# Learning Semantic Representations to Verify Hardware Designs

**Anonymous Author(s)**
Affiliation
Address
email

## Abstract

We introduce Design2Vec, a neural representation learning approach to learn semantic abstractions of hardware designs at the Register Transfer Level (RTL). The key idea of our approach is to design a Graph Convolution based neural architecture that embeds RTL syntax and semantics. We train the architecture on the task of predicting coverage points in the design given some input test stimulus. We then present an approach to use the learnt RTL representation to automatically generate new inputs for unseen coverage locations in the design. Our experimental results demonstrate that Design2Vec outperforms several baseline approaches that do not incorporate the RTL semantics. Our approach can generate instantaneous coverage predictions compared to nightly simulation times. Moreover, our tests generated using Design2Vec result in coverage of design points that are difficult to cover for design verification experts using the current manual approaches for test generation.

## 1 Introduction

Verification is the process of checking if a system implementation follows the specification correctly. Hardware design verification checks if a register transfer level (RTL) design implements the architectural specification correctly. Verification is widely considered the most serious bottleneck in the hardware design cycle [] with verification requiring 75-80% of the time, compute and human resources during the design phase [].

Chip design and system-on-a-chip (SoC) design and manufacturing in our society have progressed at a breakneck pace. Every added feature in a new generation of hardware like high performance, low power, high battery life, small size, form factor, heterogeneous integration etc. add to the scale and complexity of the design. This makes their verification a massive, perennial challenge with verification technologies trying to catch up with designs of unprecedented complexity.

While there are many decades of research in automatic hardware verification (**??**) involving exhaustive state space analysis by formal verification, static analysis, hybrid static and dynamic analysis etc., their applicability in practice is limited by scale. In contemporary hardware industry, design verification sacrifices automation and completeness for a "best effort" process based on design simulations **??**. However, this cannot circumvent the complexity of verification and ends up effectively costing multiple (expert) engineer years per hardware design. Different practical verification challenges include (a) generating high quality tests (b) ensuring coverage closure, *i.e.* 100% coverage across all metrics (c) debugging and diagnosing bugs (d) generating good properties/assertions among others. Each of these verification tasks is based on knowledge of the RTL design.

We introduce Design2Vec, a learning based approach to learn semantic representation of hardware RTL designs. The idea behind Design2Vec is to use a design-aware graph neural network architecture

Submitted to 34th Conference on Neural Information Processing Systems (NeurIPS 2020). Do not distribute.

to encode the RTL design, and apply these learnt embeddings to different downstream verification tasks ((a)-(d) above). The graph neural network encodes the following two key representations the RTL design: (i) a control data flow graph (CDFG) and (ii) the source code language. While the source code component incorporates the syntactic structure of the RTL design, the control flow graph incorporates the control semantics of the design. We train Design2Vec using tests that trigger execution paths in the RTL design, and branches covered by those tests. rising: maybe interesting to mention something about why this is a different problem than just program testing, and why program models aren't sufficient

We demonstrate that the Design2Vec architecture is able to succesfully generalize across different designs using (i) IBEX [], a RISCV design and (ii) the TPU [] an industrial scale ML chip accelerator used to power Google's datacenters. We demonstrate that the representations learnt by Design2Vec can be used for two different downstream verification tasks - coverage prediction and test parameter generation. rising: TODO: add a bit more information about key results, and some baselines that we compare with

### 1.1 Motivation and Current Approaches

rising: This sub-section looked a bit out of place here. One possible idea for reorganization could be we can move the list of contribution above, and move this 1.1 section to a separate Section 2: Problem Formulation The current approaches for testing such RTL designs follow a constrained random verification [] approach. This involves a complex program called the testbench, through which some inputs can be constrained to legal values, while others can be randomized, to widely sample the input space. Developing the testbench, generating input constraints, as well as parameterizing the testbench into high level tests are all manual and resource intensive tasks. In state of practice, verification engineers write a high level parameterized test, which triggers some constraints in the testbench, and randomizes other inputs. The testbench output is a Boolean level stimulus that is applied to the primary inputs of the RTL design. The simulation takes around a night, and is called nightly regression. Bug reports and coverage are assessed at the end of a nightly regression. This process is iterated upon until there are 0 bugs and 100% of the design cover points are covered. Simulator calls are costly and need to be minimized.

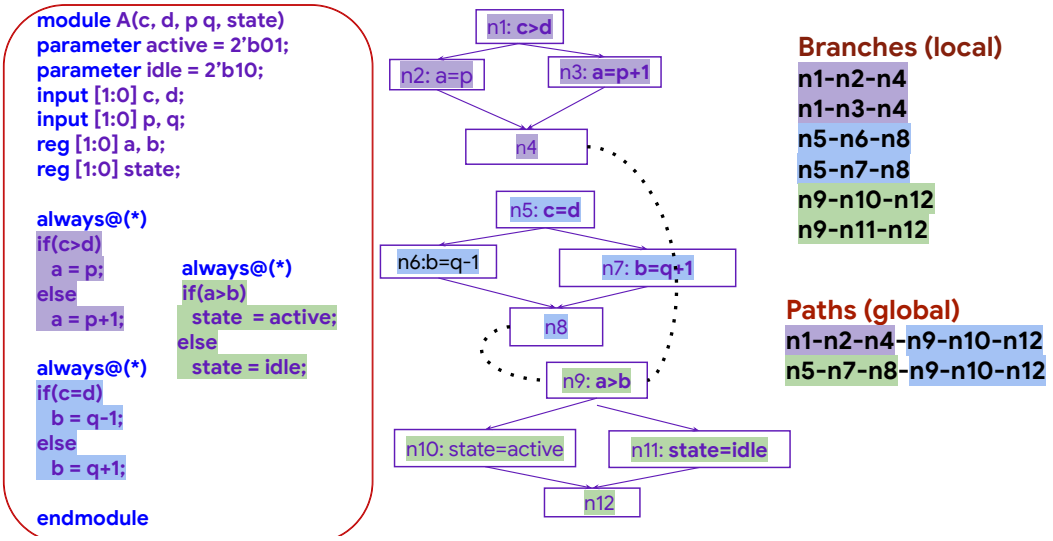
We select two verification tasks for Design2Vec: (i) coverage prediction, or predicting which cover points in the design would be covered by a given test. Such a predictive model can serve as a proxy simulator, and an engineer can query it to estimate coverage instantaneously, instead of waiting for a night of simulation. (ii) test parameter generation, or automating tests towards faster coverage closure. Design2Vec performs two functions in these tasks. First, it summarizes and generates relevant abstractions of the RTL design. These abstractions include static control data flow and source code information, as well as dynamic information about the execution paths from training data. Second, it appproximates the functionality of the testbench, and bypasses it by directly correlating the high level test parameters to design coverage.

Other verification tasks like debugging and property generation are also based on design knowledge and understanding. Since those tasks are directly related to the design, and do not involve the testbench, they are more straightforward than test generation tasks. We expect the Design2Vec architecture and modeling to generalize similarly to those tasks as well.

### 1.2 Contributions

Our contributions in this work are as follows. We demonstrate that

- It is possible to learn RTL designs using deep neural networks. This has implications for verification, synthesis, analysis and understanding of hardware designs. rising: this contribution might be a bit too general, and also following contributions capture it a bit, so maybe we can remove this contribution
- We present Design2Vec, a model that encodes the syntactic and semantic structure of an RTL design using a GCN based architecture. We show that Design2Vec can generalize across different RTL designs and different verification tasks.
- Design2Vec generates high quality tests that can cover rare and hard to cover points in the design, outperforming blackbox models by orders of magnitude.




.cm

Figure 1: Snippet of RTL source code module and corresponding CDFG.rising: could the global ordering between the three always block also be shown in RTL syntax?

- The Design2Vec based method can scale to generate directed tests for the industrial strength TPU design and integrate easily into the standard testing workflows. This shows that our modeling technique is scalable and can significantly impact productivity and efficiency of future generations of hardware designs.

## 2 Design2Vec: Representation learning of design abstractions

### 2.1 Modeling of RTL designs

Hardware is designed using RTL source code as non-terminating, continuous, reactive "programs". It is structured as a modular hierarchy, with multiple concurrent blocks executing within a module. Figure 1 shows an example snippet of Verilog RTL source code. It has three concurrent (always) blocks that execute when any of the conditions within the sensitivity list (*) are triggered. For synchronous hardware, an external clock signal triggers execution of each concurrent block per *clock cycle*. Within an always block, statements are executed in sequence every cycle. Each statement in the current cycle executes with variable values from the previous cycle. There is an implicit loop back from the end to the beginning of each block, to simulate the non-terminating nature of hardware.

We build a control data flow graph (CDFG) for Verilog RTL with heterogeneous node types and edge types. Nodes correspond to statements in code. Edges corresponds to data and control flow. Dotted lines show dependencies between concurrently executing blocks. Branches denote localized control edge sequences, while paths are global control edge sequences from root node to a given node.

Branch coverage of a test refers to branches executed by that test. Each branch is referred to as a *cover point*. A cover point over the RTL CDFG is computed as a sequence of nodes corresponding to a local branch. Two local branch cover points lie along the same global path if they have ancestor/descendant relationship.

A *test* is a set of high level parameters, each of which can be Boolean, integer or categorical. Each parameterized test triggers some constraining and randomization of inputs through the testbench. The output of the testbench is Boolean input stimuli applied to the RTL design under test. These stimuli execute different design paths. When a path is executed, all the local branches along that global path are said to be covered.

## 2.2 Architecture

We use a GCN-based supervised model architecture for Design2Vec training. We train our model on the coverage prediction task, which can act as a pretraining for downstream verification tasks. Our training data involves tests (set of parameters and constraints) and coverage vector of all code cover points with covered/not covered. For evaluation, we find if the coverage of a cover point that has never been covered/seen in the training data (always 0) can be predicted accurately for a given test. This is the quantity that is being predicted as is_hit(cover_point, test). The architecture for this is shown in Figure 3.

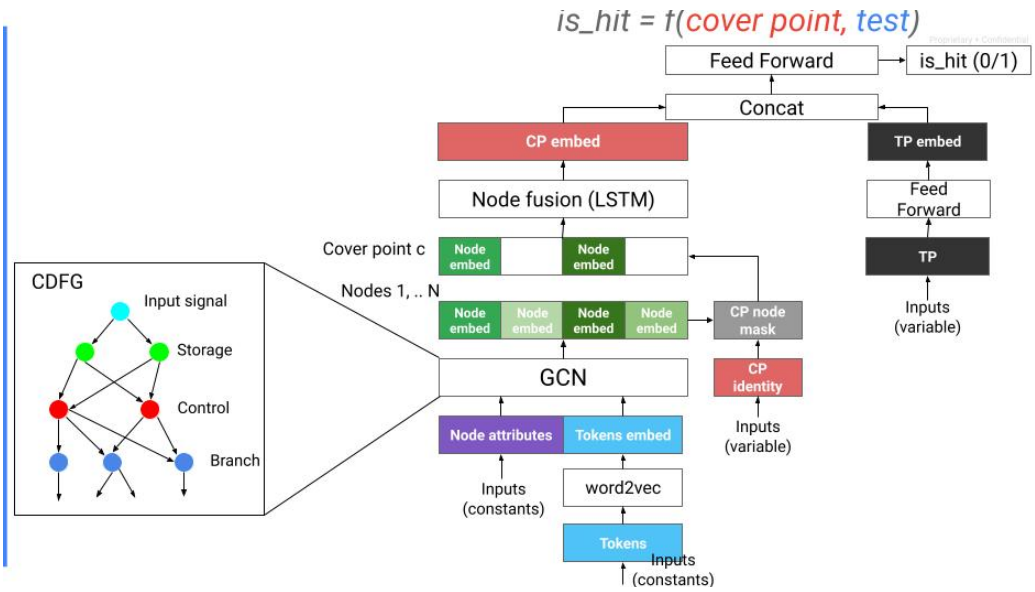



Figure 2: Design2Vec architecture.

**Test parameter embedding**: We learn an embedding for test parameters by passing them through a feed forward MLP layer. The test parameters are either integer or enum data types.

rising: TODO: Mathematical formulation of MLPs, GCNs, token embeddings etc.

**Cover point embedding:** We use a graph convolution network (GCN) over the Verilog CDFG. Each node has attributes or features like node identifier, node type, and fanin and fanout. Each node corresponds to a statement in the RTL design source code. Every statement can be considered as a sequence of tokens. An embedding is learnt over all such token sequences using the Word2Vec, as well as an LSTM based model. This is also passed as a node attribute. Embeddings are learnt for each node of the GCN. A cover point is a sequence of nodes. The node embeddings are passed along with the cover point identity to form the cover point node mask. The cover point mask is applied to the node embeddings, resulting in the sequence of nodes corresponding to each cover point. Since these sequences are of varying length, an LSTM layer is used to learn the embeddings of the cover points.

**Feed forward layer**: Test parameter embeddings and cover point embeddings are concatenated and provided to a feed forward MLP layer to predict is_hit() for each cover point. We also found that our model learns from the cover points that are from the local branch neighbors, than the ancestor nodes along the path from the root to a cover point. A reason for this is that GCNs are typically not proficient at learning long path information. In the case of a control flow, tracing and learning long paths is relevant. We use shortcut edges, where we add edges along every kth node along a path, to reinforce the relationship of long paths. We use a variation of the GCN, called the GNN-MLP [] to propagate edge information. We also used gating layers and residual layers in the GCN.

### 2.3 IPA-GNN based

We need the model to be able to predict which branch will be taken, based on the values of the branch conditional variable for a test. Since this requires tracking dynamic values, we have tried a recently proposed architecture (IPA-GNN) that can track control flow.

#### 2.3.1 Test generation

For test generation, we directly differentiate the coverage prediction model with respect to each test parameter. We use gradient based search to generate test parameters, and query the Design2Vec model for coverage prediction. The feedback is used to fine tune the gradients. The tests with the highest predicted coverage are run through VCS to validate the result against the ground truth. rising: We can also add a small algorithm here on how to use the previous supervised model to generate tests

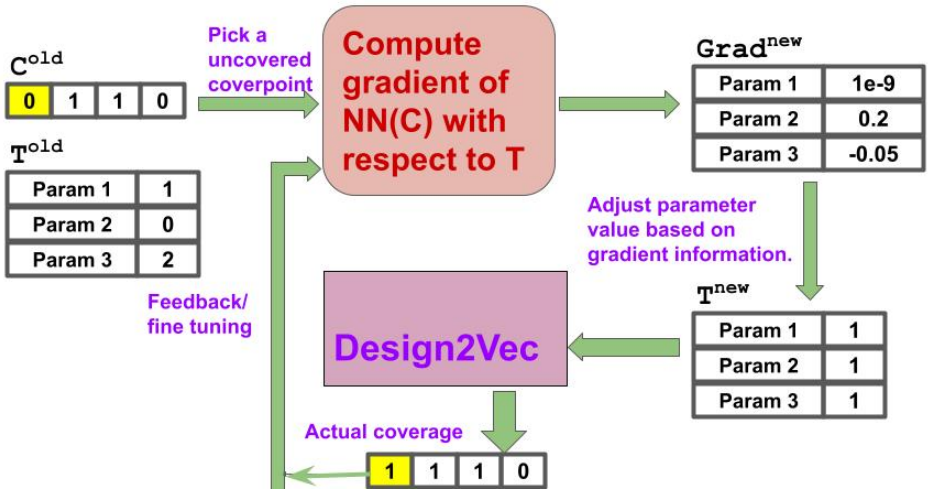



Figure 3: Design2Vec in generative loop.

## 3 Experimental results

## 4 Coverage prediction

#### 4.0.1 Comparisons with blackbox MLP

Our experimental results demonstrate that representation learning of RTL through Design2Vec adds value to the learning task. We demonstrate this value by comparison to a feed forward MLP that has identity vectors for each of the cover points. In the MLP, each coverpoint is represented by an identity function. We refer to this as the "blackbox model" (BB). The Design2Vec model with the graph, but without the language modeling is called the "greybox model" (GB) and the model with both graph and language modeling is called the "whitebox model" (WB). Our experiments compare learning metrics (prediction accuracy) between these three models. For evaluation, we hide a percentage of cover points from the training set, and use these cover points unseen by the model, as part of the evaluation set. The cover points are selected for hiding based on random sampling, as well as by hiding every "kth" (10th, 5th or 2nd) cover point. We evaluate prediction accuracy for hidden cover points when i) all the test parameters are shown during training **??** and ii) when the test parameters corresponding to the hidden cover points are unknown to the model during training **??**.

| | 10% of coverpoints hidden | | 20% of coverpoints hidden | | 50% of coverpoints hidden | |
|---|---|---|---|---|---|---|
| Hiding strategy | BB | GB/WB | BB | GB/WB | BB | GB/WB |
| Every branch | 92.8/56.8 | 82.6/89.2 | 93.3/55.5 | 84.3/82.4 | 90.1/57.1 | 87.7/78.7 |
| Random sampling | 88.3/60.4 | 85.2/84.1 | 92.1/59.1 | 85.9/81.9 | 92.3/54.5 | 85.3/77.3 |

Table 1: Coverage prediction per seen test for IBEX RTL design by MLP and by Design2Vec. Training accuracy/Validation accuracy

| | 10% of coverpoints hidden | | 20% of coverpoints hidden | | 50% of coverpoints hidden | |
|---|---|---|---|---|---|---|
| Hiding strategy | BB | GB/WB | BB | GB/WB | BB | GB/WB |
| Every branch | 92.8/58.8 | 83.5/86.1 | 95.4/48.8 | 83.6/79.2 | 93.3/53.4 | 87.9/77.5 |
| Random sampling | 93.2/54.8 | 84.1/84.4 | 93.4/56.8 | 88.2/78.9 | 92.5/57.5 | 89.0/78.5 |

Table 2: Coverage prediction per unseen test for IBEX RTL design by MLP and by Design2Vec. Training accuracy/Validation accuracy

# 5 Test parameter generation

The Design2Vec model in generative mode can be used for directed testing. Given a cover point, or multiple cover points, it generates a test to cover the point(s). We validated the top 10 recommendations made by the model through the VCS Verilog simulator []. The number of test recommendations that were found to truly cover the cover points as a fraction of the top 10 recommendations is recorded as the success rate. Table 5 and Table 6 show the result of querying a randomly selected subset of cover points through Design2Vec for the IBEX and the TPU respectively. This is the results of querying a single cover point at a time. We refer to a cover point as rare if its cover probability is less than 0.3 in the randomly generated training dataset. The coverage report from VCS indicates a "table index" that roughly measures the locality of a coverpoint in the RTL. If two coverpoints belong to the same "case" statement or "if" statement, they will correspond to the same table. Hardware design structure typically has very large switch statements (example: to decode different instructions). From Table 5, rare coverpoint 9 is covered by gradient search with a high success rate, while random search is not able to cover it with any of the recommendations. Cover point 667, also rare, is covered by neither approach. Model confidence is also of a much lower range for this cover point.

In Table 6, rare cover point 8089 is covered, while 9496 is not. However, as Table 7 demonstrates, querying multiple cover points at the same time from the same table generates highly successful tests. This implies that GCN learning isighly effective in the local neighborhood of the design. The rare cover point 9496 is covered along with other cover points in table 3791. The test that could not cover this point as well as the top 3 tests that could cover it are presented in Table 8. The difference between the first test that does not cover the point and the remaining ones that cover the point is that the model has learnt that the weight of one of the parameters () should be set to 0. This amounts to knowing not to generate any vreg instruction sequences to cover this point.

### 5.0.1 Comparisons with Vizier

<TODO> Difference between directed testing of Design2Vec and cumulative testing of Vizier Not an equal comparison due to fundamentally different capabilities Comparing cumulative coverage for the sake of a baseline Practically, Design2Vec will be used for constraint generation (order of hundreds of constraints) vs few parameters. Not conducive to Bayesian optimization

| | 10% of coverpoints hidden | | 20% of coverpoints hidden | | 50% of coverpoints hidden | |
|---|---|---|---|---|---|---|
| Hiding strategy | BB | GB/WB | BB | GB/WB | BB | GB/WB |
| Every branch | 91.9/58.8 | 85.5/87.3 | 91.7/47.1 | 85.7/88.2 | 91.8/32.5 | 86.5/87.7 |
| Random sampling | 91.7/29.6 | 85.8/89.2 | 91.8/54.3 | 85.9/88.3 | 91.2/29.1 | 85.6/88.2 |

Table 3: Coverage prediction per seen test for indstrial scale TPU RTL design by MLP and by Design2Vec. Training accuracy/Validation accuracy

| | 10% of coverpoints hidden | | 20% of coverpoints hidden | | 50% of coverpoints hidden | |
|---|---|---|---|---|---|---|
| Hiding strategy | BB | GB/WB | BB | GB/WB | BB | GB/WB |
| Every branch | 91.8/33.3 | 86.6/85.7 | 91.9/56.8 | 86.2/86.9 | 91.9/42.7 | 86.8/86.0 |
| Random sampling | 91.9/33.2 | 82.4/86.8 | 92.0/42.0 | 86.9/86.3 | 91.3/41.5 | 85.6/86.8 |

Table 4: Coverage prediction per unseen test for industrial scale TPU RTL design by MLP and by Design2Vec. Training accuracy/Validation accuracy

| Coverpoint | Location | Cover probability | Random search | Confidence | Gradient search | Confidence |
|---|---|---|---|---|---|---|
| 9 | 2 | 0.232 | 0/10 | [0.9997, 0.9999] | 9/10 | [0.9997, 0.9999] |
| 170 | 45 | 0.599 | 10/10 | [0.9988, 0.9995] | 10/10 | [0.9988, 0.9995] |
| 356 | 77 | 0.491 | 4/10 | [0.997, 0.999] | 2/10 | [0.997, 0.999] |
| 383 | 80 | 0.452 | 10/10 | [0.9044, 0.9863] | - | - |
| 534 | 83 | 0.564 | 9/10 | [0.9997, 0.9998] | 7/10 | [0.9997, 0.9998] |
| 667 | 110 | 0.191 | 0/10 | [0.520, 0.533] | 0/10 | [0.520, 0.533] |

Table 5: Test generation per unseen cover point for IBEX using random search and gradient search approaches. Success rate refers to the number of recommendations among the top 10 generated tests that were validated by the simulator as covered. Confidence is model confidence range over the interval. Cover probability is calculated from a randomly generated dataset.

# 6 Ablation experiments for architecture

# 7 Related work and discussion

## 7.1 State of the art

In hardware design input stimulus (test) generation, most techniques in academic literature focus on generating Boolean level stimuli at the inputs (ports) of the RTL design. These "tests" refer to sequences of Boolean inputs applied to the RTL design. In practice, these Boolean sequences can be millions of cycles long. Hence, the practical approach is to generate tests at a higher level of abstraction, with a testbench program that can manipulate inputs through high level symbolic constraints and parameters. These constraints and parameters are human understandable, and each such high level "test" corresponds to millions of cycles of Boolean input stimuli that run through a nightly regression. Our test generation task in this work generates the high level parameterized tests. As such, it is solving the problem at a different level of abstraction than the state of the art techniques. By virtue of operating at a higher abstraction level, it is orders of magnitude more scalable than the state-of-the-art Boolean input stimulus generation techniques. This is demonstrated by our ability to scale to industrial scale designs like TPU, a scale hitherto unseen in test generation literature.

| Coverpoint | Location | Cover probability | Success rate |
|---|---|---|---|
| 666 | 285 | 0.735 | 10/10 |
| 5216 | 2096 | 0.511 | 7/10 |
| 8089 | 3273 | 0.124 | 10/10 |
| 8795 | 3475 | 0.561 | 10/10 |
| 9496 | 3791 | 0.1083 | 0/10 |
| 30525 | 12566 | 0.7193 | 0/10 |

Table 6: Test generation per unseen cover point for TPU using random search and gradient search approaches. Success rate refers to the number of recommendations among the top 10 generated tests that were validated by the simulator as covered. Confidence is model confidence range over the interval. Cover probability pertains to randomly generated dataset for training.

| IBEX | | | | TPU | | | |
|---|---|---|---|---|---|---|---|
| Coverpoint | Location | Cover probability | Success rate | Coverpoint | Locality | Cover probability | Success rate |
| 67 | 11 | 0.26 | 10/10 | **9496** | 3791 | 0.108 | 10/10 |
| 77 | 11 | 0.25 | 10/10 | 9526 | 3791 | 0.75 | 10/10 |
| 80 | 11 | 0.256 | 10/10 | 9530 | 3791 | 0.74 | 10/10 |

Table 7: Multiple cover points that are in the same locality (same table number). Model success rate is better when genertaing tests for many cover points in the local neighborhood. Rare point 9496 is not covered during single point query but is covered during multi cover point test generation.

| Cover points | Top test recommendations |
|---|---|
| 9496 | scalar_load_store_seq_weight= 0.0 +sreg_init_seq_weight = 0.0, +scalar_alu_seq_weigh = 15.0 +vector_alu_seq_weight = 0.0, +vme |
| 9496, 9526, 9530 | scalar_load_store_seq_weight= 0.0 +sreg_init_seq_weight = 5.0, +scalar_alu_seq_weigh = 15.0 +vector_alu_seq_weight = 0.0, +vme |
| 9496, 9526, 9530 | scalar_load_store_seq_weight= 0.0 +sreg_init_seq_weight = 0.0, +scalar_alu_seq_weigh = 15.0 +vector_alu_seq_weight = 0.0, +vme |
| 9496, 9526, 9530 | scalar_load_store_seq_weight= 5.0 +sreg_init_seq_weight = 0.0, +scalar_alu_seq_weigh = 15.0 +vector_alu_seq_weight = 0.0, +vme |

Table 8: Sample tests generated by Design2Vec for a rare cover point in TPU. First test is a single coverpoint query.Other 3 tests cover all the queried points. Model learns that +vreg_init_seq_weight should be set to 0, implying that no vreg instruction sequences should be generated in order to cover this point.

Despite the difference in abstraction levels, we discuss the state-of-the-art approaches with respect to our approach from an algorithmic standpoint. [1] There are two kinds of approaches in practical hardware design verification- directed testing [] and random testing[]. Directed testing refers to targeted testing of specific functionality or structural constructs in RTL. Random testing refers to random perturbation of inputs, with the purpose of covering the design space maximally. Static analysis based approaches for test generation[] based on traversing the state transition graph at a logic gate level, or the RTL design graph [] have inherent scalability limitations. Approaches that combine static and dynamic analysis like HYBRO[] and concolic testing[] in RTL design test generation are more scalable than pure static analysis. These are directed testing approaches that rely on SMT/SAT solvers, which are exhaustive analysis engines, also limited by scale. Some other approaches that combine ML techniques like random forests, decision trees etc. with static analysis and formal verification [] are able to scale to larger designs. However, they require manual feature engineering and handcrafted algorithms.

In contrast, our deep learning based approach is able to

### 7.2 Abstractions using deep learning

*Traditional abstractions in verification*: Abstraction, or strategic information hiding, has been used to scale design verification for many decades[]. Some of them are property specific (counterexample guided abstraction refinement)[], design specific (cone-of-influence or program slicing based)[], data abstractions (word level abstractions)[], language based(abstract interpretation)[], execution or structure based (partial order and symmetry)[]. Irrespective of the abstraction technique, firstly, it has been very difficult to scale verification using any of these abstractions. Secondly, the level of abstraction is tricky to construct. If the abstraction is too coarse, it can result in an overapproximation, where properties cannot be falsified. Similarly for an underapproximation, properties cannot be proven true. Exact abstractions are hard to construct and do not provide much benefit over the original design. Thirdly, many abstractions do not hold across properties of the same design, or across designs. Under and over approximations here refer to states, given the discrete nature of hardware systems.

*Abstractions created by deep learning*: Deep learning, in general, generates complex approximation functions that summarize the examples in the training data, which it then uses for prediction or inference tasks. Deep representation learning is able to leverage underlying structure in software programs [], compilers [] etc.

In a hardware design, representation learning can be used with RTL CDFGs and the source code tokens as representations. This also pertains to the *design state space*. A state is a unique valuation of all the variables in the design. The state space of the design comprises all the unique states that

[1] We tried to obtain these tools to validate the scalability and scope arguments made here. Most of the tools are not in the open source. For those in the open source [], they are not able to accept designs in the format and language that is widely used as an industry standard (System Verilog). As such we were not able to run our benchmarks through these tools.

the design can reach and the legal transitions to reach every state. The GCN then, traverses the state space of the design and learns a notion of similarity and difference between states that is based on the path of traversal. If two nodes of the directed CDFG are on the same path, that corresponds to those two states of the design being reachable from each other. The high performance accuracy achieved by the GCN demonstrates that the proximity/neighborhood of states is effective in predicting which input is most effective.

Deep learning generates on-the-fly, task-specific design abstractions that can generalize across designs, as well as across different tests, properties and/or bugs. The training data helps create an approximation or abstraction of the design that pertains to the task it is trained for. These task-guided abstractions can generalize to other tasks as well. For example, an embedding learnt by a pre-training task of coverage prediction could be used for constraint generation or test generation. With representation learning, the structure of the design itself can be thought of as a "dormant", static representation, and the concrete examples of a given task, "activating" this database, creating a task specific abstraction.

A reason why the pretrained embeddings are transferable between design verification tasks, is due to the learning of the design state space. While the test generation task correlates inputs to the different states of the design, it also learns the connections between design states.

An advantage of deep learning is that adding new features to the representations is fairly straightforward. This eliminates the need for feature engineering to determine which features are important. This process is also much easier than custom crafting of abstractions.

# 8 Conclusion