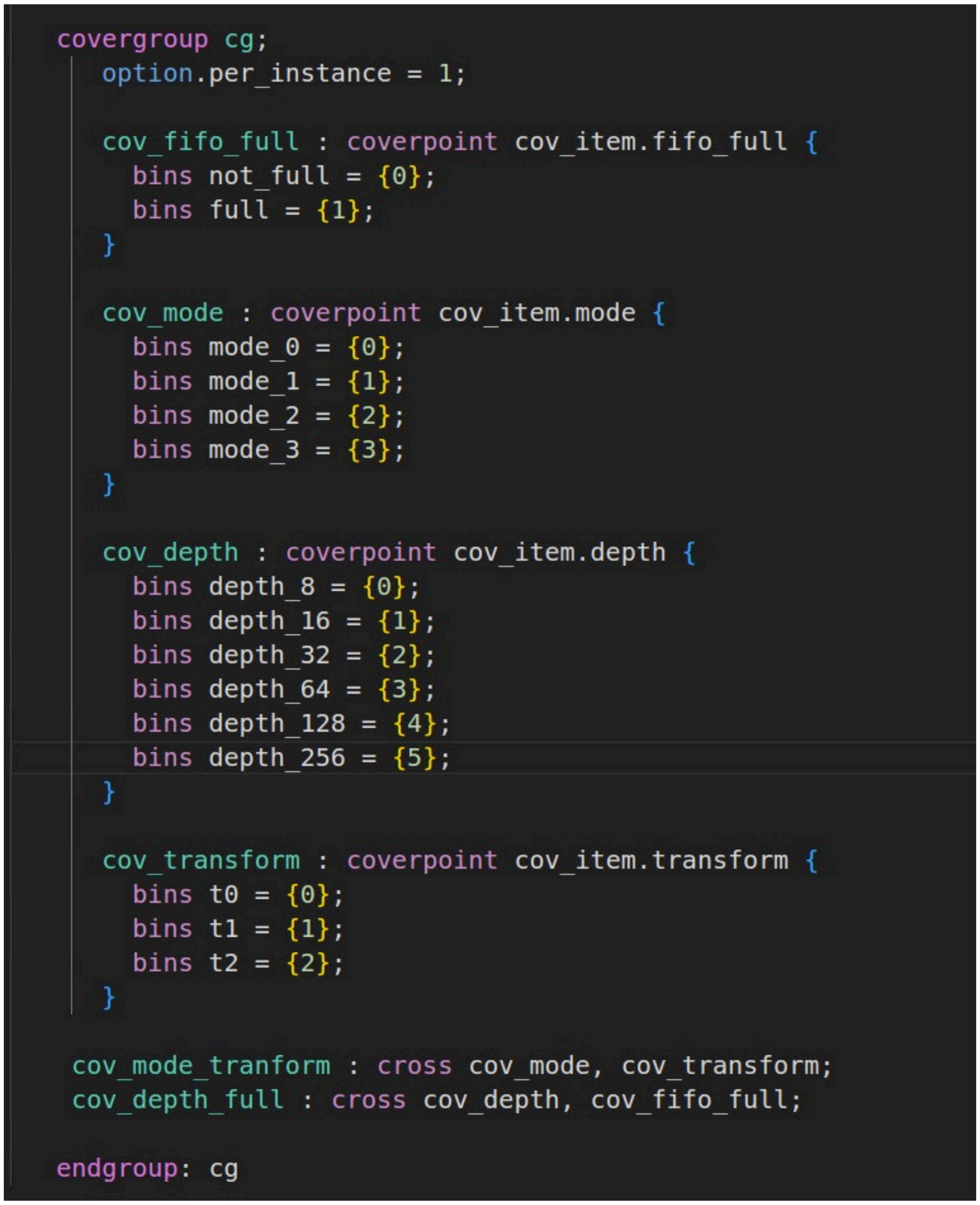
SystemVerilog coverage is developed in an object called a covergroup. This code outlines the features to be covered and how they are placed in bins. There is also cross coverage that creates a matrix from the items to be covered.



```
covergroup cg;
   option.per_instance = 1;

   cov_write_freq : coverpoint cov_item.clk_freq iff(cov_item.trans == TRANS_WRITE) {
                          bins freq_24 = {FREQ_24MHz};
                          bins freq_48 = {FREQ_48MHz};
                          bins freq_96 = {FREQ_96MHz};
   }

   cov_read_freq  : coverpoint cov_item.clk_freq iff(cov_item.trans == TRANS_READ) {
                          bins freq_24 = {FREQ_24MHz};
                          bins freq_48 = {FREQ_48MHz};
                          bins freq_96 = {FREQ_96MHz};
   }

   cov_write_enable : coverpoint cov_item.write_enable iff(cov_item.trans == TRANS_WRITE) {
                             bins enabled = {1};
                             bins disabled = {0};
   }
   cov_read_enable  : coverpoint cov_item.read_enable iff(cov_item.trans == TRANS_READ) {
                             bins enabled = {1};
                             bins disabled = {0};
   }

   cov_full  : coverpoint cov_item.wfull iff(cov_item.trans == TRANS_WRITE);
   cov_empty : coverpoint cov_item.rempty iff(cov_item.trans == TRANS_READ);

   cov_dsize : coverpoint param_item.DSIZE {
                  bins DSIZE_8  = {8};
                  bins DSIZE_16 = {16};
                  bins DSIZE_32 = {32};
                  bins DSIZE_64 = {64};
   }

   cov_asize: coverpoint param_item.ASIZE {
                  bins ASIZE_4    = {4};
                  bins ASIZE_8    = {8};
                  bins ASIZE_16   = {16};
                  bins ASIZE_32   = {32};
                  bins ASIZE_64   = {64};
                  bins ASIZE_128  = {128};
                  bins ASIZE_512  = {512};
                  bins ASIZE_1024 = {1024};
   }

endgroup: cg
```

Figure 18. FIFO Coverage

## A.3 Coverage Model: Abstract Model



```
covergroup cg;
  option.per_instance = 1;

  cov_fifo_full : coverpoint cov_item.fifo_full {
    bins not_full = {0};
    bins full = {1};
  }

  cov_mode : coverpoint cov_item.mode {
    bins mode_0 = {0};
    bins mode_1 = {1};
    bins mode_2 = {2};
    bins mode_3 = {3};
  }

  cov_depth : coverpoint cov_item.depth {
    bins depth_8 = {0};
    bins depth_16 = {1};
    bins depth_32 = {2};
    bins depth_64 = {3};
    bins depth_128 = {4};
    bins depth_256 = {5};
  }

  cov_transform : coverpoint cov_item.transform {
    bins t0 = {0};
    bins t1 = {1};
    bins t2 = {2};
  }

 cov_mode_tranform : cross cov_mode, cov_transform;
 cov_depth_full : cross cov_depth, cov_fifo_full;

endgroup: cg
```

Figure 19. Abstract Model Coverage

# Appendix B

# Specifications

## B.1 Specification: FIFO

This document lists all the features the FIFO supports : (Pretet, 2020)

- The module is a full synchronized module, working on clock rising edge.
- It can be put under reset on both sides. For proper behavior, both side have to be reset at the same time before any data transmission/reception.
- It can be synthesized either for Xilinx and Altera FPGAs.
- It supports built-in RAM of the FPGAs, used by inference. This ensures an easy way to include the module in a design, regardless the FPGA family.
- It can be configured for any data bus width, specified in bits.
- Its depth can be configured in bytes.
  - If the FIFO depth is not modulo the datapath, the real FIFO depth infered is round up to the next datapath width. For instance, if the datapath width is 16 bytes and the depth specified being 20 bytes, the effective FIFO size will be 32 bytes
  - If the depth is modulo the datapath, the specified depth will be the effective depth
- The FIFO handles for the user all the RAM addressing.

- The FIFO is composed by two asynchronous sides
  - Write side uses
    - A write enable control (wren), enabling the data recording. This control increments the write pointer to point the next RAM address to write.
      - wren doesn't have to be asserted when "full" flag is asserted. The word passed to the write side will be losted.
      - wren can be asserted continuously, or occasionally.
    - A full flag, asserted when the FIFO is full. The flag is asserted on the next clock cycle the last available word has been written.
    - A data bus, passing the information to store.
  - Read side uses
    - A read enable control (rden), enabling the data read. This control increments the read pointer to address the next word to read.
      - rden doesn't have to be asserted when empty flag is enabled. If asserted, the data under read can be a valid data.
      - rden can be asserted continuously, or occasionally.
    - An empty flag, asserted when the FIFO is empty. The flag is asserted on the next clock cycle last available word has been read.
    - A data bus, receiving the information to read.

## B.2 Specification: Abstract Model

The Abstract Model contains a FIFO with depths 8, 16, 32, 64, 128, and 256. The depth can be scaled to any length. It also contains a Victim Buffer of depth 4. The functionality of the model is based on the modes shown in Table 4. The corresponding transformations that are enabled based on the Mode selection are detailed in Table 5. The implementation is detailed in Appendix C.

| Mode | Description |
|---|---|
| 0 | Normal read and write operations at the positive edge of the clock. If the FIFO is full, write data is lost. |
| 1 | If the FIFO is full on a write, data is written to the victim buffer if it is not full. On a read, if the FIFO is full, data is read and an entry from the victim buffer is moved to the FIFO. |
| 2 | Normal operation like Mode 0, but the write data goes through a transformation based on the setting. |
| 3 | Victim buffer operation like Mode 1, but the write data goes through a transformation based on the setting. |

Table 4. Abstract Model Modes

| Transform | Description |
|---|---|
| 0 | Data is shifted left by 2. |
| 1 | Data is shifted right by 2. |
| 2 | The endianness of the data is swapped. In other words, bit 0 is not bit 31 and so forth. |

Table 5. Abstract Model Transforms

# Appendix C

# Designs Used: RTL

## C.1 Asynchronous Fifo

```
//-------------------------------------------------------------------------
// Copyright 2017 Damien Pretet ThotIP
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//-------------------------------------------------------------------------

`timescale 1 ns / 1 ps
`default_nettype none

module async_fifo
```

```

#(
parameter DSIZE = 8,
parameter ASIZE = 4,
parameter FALLTHROUGH = "TRUE" // First word fall-through
)(
input  wire             wclk,
input  wire             wrst_n,
input  wire             winc,
input  wire [DSIZE-1:0] wdata,
output wire             wfull,
output wire             awfull,
input  wire             rclk,
input  wire             rrst_n,
input  wire             rinc,
output wire [DSIZE-1:0] rdata,
output wire             rempty,
output wire             arempty
);

wire [ASIZE-1:0] waddr, raddr;
wire [  ASIZE:0] wptr, rptr, wq2_rptr, rq2_wptr;

// The module synchronizing the read point
// from read to write domain
sync_r2w
#(ASIZE)
sync_r2w (
.wq2_rptr (wq2_rptr),
.rptr     (rptr),
.wclk     (wclk),
.wrst_n   (wrst_n)
);

// The module synchronizing the write point
// from write to read domain
sync_w2r
#(ASIZE)
sync_w2r (
.rq2_wptr (rq2_wptr),
.wptr     (wptr),
.rclk     (rclk),
.rrst_n   (rrst_n)
);

// The module handling the write requests
wptr_full
#(ASIZE)
wptr_full (
.awfull   (awfull),
.wfull    (wfull),
.waddr    (waddr),
.wptr     (wptr),
.wq2_rptr (wq2_rptr),
.winc     (winc),
.wclk     (wclk),
.wrst_n   (wrst_n)
);
```

```

// The DC-RAM
fifomem
#(DSIZE, ASIZE, FALLTHROUGH)
fifomem (
.rclken (rinc),
.rclk   (rclk),
.rdata  (rdata),
.wdata  (wdata),
.waddr  (waddr),
.raddr  (raddr),
.wclken (winc),
.wfull  (wfull),
.wclk   (wclk)
);

// The module handling read requests
rptr_empty
#(ASIZE)
rptr_empty (
.arempty  (arempty),
.rempty   (rempty),
.raddr    (raddr),
.rptr     (rptr),
.rq2_wptr (rq2_wptr),
.rinc     (rinc),
.rclk     (rclk),
.rrst_n   (rrst_n)
);
```

```

endmodule

`resetall
```

## C.2 Abstract Model

```
//----------------------------------------------------------------------
// Copyright 2023 Vanessa Cooper
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
//----------------------------------------------------------------------

`timescale 1 ns / 1 ps

module model
(
    input clk,
    input reset_n,
    input wr,
    input rd,
    input [1:0] mode,
    input [1:0] transform,
    input [2:0] depth,
    input [31:0] wdata,
    output reg       fifo_full,
    output reg       fifo_empty,
    output reg [31:0] rdata
);

   //Fifo Depth:
   // 0 = 8
   // 1 = 16
   // 2 == 32
   // 3 == 64
   // 4 == 128
   // 5 == 512

   bit [31:0] fifo[$];
   bit [31:0] victim_buffer[$:3];
   bit [31:0] hold_data;
   bit [31:0] vb_data;
   int fifo_max_size;
   int current_size;
   int vb_size;
   bit burst_read;

```

```
//Flags
always @(posedge clk) begin : fifo_flags
   if(reset_n === 1'b1) begin
     current_size = fifo.size();
     fifo_max_size = get_max_size();
     if((current_size == fifo_max_size) && current_size != 0) begin
       fifo_full <= 1'b1;
       fifo_empty <= 1'b0;
     end
     else if(current_size > 0 && current_size < fifo_max_size) begin
       fifo_empty <= 1'b0;
       fifo_full <= 1'b0;
     end
     else if(current_size == 0) begin
          fifo_empty <= 1'b1;
          fifo_full <= 1'b0;
     end
   end
end

//Write
always @(posedge clk) begin : write
   if(reset_n === 1'b1) begin
     if(wr) begin
       fifo_max_size = get_max_size();
       case(mode)
          0: begin
               burst_read = 0;
               if(current_size < fifo_max_size) begin
```

```
                    fifo.push_back(wdata);
                  end
                end
             1: begin
                  burst_read = 1;
                  if((current_size == fifo_max_size) && current_size != 0) begin //push to victim buffer
                    if(victim_buffer.size() < 4)
                       victim_buffer.push_back(wdata);
                  end
                  else if(current_size < fifo_max_size) begin
                    fifo.push_back(wdata);
                  end
                end
             2: begin
                  burst_read = 0;
                  if(current_size < fifo_max_size) begin
                    hold_data = transform_data(wdata);
                    fifo.push_back(hold_data);
                  end
                end
             3: begin
                  burst_read = 1;
                  hold_data = transform_data(wdata);
                  if((current_size == fifo_max_size) && current_size != 0) b
                    if(victim_buffer.size() < 4)
                    victim_buffer.push_back(hold_data);
                  end
                  else if(current_size < fifo_max_size) begin
```

```
                 fifo.push_back(hold_data);
               end
             end
       endcase
     end
   end //!reset_n
end //always

//Read
always @(posedge clk) begin : read
  if(reset_n === 1'b1) begin
     if(rd == 1 && wr == 0) begin
       if(burst_read) begin
         rdata <= fifo.pop_front();
         if(vb_size != 0) begin
           vb_data = victim_buffer.pop_front();
           fifo.push_back(vb_data);
         end
       end
       else begin
         rdata <= fifo.pop_front();
       end
     end //rd && !wr
  end //!reset_n
end //always

function int get_max_size();
   case(depth)
      3'b000: return 8;
```

```
          3'b001: return 16;
          3'b010: return 32;
          3'b011: return 64;
          3'b100: return 128;
          3'b101: return 256;
       endcase
    endfunction

    function bit [31:0] transform_data(bit [31:0] data);
       bit [31:0] t_data;
       if(transform == 0) begin
         t_data = data << 2;
       end
       else if (transform == 1) begin
         t_data = data >> 2;
       end
       else if (transform == 2) begin
         for(int i=0,j=31; i<32; i++, j--) begin
             t_data[j] = data[i];

         end
       end
       return t_data;
    endfunction




endmodule
```

## References

Aboelmaged, M., Mashaly, M., & Abd El Ghany, M. A. (2021). Online constraints update using machine learning for accelerating hardware verification. https://doi.org/10.1109/niles53778.2021.9600485

Bergstra, J., Bardenet, R., Bengio, Y., & Kégl, B. (2011). Algorithms for hyper-parameter optimization. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, & K. Q. Weinberger (Eds.), *Advances in neural information processing systems*. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2011/file/86e8f7ab32cfd12577bc2619bc635690-Paper.pdf

Brochu, E., Cora, V. M., & De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

Carlen, S., & Nahrstaedt, H. (2020). *Exploration vs. exploitation*. https://scikit-optimize.github.io/stable/auto_examples/exploration-vs-exploitation.html

Chatterjee, D., Kachhadiya, S., Bayraktaroglu, I., & Dhodhi, S. (2021). Systematic constraint relaxation scr hunting for over constrained stimulus.

Chauhan, A. (2022). Automatic translation of natural language to systemverilog assertions.

Chauhan, A., & Ahmad, A. (2021). Ml-based veriification and regression automation.

Chen, W., Ray, S., Bhadra, J., Abadir, M., & Wang, L. C. (2017). Challenges and trends in modern soc design verification. *IEEE Design amp; Test*, *34*(5), 7–22. https://doi.org/10.1109/mdat.2017.2735383

Cooper, V. R. (2013). *Getting started with uvm: A beginner's guide* (1st ed.). Verilab Publishing.

Foster, H. D. (2015). Trends in functional verification: A 2014 industry study. *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 1–6. https://doi.org/10.1145/2744769.2744921

Foster, H. (2023). Part 8: The 2022 wilson research group functional verification study. https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/

Gad, M., Aboelmaged, M., Mashaly, M., & El Ghany, M. A. A. (2021). Efficient sequence generation for hardware verification using machine learning. https://doi.org/10.1109/icecs53924.2021.9665495

Gogri, S., Hu, J., Tyagi, A., Quinn, M., Ramachandran, S., Batool, F., & Jagadeesh, A. (2020). Machine learning-guided stimulus generation for functional verification. *Design and Verification conference 2020, virtual conference*.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* [http://www.deeplearningbook.org]. MIT Press.

Hughes, W., Srinivasan, S., Suvarna, R., & Kulkarni, M. (2019). Optimizing design verification using machine learning: Doing better than random. *CoRR*, *abs/1909.13168*. http://arxiv.org/abs/1909.13168

Janiesch, C., Zschech, P., & Heinrich, K. (2021). Machine learning and deep learning. *Electronic Markets*, *31*(3), 685–695. https://doi.org/10.1007/s12525-021-00475-2

Kumar, M., Head, T., Louppe, G., Shcherbatyi, I., Nahrstaedt, H., & Contributors. (2020, September 1). *Scikit-optimize*. https://scikit-optimize.github.io/stable/

Lee, W. (2019). *Python machine learning*. Wiley. https://books.google.com/books?id=c6W0zQEACAAJ

Li, D., & Kanoulas, E. (2018). Bayesian optimization for optimizing retrieval systems. *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, 360–368. https://doi.org/10.1145/3159652.3159665

OpenHW Group. (2021, March 31). *An open source 64-bit RISC-VApplication*. https://github.com/openhwgroup/cva6

Pfeifer, N., Zimpel, B. V., Andrade, G. A. G., & Santos, L. C. V. d. (2020). A reinforcement learning approach to directed test generation for shared memory verification.

Pretet, D. (2020, September 11). *$async_fifo$*. https://github.com/dpretet/async_fifo

Roy, R., Benipal, M. S., & Godil, S. Dynamically optimized test generation using machine learning. In: DVCON US. 2021, March, 6.

Russell, S. J., & Norvig, P. (2021). *Artificial intelligence: A modern approach* (4th ed.). Pearson.

Sethulekshmi, R., Jazir, S., Rahiman, R. A., Karthik, R., Abdulla, M. S., & Sree Swathy, S. (2016). Verification of a risc processor ip core using systemverilog. https://doi.org/10.1109/wispnet.2016.7566385

Vanaraj, A. T., Raj, M., & Gopalakrishnan, L. (2020). Functional verification closure using optimal test scenarios for digital designs. *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*, 535–538. https://doi.org/10.1109/ICSSIT48917.2020.9214097

Vangara, R. K. M., Kakani, B., & Vuddanti, S. (2021). An analytical study on machine learning approaches for simulation-based verification. https://doi.org/10.1109/icissgt52025.2021.00049

Wang, C., Tseng, C.-H., Tsai, C.-C., Lee, T.-Y., Chen, Y.-H., Yeh, C.-H., Yeh, C.-S., & Lai, C.-T. (2022). Two stage framework for corner case stimuli generation using transformer and reinforcement learning.

Wang, F., Zhu, H., Popli, P., Xiao, Y., Bodgan, P., & Nazarian, S. (2018). Accelerating coverage directed test generation for functional verification. https://doi.org/10.1145/3194554.3194561

Wu, J., Chen, X.-Y., Zhang, H., Xiong, L.-D., Lei, H., & Deng, S.-H. (2019). Hyperparameter optimization for machine learning models based on bayesian optimizationb. *Journal of Electronic Science and Technology*, *17*(1), 26–40. https://doi.org/https://doi.org/10.11989/JEST.1674-862X.80904120

Zaruba, F., & Benini, L. (2019). The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *27*(11), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114