A STUDY ON MACHINE LEARNING-BASED HARDWARE BUG LOCALIZATION

A Thesis

by

SANJAY RAJASHEKAR

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2020

Major Subject: Computer Engineering

ABSTRACT

Simulation-based verification is a very essential technique in ensuring the correct functionality of any digital integrated circuit design before it goes on silicon. One of the major challenges of running simulation-based verification on complex designs is the tradeoff between simulation time and the time taken for failure localization or to root cause. This is because the simulation run times could be very high when there are many checkers used per cycle of execution. However, when lesser checkers are turned on, the amount of time for manual debug increases because, after failure, the verification engineer has to manually analyze the failure and turn on the more granular checkers individually and re-simulate; or invest lots of time, memory and resources to manually go through the simulation cycles dumps before the failure which is not good given the current complexity of designs.

Machine learning has emerged to be a popular technique to construct mathematical models that can understand the expected patterns from a given dataset. To address the aforementioned trade-off problem, an idea is investigated to use the failing signatures from fewer active high-level checkers during simulation to train a machine learning model to predict the location of the bug in the design. This information would in turn be used to turn on relevant checkers in the design before re-simulation. Other methods to analyze the signals in design after failure to predict bug location were also studied. This idea is implemented and tested on a MIPS processor with total of ~ 700 bugs injected in 15 different units to distinguish them with good accuracy.

I am dedicating this page to my mother for her immense support & encouragement throughout my journey at Texas A&M University.

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Jiang Hu, for constantly guiding me with attention to detail, which has pushed me many times to rethink the problem under consideration from many different angles. The many discussions I have had with him have proven to be instrumental in completing my research work.

I would also like to thank my committee co-chair, Dr. Aakash Tyagi for being a constant motivational influence for my research. He has always been very patient and supportive in discussing and critically evaluating some of the ideas and the limitations associated with them.

I would also like to thank Prof. Mike Quinn, for introducing me to the world of hardware verification via two great courses at A&M and for being very helpful in the discussions of evaluating this work.

I would like to thank Dr. Gwan Choi for being a part of my thesis committee and providing constructive feedback on my thesis. I also convey my thanks to my friends and the department faculty and staff for making my time at Texas A&M University a great experience.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Prof. Jiang Hu [advisor] and Prof. Gwan Choi of the department of ECE and Prof. Aakash Tyagi [co-advisor] of the department of computer science.

The MIPS CPU design used for this thesis work is leveraged from the lab syllabus of the course ECEN 651 microprogrammed control of digital systems offered at Texas A&M.

The rest of the work conducted for the thesis was independently completed by the student.

Funding Sources

Graduate study was partly supported through a scholarship from the ECE department at Texas A&M University.

NOMENCLATURE

ASIC – Application Specific Integrated Circuit

RTL – Register Transfer Level

HDL – Hardware Description Language

EDA – Electronic Design Automation

VLSI – Very Large Scale ICs

HAS – High-level Architecture Specification

ATPG – Automatic Test Pattern Generation

GDS – Graphic Data System

DRC – Design Rule Check

LVS – Layout vs Schematic

LEC – Logical Equivalence Check

DV – Design Verification

DUT – Design Under Test

DUV – Design Under Verification

SBV – Simulation-based Verification

FV – Formal Verification

ISA – Instruction Set Architecture

GPR – General Purpose Registers

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In the 21st century our lives have been dominated by the use of electronic devices like smartphones, laptops, PCs, smartwatches and other IoT & health equipment, etc. To enable the functionality of every single of these devices, we have at its heart a small piece of silicon designed by the efforts of hundreds of engineers. The process of designing an ASIC (application specific integrated circuit) is tedious and involves several important steps, evolving from a concept to specification to final tape-outs. Given that the final product is typically quite small (measured in mm/cm), this long journey is filled with many engineering challenges which makes it interesting.

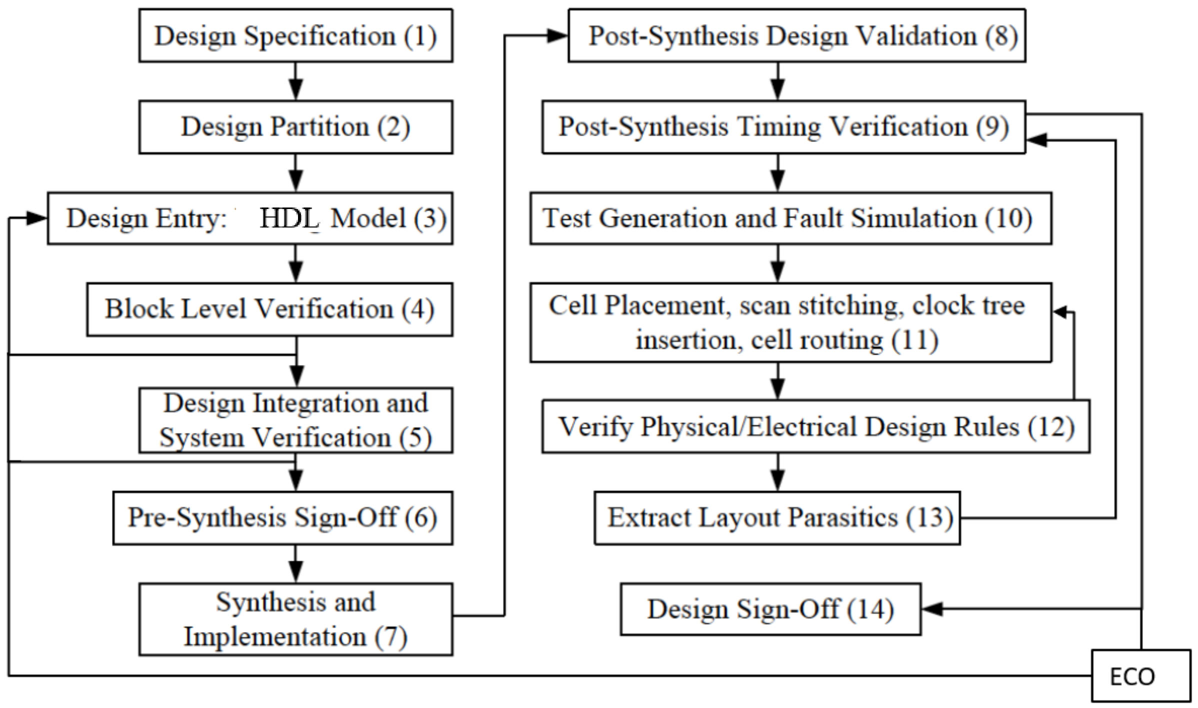

Design Specification (1)
Design Partition (2)
Design Entry: HDL Model (3)
Block Level Verification (4)
Design Integration and System Verification (5)
Pre-Synthesis Sign-Off (6)
Synthesis and Implementation (7)
Post-Synthesis Design Validation (8)
Post-Synthesis Timing Verification (9)
Test Generation and Fault Simulation (10)
Cell Placement, scan stitching, clock tree insertion, cell routing (11)
Verify Physical/Electrical Design Rules (12)
Extract Layout Parasitics (13)
Design Sign-Off (14)
ECO

Figure 1: Digital IC Design Cycle

The design flow of an ASIC is a well-tested and proven IC design process which includes various steps like conceptualization of design, efficient verification, optimization of chip performance/power/area, logical/physical implementation, and extends to post silicon validation.

Figure 1 gives a brief overview of the ASIC design flow. Digital circuits are designed using the top-down approach. There are two main parts of the design.

a) Front end design

b) Back end design

1.1 Front End Design

Based on the custom product requirements of the chip, architecture & design experts come together to put forth a design specification required for their product. This specification of the architecture & design is captured as an abstract specification document, also called high level architecture specification (HAS). After understanding the design specifications, the engineer's task is to partition the entire ASIC into multiple hierarchical modules (functional blocks), while keeping in mind ASIC's best performance, resource allocation and technical feasibility in terms of power, area, time & budgets. Once all the modules are designed in the architectural document, the engineers tend to discuss the partitioning ASIC design by reusing IPs from previous projects or designing them from scratch as needed. The HAS specifications are then translated to HDL code by the RTL designers.

Functional verification verifies the functionality and logical behavior of the circuit by performing simulation on a design entry level. This is an important stage where the design team and verification team come into the cycle where they verify RTL code using test-benches. This is done on different hierarchical levels from block to the entire system on chip. After this step, there are some DFT insertions either by scan or ATPG methods. Then, the HDL code is synthesized into a netlist of logic gates which is done by EDA tools. There are some preliminary timing verifications done in the front-end phase.

1.2 Back End Design

This process starts with the floor-planning and partitioning. In back-end design, this is the first step in transformation towards GDSII from the RTL design. It is the process of physically placing different blocks in the chip. It includes, design portioning, block placement, power optimization and efficient pin placement. Clock tree synthesis is a process of designing the clock paths to ensure that the specification requirements of power, timing and area are met. It is a challenge to provide the clock connection to the clock pin of a sequential element within the constraints of time and area and low power consumption. The next step is routing all the connections within the chip. Following this, a series of physical verification steps including LEC, DRC and LVS are run using EDA tools. Final timing checks are done based on extracted parasitic values and the chip is ready for tape-out.

1.3 Design Verification

Design verification (DV), also sometimes referred to as functional verification or logic verification, is a process to verify thoroughly that the intent of the design under test (DUT) matches with the architectural specifications. It is the one of the biggest factors influencing the trade-off among the triple constraints:

- Timing/Schedule of chip design completion.
- Costs incurred to complete the chip design.
- Quality of design after testing.

Fewer revisions through the fabrication process means lower costs and faster time to market. Only companies who get it right in fewer revisions will survive the competition. This is becoming increasingly challenging due to the enormous rise in complexity of the designs. Figure 2 gives a good depiction of the general trend of how the cost of fixing bugs can accumulate very quickly over time.

A bug found early (during simulation) has low cost of fixing it. The same bug found during on-chip testing has relatively higher cost, since it requires more isolation time and debug time and that needs readjusting of schedule. Finding a bug during system level validation, where the chip is combined with other designs and tested together as a complete system, requires much higher bug fixing costs. The worst possibility of sensing a bug in customer's environment can damage the reputation of the company apart from costing millions of US Dollars. For these reasons, design verification deserves and often receives the most attention for efficiency improvement measures.

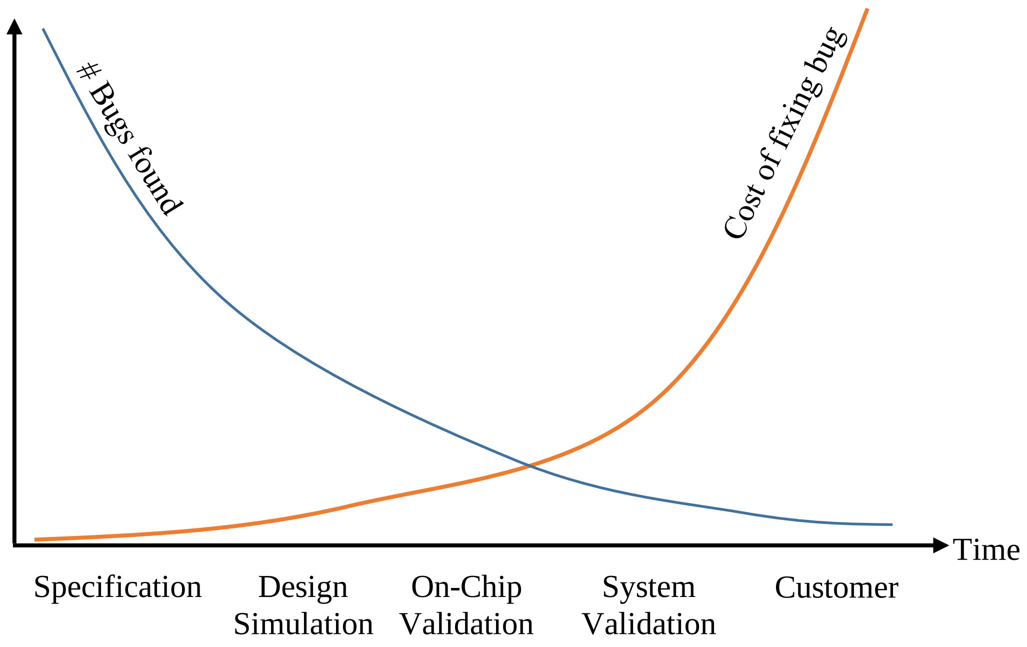


Figure 2: Trend of # Bugs & Cost w.r.t Time

A typical SoC chip would contain many different individual IPs with several million flipflops. Verifying this enormous state-space takes a lot of effort and intelligent planning. To ensure the correctness of a design, hardware verification is performed majorly in two ways.

a) Simulation-based verification (SBV)

b) Formal verification (FV)

In simulation-based verification, the verification engineer must carefully analyze the DUT specifications and come-up with a set of testcases(stimulus) that could efficiently verify the DUT. To verify the correctness, we make use of an error-free high-level model of the same DUT. The chosen stimulus is simulated in both the DUT and the error-free model. The output values are compared to verify the correct functionality of the DUT.

However, in formal verification, these are handled differently. Formal verification is the way to prove or disprove the correctness of a certain specification or property in the design mathematically. Formal verification exhaustively checks for correctness of the property in the design, with no concern of input stimulus. Hence, the DV engineer need not put effort to create specialized stimulus as in case of SBV. However, the property or specification needs to be entered accurately. FV has a few downsides. Primarily, the testing time can be very high when the design is complex. Secondly, owing to complicated designs, some properties could be very difficult to be captured accurately. Hence, it becomes difficult to map the complex features of the design to formal mathematical property. Due to these drawbacks of FV and relatively practical scalability of SBV, SBV is one of the most widely used verification technique in the industry.

Hence, this research is focused on the simulation-based verification approach. As chip complexity continues to grow, simulation-based functional verification is becoming a bottleneck in the overall chip design cycle. In this research, some applications of machine learning are applied to reduce the overall debug time and improve debug quality, potentially reducing verification costs and time to market.

2 BACKGROUND

2.1 Simulation Based Verification

As discussed in the previous section, simulation-based verification (SBV) is the most widely used technique for verifying RTL designs. SBV is based on the principle of applying a set of stimuli to exercise the design and then checking that the design behaves as expected. Due to the complexity of current designs in industry, the ratio of human resources for verification to design is around 2-3 :1. This just explains how important verification is in the design process of a chip. For an SBV to be efficient and successful, a few main aspects need to be understood.

2.1.1 Stimulus Generation

Stimulus generation mainly refers to the process of generating a set of desirable input sequences that could exercise the design under verification (DUV) effectively. In other words, it is a phase where relevant input stimuli vectors are generated and applied to the design to verify the implementation against the specification. The hierarchy for different types of stimulus is as follows:

a) Transaction: Input with the highest granularity of stimulus generation. Each transaction is an atomic operation on the design, for e.g., adding 2 registers and storing result into another register.

b) Sequence: It can be defined as a set of serial transactions at a data port, for e.g., running a set of instructions on a single core of a CPU.

c) Test: Input with a set of sequences that are generally executed parallelly to create more complex stimuli involving more than one data ports.

For larger designs, it becomes extremely complicated to manually write transactions or sequences. Generally, a constrained random generation of stimuli is employed. It is important that tests are designed or constrained such that they exercise all the design's properties & specifications in the least test count.

2.1.2 Checker Generation

Checking is a process to monitor the activity of the design and indicate any erroneous behavior. It is responsible for creating failing conditions, which would guide in finding bugs in the design. An ideal checker would point directly to the bug, making debugging easy. One way to perform checking is to code checkers called assertions within the RTL designs. An assertion is an 'if' statement with an error condition that throws error when the condition is not met. System Verilog provides constructs to write temporal and concurrent assertions. Such checking is done online, while the simulation is running.

Another popular way for checking is using reference model or a scoreboard. Scoreboard/reference model is a golden model (generally designed in simple and non-HDL language) that maintains the correct functionality of one or more design features from the specification. The scoreboard is provided the same set of input vectors as the DUT. The output of the golden model and the DUT output are compared for any inconsistency. Unlike assertions, here, the checking is done offline, at the end of the simulation. One of the advantages of this method over assertions is that having many assertions in the RTL design tends to slow down the simulation.

2.1.3 Coverage

Coverage is a metric that is utilized to gauge the progress & assess the effectiveness of the stimuli & checkers used for verification. Hence, it is very crucial in determining when the design is robust enough for tape-out. The coverage results provide the guidance to make critical decisions on the next steps in the verification cycle. Coverage driven verification, is a popular methodology that is built around coverage metrics which are used as primary gauge to manage verification. Coverage is represented by a set of models, both simple and complex, that capture the design intent. There are mainly 2 types of coverage.

a) Code coverage: It measures the extent to which the code in design is exercised during the simulation.
b) Functional coverage: It measures the extent to which all the important features and functionality of any design are verified, and hence is of interest in most of the cases.

Coverage closure is the point of time at which nearly 100% of the design intent has been verified (or covered).

2.1.4 Debugging

There are two main challenges of simulation-based verification. The first one is to find the right set of stimuli to exercise the design efficiently. The second one is to efficiently debug, localize, and root cause the bug after there is a failure seen in the simulation runs. Although debugging strategies aren’t necessarily included in verification plan, it is a very important step which affects the schedules of the hardware design. In many cases, verification engineers end up needing to depend on logic traces of all the relevant signals for many cycles. This is naïve and very inefficient. A common methodology to better aid debugging includes simulating the design under verification in conjunction with a golden model, while checking the RTL & golden model output values at regular points in time(intervals). If the golden model used is high-level only, then there are fewer compare operations and thus the penalty on RTL simulation time is reduced. However, this makes the task of debug much harder as there could be more than one unchecked block where the error could have occurred. The manifestation could also have happened at a different design unit or cycle depending on the frequency of checking. On the other hand, if the golden model is very granular, then most of the internal signals are checked at every step which would improve the debug time but greatly penalize the simulation time due to the additional checks every cycle. There is a clear trade-off between simulation time and debug time, given the ever-increasing complexity of functionalities expected on the ASIC. Achieving this tradeoff is the focus of my thesis work.

2.2 MIPS Design

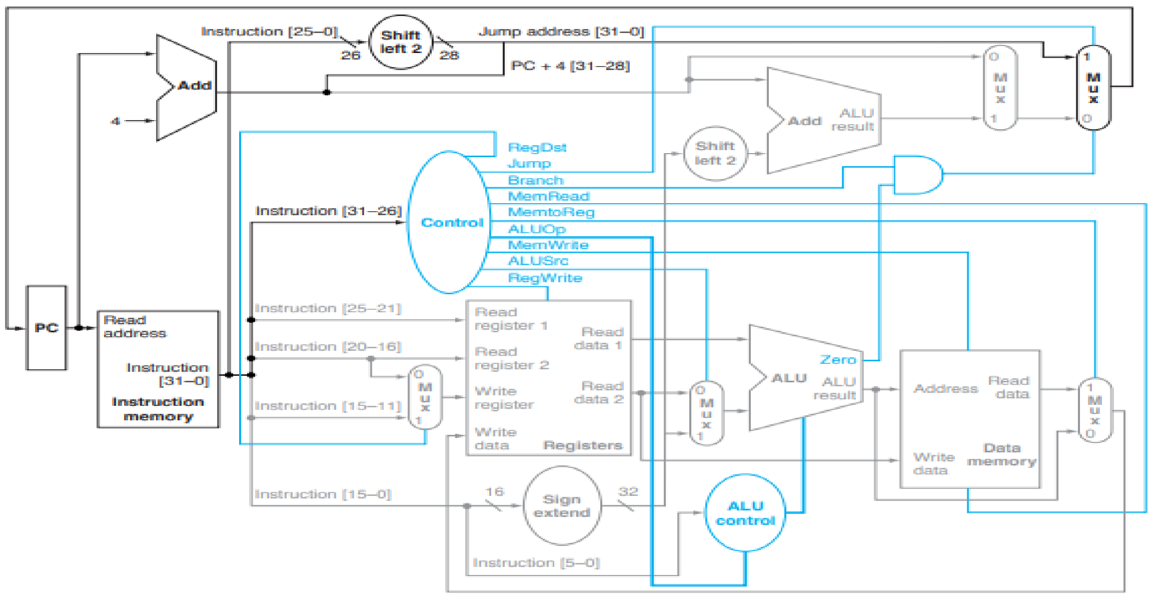


Figure 3: 32-bit MIPS Processor Block Diagram.
(Reprinted from [11] ECEN 651 Lab Texas A&M University)

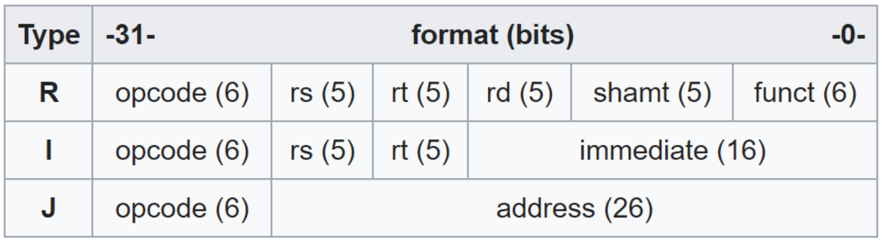
The design used in the current thesis is based on the popular MIPS architecture. This section introduces MIPS microarchitecture & ISA. MIPS (Microprocessor without Interlocked Pipelined Stages) is a reduced instruction set computer (RISC) instruction set architecture (ISA) developed by MIPS Computer Systems, now MIPS Technologies, based in the United States [10]. There are multiple versions of MIPS since its introduction in 1985.

The one used in this thesis is the 32-bit only version named MIPS I. MIPS is a load/store architecture (also known as a register-register architecture); except for the load/store instructions used to access memory, all instructions operate on the registers. The following sections discuss more on the MIPS Architecture.

2.2.1 Registers

MIPS has total of thirty-two 32-bit general-purpose registers (GPR). Register 0 is hardwired to zero and writes to it are discarded. The program counter has 32 bits. The two low-order bits always contain zero since MIPS I instructions are 32 bits long and are aligned to their natural word boundaries.

2.2.2 Instruction Formats



Type	-31-			format (bits)		-0-
R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
I	opcode (6)	rs (5)	rt (5)	immediate (16)		
J	opcode (6)	address (26)				

Figure 4: 32-bit MIPS I Instruction Format.
(Reprinted from [10] MIPS Wikipedia)

Instructions are divided into three types: R, I and J. Every instruction starts with a 6-bit opcode. In addition to the opcode, R-type instructions specify three registers, a shift amount field, and a function field. I-type instructions specify two registers and a 16-bit immediate value. J-type instructions follow the opcode with a 26-bit jump target.

2.2.3 Load and Stores

MIPS I has instructions that load and store 8-bit bytes, 16-bit halfwords, and 32-bit words. Only one addressing mode is supported: base + displacement. Since MIPS I is a 32-bit architecture, loading quantities fewer than 32 bits requires the data to be either signed or zero-extended to 32 bits. The load instructions suffixed by "unsigned" perform zero extension; otherwise sign extension is performed. Load instructions source the base from the contents of a (GPR) general-purpose register (rs) and write the result to another GPR (rt). Store instructions source the base from the contents of a GPR (rs) and the store data from another GPR (rt). All load and store instructions compute the memory address by summing the base with the sign-extended 16-bit immediate. MIPS I requires all memory accesses to be aligned to their natural word boundaries.

2.2.4 ALU

MIPS I has instructions to perform addition and subtraction. These instructions source their operands from two GPRs (rs and rt) and write the result to a third GPR (rd). Alternatively, addition can source one of the operands from a 16-bit immediate (which is sign-extended to 32 bits).

MIPS I has instructions to perform bitwise logical AND, OR, XOR, and NOR. These instructions source their operands from two GPRs and write the result to a third GPR. The AND, OR, and XOR instructions can alternatively source one of the operands from a 16-bit immediate (which is zero-extended to 32 bits).

The “Set on relation” instructions (e.g., SLT) write one or zero to the destination register if the specified relation is true or false. These instructions source their operands from two GPRs or one GPR and a 16-bit immediate (which is sign-extended to 32 bits). The result is written to a third GPR. By default, the operands are interpreted as signed integers. The variants of those instructions, that are suffixed with "unsigned", interpret the operands as unsigned integers (even those that source an operand from the sign-extended 16-bit immediate).

The load upper immediate (LUI) instruction copies the 16-bit immediate into the high-order 16 bits of a GPR. It is used in conjunction with the Or Immediate instruction to load a 32-bit immediate into a register.

MIPS I has instructions to perform left and right logical shifts and right arithmetic shifts. The operand is obtained from a GPR (rt), and the result is written to another GPR (rd). The shift distance is obtained from either a GPR (rs) or a 5-bit "shift amount" (the "shamt" field).

2.2.5 Control Instructions

There are two types of control instructions in MIPS, branch & jump. The branch instructions compare the contents of a GPR (rs) against zero or another GPR (rt) as signed integers and branch if the specified condition is true. Control is transferred to the address computed by shifting the 16-bit offset left by two bits, sign-extending the 18-bit result, and adding the 32-bit sign-extended result to the sum of the program counter (instruction address) and 4.

Jumps are unconditional branch instructions which are resolved after instruction decode rather than after the ALU as compared to branch. The final address is calculated by shifting the

26-bit “address” left by two bits and concatenating the 28-bit result with the four high-order bits of PC+4.

2.3 Machine Learning

Machine learning (ML) can be defined as a scientific study of statistical models and algorithms that is used to perform specific tasks by computer systems, without explicitly using instructions and relying on inferences from different patterns instead. It is seen as a subset of artificial intelligence. Machine learning algorithms build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to perform the task. Machine learning algorithms have been used widely in many applications such as image recognition, face recognition, disease diagnosis, etc., where it is infeasible or difficult for a conventional algorithm to effectively complete the task.

2.3.1 Machine Learning Terminologies

Here are some of the common terms used in machine learning[9].

a) Dataset: For any machine learning application, data is the most important part. All machine learning systems would require the user to either get the data (for e.g., from online open sources) or collect it with custom designed experiments. The term dataset refers to all data that is used for either building or testing the ML model. Datasets are divided into 3 groups:

   Training data: The data used during training phase. The ML model learns to detect patterns from the data and determines which features are more relevant during prediction.

   Validation data: The data is used for comparing different models in order to determine the best ones and for tuning model parameters.

Test data: Test data is completely unseen data, used for inference purposes for gauging the performance of ML algorithm.

b) Input Attribute: Features extracted from training dataset and used for output prediction.

c) Target Label: Output values/labels to be predicted. This is also called Class.

d) ML Algorithm: Program that provides a model for prediction suitable for the training Dataset.

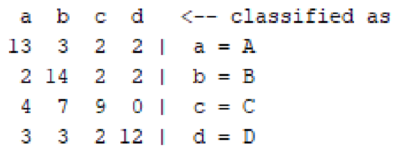
e) ML model: The artifact created by applying the algorithm on training dataset.

f) Labelled data: It consists of a set of attributes coupled with the respective target label data, an example would include all the labelled cats or dogs' images in a folder, all the prices of the house based on size, etc.

g) Classification: Separating the data into discrete groups having unique characteristics, e.g., dog or cat, 1 or 0.

h) Regression: Estimating the most probable values or relationship among variables, e.g., estimation of the price of the house based on size.

i) Confusion Matrix: It is a common form of evaluation of a ML algorithm by analyzing how many samples of a known class X are classified as X. This analysis is done for all samples of every class in the dataset and expressed in the form of a matrix. Figure 5 shows a simple example where samples of each class(row) are classified among the available classes (columns).



```
  a  b  c  d   <-- classified as
 13  3  2  2 |  a = A
  2 14  2  2 |  b = B
  4  7  9  0 |  c = C
  3  3  2 12 |  d = D
```

Figure 5: An Example Confusion Matrix

There are four major types of machine learning techniques.

a) Supervised learning: The outcome or output for the given training input is known and the machine must be able to map or assign the given input to the output. For example, multiple images of a cat, dog, orange, apple, etc., here the images are labelled. It is fed into the machine for training and the machine must identify the same. Just like a human child is shown a cat and told so, when it sees a completely different cat among others it still identifies it as a cat. The same method is employed here.

b) Unsupervised learning: The outcome or output for the given inputs is unknown, i.e., there is no label for the given input data. Here, the goal for the ML algorithm is to study the relative structure of various data elements and group them into different bins with similar features. The main algorithms include clustering algorithms.

c) Semi-supervised learning: It is in-between that of supervised and unsupervised learning, where the combination is used to produce the desired results and it is the most important in real-world scenarios where all the data available are a combination of labelled and unlabeled data.

d) Reinforced learning: The machine is exposed to an environment where it gets trained by trial and error method. It is trained to make a specific decision. The machine learns from past experience and tries to capture the best possible knowledge to make accurate decisions based on the feedback received. Its practical applications include computer playing board games such as chess and Go, self-driving cars also use this learning.

2.3.2 Why Use ML to Aid Verification Tasks?

Manual debug and root cause of bugs based on human judgment are time consuming and prone to errors. Machine learning has proven to be a very good way to train models to learn relevant information pattern from huge datasets and perform specific tasks. Although machine learning algorithms existed from decades ago, only in the recent years have we been able to provide the necessary processing power to produce the results in a practical time span. This motivated me to incorporate ML to the task of bug localization.

2.3.3 Metrics to Gauge Performance of an ML Algorithm

There are many metrics available to gauge the performance of an ML algorithm. It really depends on the requirement of the application and our dataset to choose the most relevant. Misclassification error (or classification accuracy) alone makes sense when given a uniform distribution of class labels. Also, we need to understand the importance of other metrics like Precision (PRE), Recall (REC) & F1-score. Figure 6 captures the definition of what each metric means. Different metrics assume importance in different applications. Precision is important for applications which require high penalty for false positives, e.g., an automated missile launcher falsely predicting a hostile element could be devastating. When PRE is high, false positives are low. On the other hand, Recall is important for applications which require to highly penalize false negatives, e.g., predicting the presence of a deadly disease like HIV as absent could result in further misdiagnosis with serious consequences. When REC is high, false negatives are low. F1 is a combined score of both PRE & REC. These metrics are based on binary classifications (2 classes). However, they can be easily extended to multiclass classification tasks. Generalizing this to multi-class, while assuming we have a 1 vs all classifier, we can go with either the “macro” or the “micro” average.

In micro averaging, we would calculate the performance, e.g., precision, from the individual true positives, true negatives, false positives, and false negatives of the k-class model. In macro-averaging, we average the performances of each individual class as shown in Figure 6.

		Predicted Class	
		P	*N*
Actual Class	*P*	True Positives (TP)	False Negatives (FN)
	N	False Positives (FP)	True Negatives (TN)

$$Classification\ Accuracy = \frac{TP + TN}{TP + FP + TN + FN}$$

$$PRE = \frac{TP}{TP + FP}$$

$$REC = \frac{TP}{FN + TP}$$

$$F_1 = 2 \cdot \frac{PRE \cdot REC}{PRE + REC}$$

$$PRE_{micro} = \frac{TP_1 + \cdots + TP_k}{TP_1 + \cdots + TP_k + FP_1 + \cdots + FP_k}$$

$$PRE_{macro} = \frac{PRE_1 + \cdots + PRE_k}{k}$$

Figure 6: Definition of Metrics for ML

3 PREVIOUS WORKS

In this section, I discuss the previous works in hardware bug localization or root cause analysis. The work [1] by Park, et al., proposed an automated way to analyze the post-silicon failures offline. The data passing through every design block is stored in trace buffers during program execution. Next, using the design structure, a graph with designs units as nodes is constructed and the behavior of each node is verified using the input and output signals. Links in the graph are traversed back until the failure is found. Although, this provides good root cause, it needs high memory and time to store and backtrack every stimulus that was executed.

Another work [2] by D. Lin, et al., proposed an approach to post-silicon validation and debug using symbolic quick error detection (QED). Here the stimulus is edited on the fly to re-simulate the same operation on a different set of registers and compare the result in both the cases. This process would cause the bug manifestation, if any, to occur much earlier than without QED. Though it helps to get the bug occurrence & manifestation in the same cycle, it increases the total instructions to be tested and needs more register space to trigger the duplicate execution.

Machine learning was used in more recent works [3] like the one by Valeria Bertacco, et al., where they propose a pre-silicon bug localization technique (which was extended from their previous work [4], on Post Silicon bug-diagnosis). They relied on only architectural state (registers, memory & PC) mismatches to extract signatures to train a machine learning model to predict the bug location. They also resync the golden model with DUV state to prevent cascading failures. However, this work doesn’t utilize information from the internal signals that could help improve localization. It also doesn’t provide a realistic error injection methodology since the errors are injected on the already available error-free netlist. It is logically not possible to have error-free netlist, before the design phase is complete, to create ML error models in the first place. This work leaves more room for improvement in pre-silicon design verification when considering the above disadvantages. Other works like [5][6] which weren't a direct application to bug localization were also studied to understand newer techniques to adopt ML into the current problem statement.

4 PROPOSED IDEA AND IMPLEMENTATION

4.1 Preliminary Experiments

In this section, we see how the capabilities of ML algorithms are gauged and understood by running some preliminary experiments. The rudimentary idea is that given a simple design one could use the combinations of input-output value pairs along with some internal signals as features to train a machine learning model. This is depicted in Figure 7. This means that different errors are injected in different parts of the good design to create data points that help the ML algorithm to classify different error cases.

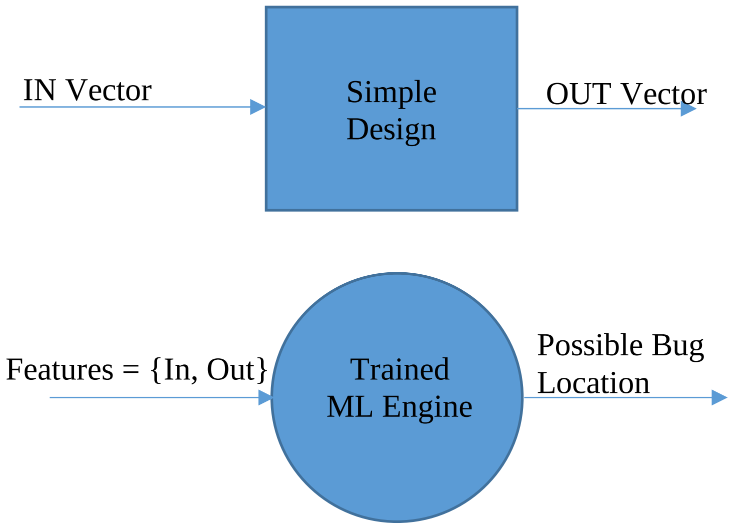


Figure 7: Idea of Bug Localization for a Simple Design

A simple example is considered to analyze this method. A full-adder circuit is considered as DUT as shown in Figure 8. The error injection in the sum & carry out calculation is injected as shown in the table 1. A total of 4 different classes good, error 1,2 & 3 are considered.

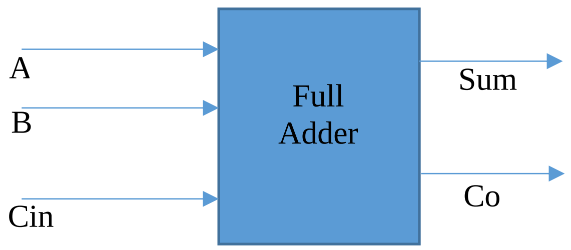


Figure 8: A Full Adder Circuit

Full Adder	**Sum**	**Carry (Co)**
Good	A^B^Cin	(A&B)|(B&Cin)|(Cin&A)
Error Case 1	A&B&Cin	(A&B)|(B&Cin)|(Cin&A)
Error Case 2	A^B^Cin	(A|B)&(B|Cin)&(Cin|A)
Error Case 3	(A&B)|(B&Cin)|(Cin&A)	A^B^Cin

Table 1: A Full Adder Circuit Error Injection

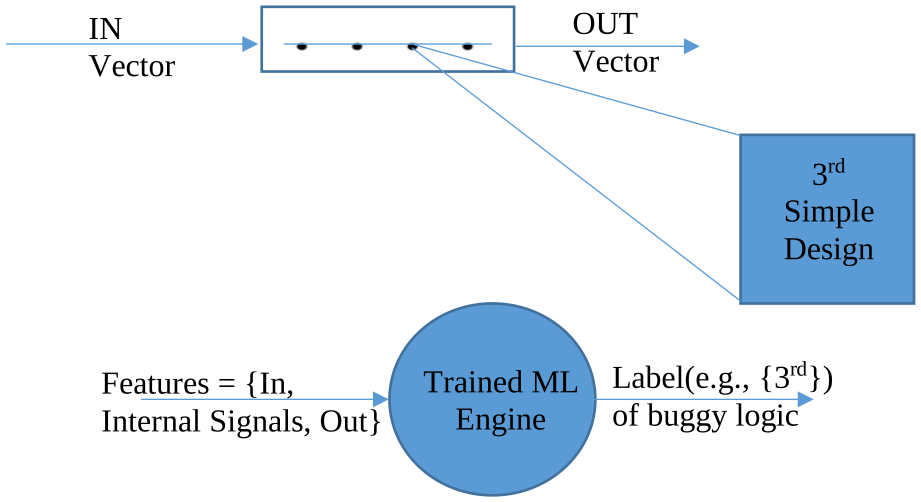
For the dataset, 10 random inputs are generated, and the output is captured for all the 4 cases. The inputs & outputs are used as features. This dataset for the good and the 3 error-cases is trained on many well-known ML engines with 2:1 split for training and testing. The top two performing ones are captured in the Table 2 (left) below.

Classifier	**Accuracy%**
Random Forest	56
Multilayer Perceptron	45

Classifier	**Accuracy%**
Random Forest	55
Multilayer Perceptron	65

Table 2: Exp1 Classification numbers for All 4 Classes (L) and Only Error Classes (R)

The same experiment is repeated to classify only among the 3 error injected cases, since the bug localization needs to happen only among the error cases. Table 2 (right) summarizes the result. The idea of this experiment is to understand how well ML algorithms can pick-up the relationship between the inputs and the outputs given a large number of data points. It is seen that the ML algorithms performed pretty well when compared to the random guessing accuracy (= 1 / total #classes = 25% & 33% for Exp1 Left & Right respectively). This in turn motivated me to extend this idea to a more generic case as shown in Figure 9, where a set of simple designs are combined to form a more complex one.



IN Vector
OUT Vector
3rd Simple Design
Features = {In, Internal Signals, Out}
Trained ML Engine
Label(e.g., {3rd}) of buggy logic

Figure 9: Generic Idea of Bug Localization

The motivation is that the same idea could be used to identify bugs or errors in different parts of an integrated design. For the initial tests a simple system of 4 bit adder using 4 series full adders is considered as shown in Figure 10.

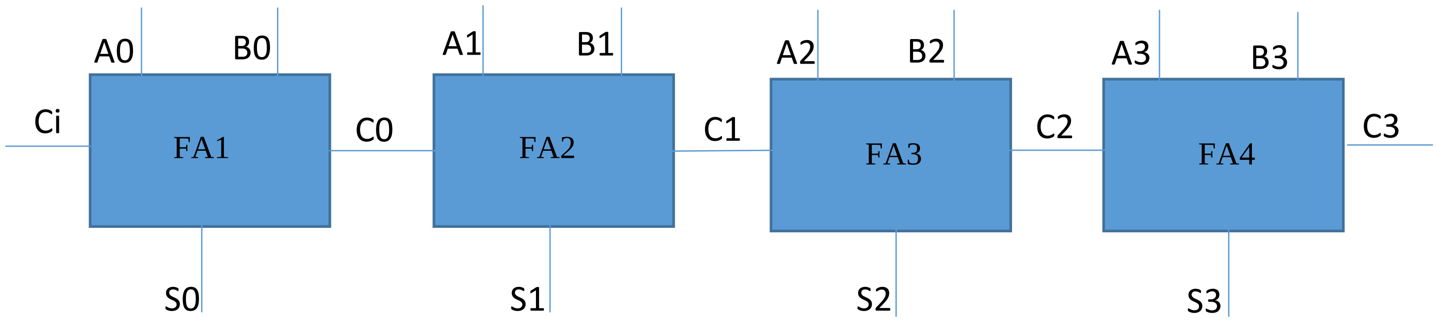


Figure 10: A 4 bit Adder Circuit with 4 Series Full Adders

For the dataset, along with the inputs A[0:3], B[0:3] & Ci and outputs S[0:3],C3, even the intermediate signals C0,C1,C2 are monitored and used as features. The bug used is a switch in functionality of sum and carry operation. This bug is activated in all full adder circuits one at a time. A total of 100 random vectors are simulated for all 5 classes (good + 4 errors). The results for the classification are tabulated in Table 3 (left). The classification is repeated considering only the 4 error cases, where the accuracy increased as shown in Table 3 (right). This could be due to the reduction in total number of classes.

Classifier	**Accuracy%**
Random Forest	35
Multilayer Perceptron	74

Classifier	**Accuracy%**
Random Forest	60
Multilayer Perceptron	84

Table 3: Exp2 Classification numbers for All 5 Classes (L) and Only Error Classes (R)

The previous two experiments show that for a simple system, using an ML algorithm could be beneficial to capture the expected properties of the design. However, this needs to be tested on more complex designs with higher range of input space. Also, there are other useful aspects in the verification environment that could be made use of. For instance, the golden value output from the golden model could be used to better enhance the decision process. This step is incorporated in the next experiment.

The MIPS CPU design as shown in Figure 3 seemed to be an apt choice given its fairly decent complexity. Before using the complete design, the ALU unit is tested first. As per the design ALU unit has 2 input buses A & B of 32 bit width and an output result W of 32 bit width. It also has a 0-indicator output bit which is set when the output value on Bus W is 0. It has a 4 bit

ALUCtrl bus that is used to control the type of operation performed by the ALU unit. Figure 11 shows the block diagram of the ALU Unit.

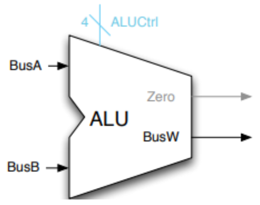


Figure 11: Block Diagram of ALU Unit.
(Reprinted from [11] ECEN 651 Lab Texas A&M University)

Operation	ALU Control Line
AND	0000
OR	0001
ADD	0010
SLL	0011
SRL	0100
SUB	0110
SLT	0111
ADDU	1000
SUBU	1001
XOR	1010
SLTU	1011
NOR	1100
SRA	1101
LUI	1110

Table 4: ALU Control Line Mapping

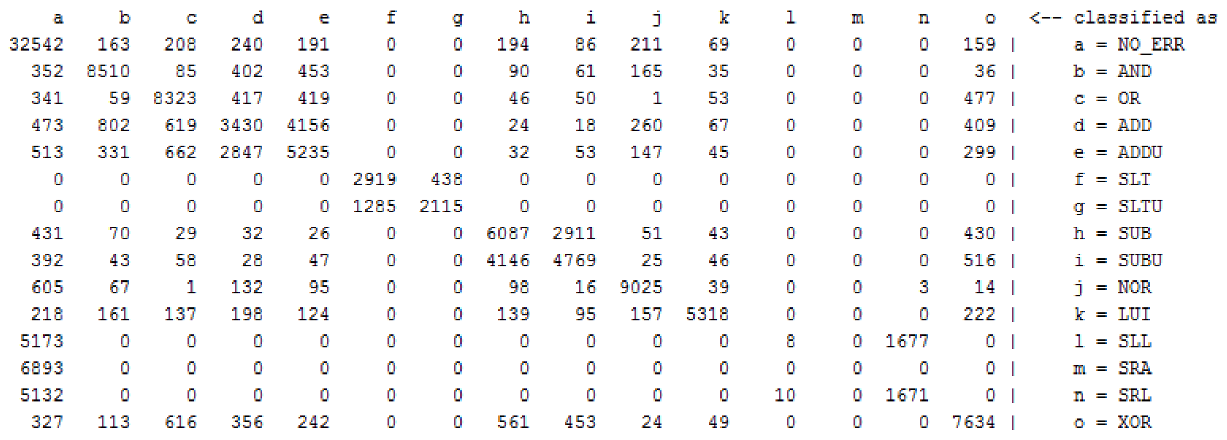
The ALU Unit supports 14 operations as listed in the Table 4. For the error injections the scheme depicted in Table 5 is used.

Original Operation	Error Injection Substituted Operation
AND	OR, ADD, SUB
ADD	SUB, AND, OR
ADDU	SUB, XOR, AND
LUI	LLI
NOR	NAND, OR, XNOR
OR	AND, SUB, ADD
SLL	SRA, SRL
SRA	SLL, SRL
SRL	SLL, SRA
SLT	GT
SLTU	GT
SUB	ADD, OR, XOR
SUBU	ADD, AND, XOR
XOR	AND, OR, XNOR

Table 5: ALU Error Injection Map

The errors injected are kept as close as possible to the logical mistakes that the designer could make. For the dataset, since the number of error cases are much higher than the single good cases, this imbalance is fixed by simulating 10X more inputs than the individual error cases. The inputs A, B & ALUCtrl are generated randomly and simulated for 100K data points for the good (no error) case and 10K data points for each of the error mapping as mentioned in Table 5.

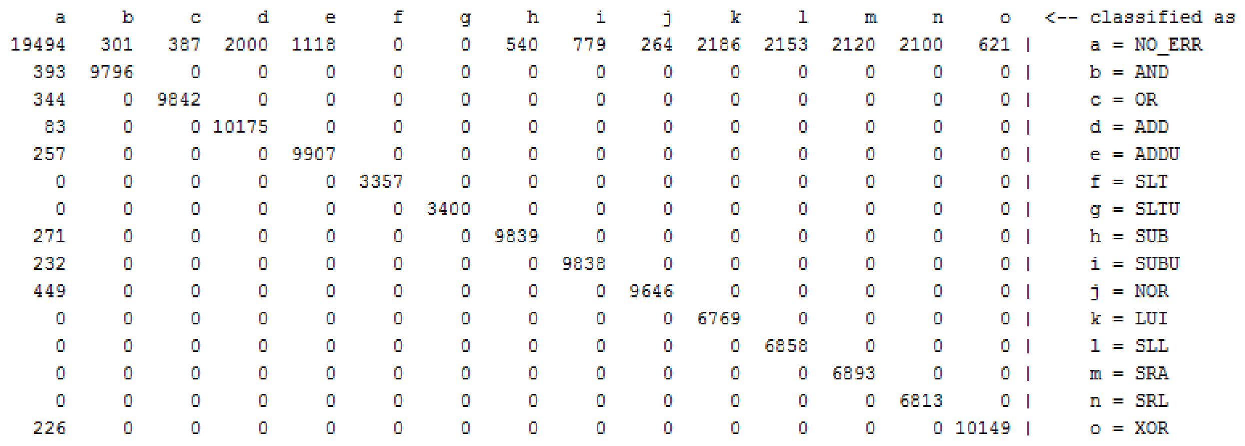
The data element used for training consisted of {Inputs AB, Output W, Expected_Out, Class}. The Expected_Out is obtained from the golden model. Using the ML algorithms with the standard 2:1 split for training and test, J48 decision tree gives the best results upto 65%(as compared to 6.6% when randomly classified). The confusion matrix for the same is captured in Figure 12.



```
     a     b     c     d     e     f     g     h     i     j     k     l     m     n     o   <-- classified as
 32542   163   208   240   191     0     0   194    86   211    69     0     0     0   159 |     a = NO_ERR
   352  8510    85   402   453     0     0    90    61   165    35     0     0     0    36 |     b = AND
   341    59  8323   417   419     0     0    46    50     1    53     0     0     0   477 |     c = OR
   473   802   619  3430  4156     0     0    24    18   260    67     0     0     0   409 |     d = ADD
   513   331   662  2847  5235     0     0    32    53   147    45     0     0     0   299 |     e = ADDU
     0     0     0     0     0  2919   438     0     0     0     0     0     0     0     0 |     f = SLT
     0     0     0     0     0  1285  2115     0     0     0     0     0     0     0     0 |     g = SLTU
   431    70    29    32    26     0     0  6087  2911    51    43     0     0     0   430 |     h = SUB
   392    43    58    28    47     0     0  4146  4769    25    46     0     0     0   516 |     i = SUBU
   605    67     1   132    95     0     0    98    16  9025    39     0     0     3    14 |     j = NOR
   218   161   137   198   124     0     0   139    95   157  5318     0     0     0   222 |     k = LUI
  5173     0     0     0     0     0     0     0     0     0     0     8     0  1677     0 |     l = SLL
  6893     0     0     0     0     0     0     0     0     0     0     0     0     0     0 |     m = SRA
  5132     0     0     0     0     0     0     0     0     0     0    10     0  1671     0 |     n = SRL
   327   113   616   356   242     0     0   561   453    24    49     0     0     0  7634 |     o = XOR
```

Figure 12: Confusion Matrix of ALU Test 1

However, when the ALUCtrl is also used as one of the features, the classification accuracy by J48 decision tree went up to 88%. This can be understood by the fact that ALUCtrl signal contains information that is strongly correlated to the bug location. The confusion matrix for this Test 2 is captured in Figure 13.



```
     a     b     c     d     e     f     g     h     i     j     k     l     m     n     o   <-- classified as
 19494   301   387  2000  1118     0     0   540   779   264  2186  2153  2120  2100   621 |     a = NO_ERR
   393  9796     0     0     0     0     0     0     0     0     0     0     0     0     0 |     b = AND
   344     0  9842     0     0     0     0     0     0     0     0     0     0     0     0 |     c = OR
    83     0     0 10175     0     0     0     0     0     0     0     0     0     0     0 |     d = ADD
   257     0     0     0  9907     0     0     0     0     0     0     0     0     0     0 |     e = ADDU
     0     0     0     0     0  3357     0     0     0     0     0     0     0     0     0 |     f = SLT
     0     0     0     0     0     0  3400     0     0     0     0     0     0     0     0 |     g = SLTU
   271     0     0     0     0     0     0  9839     0     0     0     0     0     0     0 |     h = SUB
   232     0     0     0     0     0     0     0  9838     0     0     0     0     0     0 |     i = SUBU
   449     0     0     0     0     0     0     0     0  9646     0     0     0     0     0 |     j = NOR
     0     0     0     0     0     0     0     0     0     0  6769     0     0     0     0 |     k = LUI
     0     0     0     0     0     0     0     0     0     0     0  6858     0     0     0 |     l = SLL
     0     0     0     0     0     0     0     0     0     0     0     0  6893     0     0 |     m = SRA
     0     0     0     0     0     0     0     0     0     0     0     0     0  6813     0 |     n = SRL
   226     0     0     0     0     0     0     0     0     0     0     0     0     0 10149 |     o = XOR
```

Figure 13: Confusion Matrix of ALU Test 2

At the end of this experiment, it is convincing that the ML engines can handle a reasonably complex design by learning from the inputs & outputs when aided with the comparison data from the golden model.

4.2 Proposed Solution and Setup

Given the complex nature of the problem of bug localization, there is a need to aptly specify the assumptions of the verification environment to understand the challenges and how the proposed solution helps to solve these challenges correctly.

4.2.1 Assumptions of the Environment

The assumptions mentioned here are applicable to any generic digital design that would want to use the techniques proposed in this thesis.

1. All the checkers in the design are implemented.

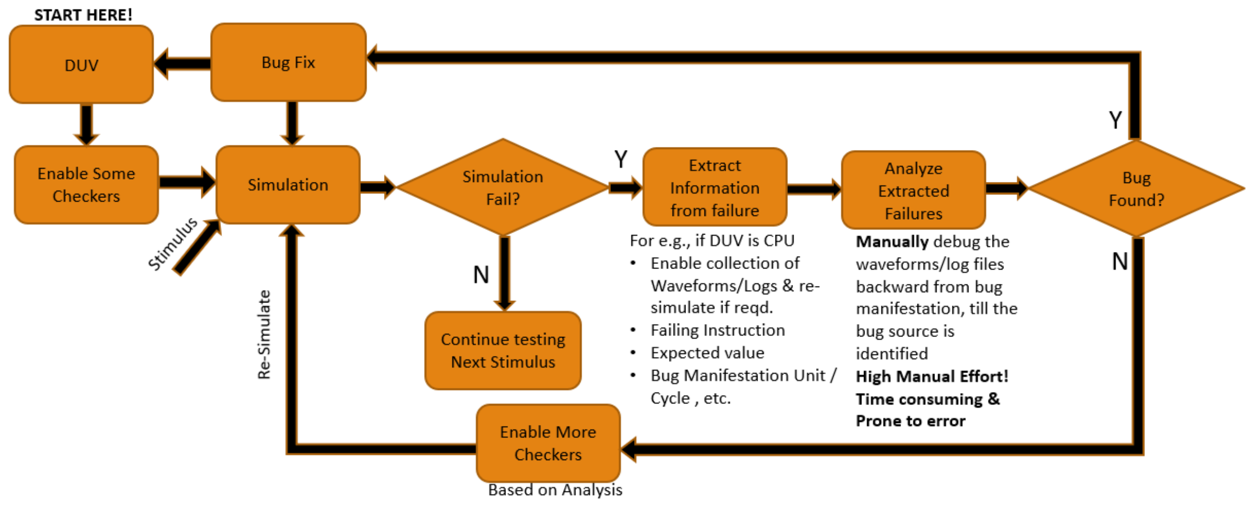
   This means that the checkers at the outputs of various points in the design are ready to be used during simulation tests. This is a practical assumption as most of the important checkers would be needed to run the test vectors and verify the design effectively. This assumption is linked to the next point which helps to understand the challenges involved in such a case of having all the checkers turned on during simulation.

2. During regular simulation only the primary(high-level) checkers are active.

   This assumption explains about the hierarchical importance of checkers. Out of all the checkers in the design, there are some checkers that are of higher importance, for e.g., in a CPU design executing an ALU, the checker that checks the state changes, i.e., register file, memory & PC is much more important that the checker used for sign extend enable. Of course, there are chances that the error in sign extend could cause the failure in the state errors, but in the initial stages of simulation it is important to enable checkers that flag all the major errors that affect the system. Based on this, it would be more important to monitor the state of the CPU. There is also a tradeoff between the simulation speed and the localization accuracy which has already been explained in detail in the Section 2.1.4.

4.2.2 Proposed Solution

In modern designs whose complexities are ever increasing, it proves to be temporally impractical to run the simulation with all the checkers in the design turned on. Hence, during the process of simulation, the key is to enable only the high-level (or primary) checkers to speed up simulation. For the complete simulation, data is collected for mismatches at these high-level checkers when there are failures. If the checker failure causes a state change (i.e., change in the register file, instruction/data memory, PC), the golden model adjusts itself to match the DUV state to prevent further cascading of the same error in the upcoming cycles. This method of resyncing golden model with DUV is inspired by [3]. However, in [3], the assumption of low observability prevents using data from the internal signals. It is important to note that for the pre-silicon design verification, this limitation doesn't exist.


START HERE!
DUV
Bug Fix
Enable Some Checkers
Simulation
Stimulus
Simulation Fail?
Y
N
Extract Information from failure
Analyze Extracted Failures
Bug Found?
Y
N
For e.g., if DUV is CPU
• Enable collection of Waveforms/Logs & re-simulate if reqd.
• Failing Instruction
• Expected value
• Bug Manifestation Unit / Cycle , etc.
Manually debug the waveforms/log files backward from bug manifestation, till the bug source is identified
High Manual Effort! Time consuming & Prone to error
Continue testing Next Stimulus
Re-Simulate
Enable More Checkers
Based on Analysis

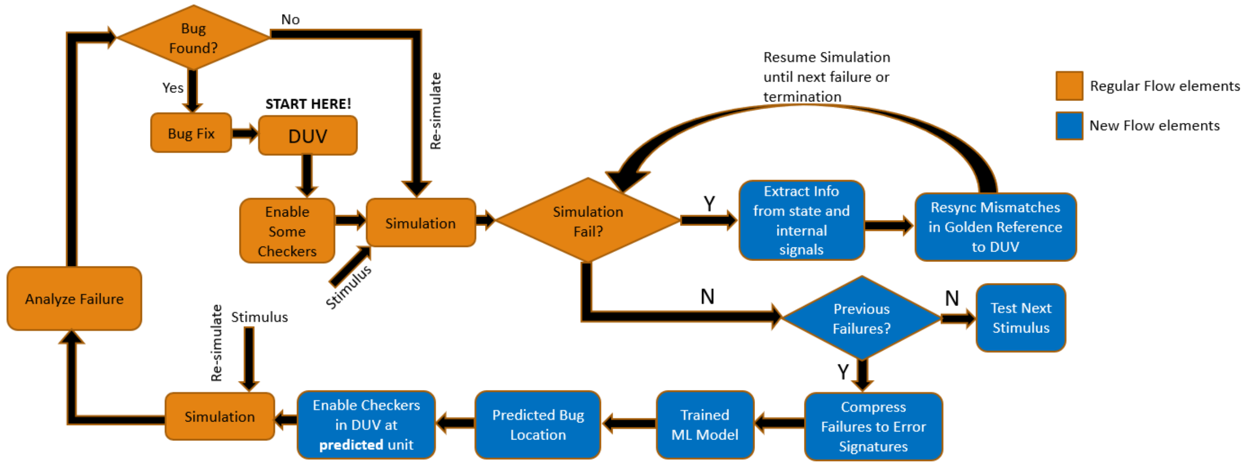


Figure 14: Process of Regular Debug Flow(top) vs Proposed Bug Localization Process(bottom)

The key improvement attributed to the proposed approach is that along with the state level mismatches, few important internal signals would also be monitored. The information from these extra signals gives more insight to provide better features that classify the errors into different units. Figure 14 highlights the process in the form of a comparison between regular debug flow and proposed changes. The other improvement is in the proposed error injection methodology which is discussed in Section 4.2.5.

4.2.3 Setup

For the experimental setup, a 32 bit MIPS processor design is used. There is a separate instruction memory and data memory. The properties of this design are explained in detail in Section 2.2.

There is a total of 16 different regions in the design which are selected for bug injection to create possible error scenarios. These are highlighted in Figure 15. The error injections are made such that only one bug in a single unit is active per simulation run. More about this is explained in the later section.

A total of around 700 bugs are injected in this system. The details of bug types are discussed below.

- Multiplexers (4,9,10,13,16):
    - Invert the mapping of select
    - Mimic output as though select line is stuck at 0
    - Mimic output as though select line is stuck at 1

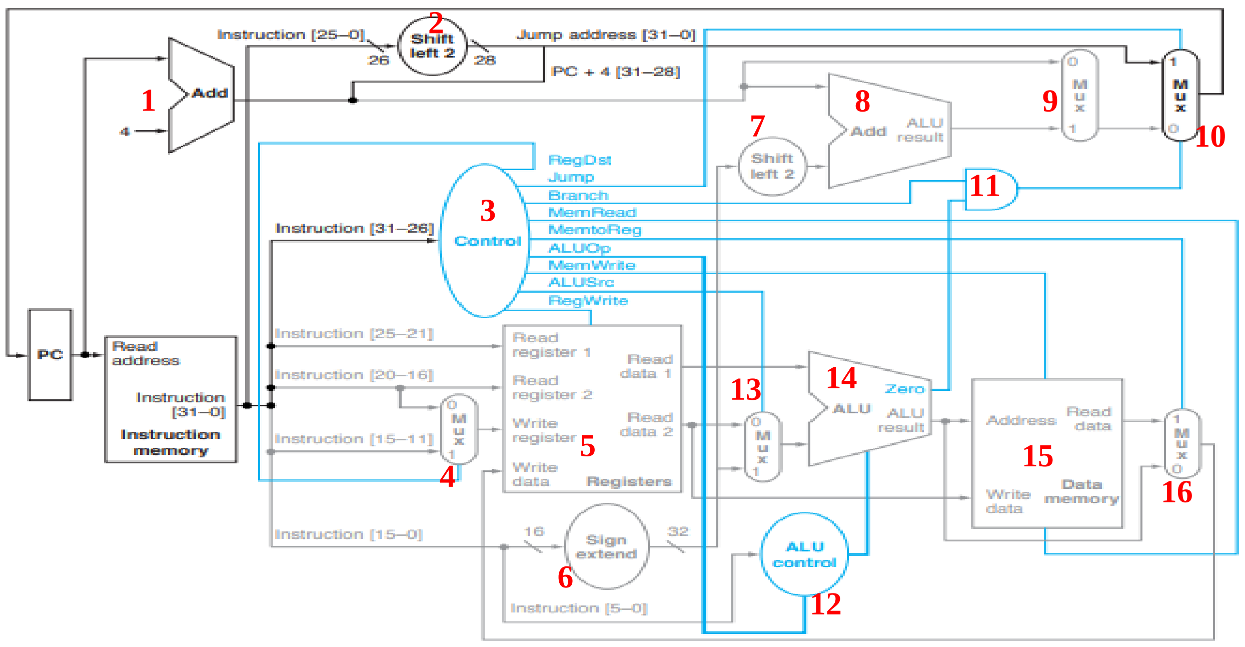


Figure 15: 16 Bug Injected Blocks in the Design.
(Modified from [11] ECEN 651 Lab Texas A&M University)

- Pc + 4 Adder (1):
  - PC+8
  - PC+12
  - PC+16
  - PC
  - PC – 4
- Jump Addr Shift (2):
  - >>2
  - >>4
  - <<4
  - <<1
  - >>1

  - No Shift

- Control Unit (3): The main function of this unit is to map the 6-bit OpCode to 9 single bit control signals (RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, SignExt) and a 4 bit ALUOp. There are a total of 13 OpCodes & 15 ALUOp codes supported. The bugs are injected such that for every OpCode, there is a bit flip for each of the control signals generated for that OpCode, one at a time. Hence, an independent single bit flip in any outputs for a particular OpCode is modelled as 1 bug. Following this process, a total of 299 bugs are injected in the control unit.
- Register File (5):

  Read path

  - Swap of read reg 1 and read reg 2
  - Both read reg 1
  - Both read reg 2
  - Reg 1 reads wrong register(reg1<<1)
  - Reg 1 reads wrong register(reg1>>1)
  - Repeat prev 2 steps for reg 2

  Write path

  - Ignore writes
  - Write to wrong register (WReg>>1)
  - Write to wrong register (WReg<<1)
  - Invert write enable behavior
  - Always write irrespective of write enable
- Sign Ext (6):

    - Skip sign extension
    - Invert sign extension function on sign extend enable
    - Invert sign extension function on MSB bit
    - SignExt by 1 if sign extend enable is 1(irrespective of MSB bit)
    - SignExt by 1 if MSB is 1(irrespective of sign extend enable bit)
    - Always sign extend by 1
- Branch path shift left (7):
    - >>2
    - <<1
    - >>1
    - No Shift
    - >>3
    - <<3
- Branch path addr calculation/addition (8): ADD operation substituted by
    - SUB
    - XOR
    - NOR
    - NAND
    - OR
    - AND
- Branch path branch enable check (11): AND substituted by:
    - OR
    - NOR

    - NAND
    - XOR

- ALU Ctrl (12): This unit maps different ALUOp inputs to 13 different ALUCtrl based on FUNC bits. Based on the same single point mismatch error scheme as discussed in control unit error injection, there are a total of 155 independent errors injected.
- ALU Unit (14): This unit performs 14 different operations on the two 32-bit inputs and returns a 32bit output with a zero-value indicator bit. Following the single point mismatch error scheme, a total of 185 bugs are injected.
- Memory (15):

  Read Path

    - Read from wrong addr (addr<<1)
    - Read from wrong addr (addr>>1)
    - Read enable function inverted
    - Always read without checking on read enable

  Write Path

    - Ignore writes to data memory
    - Write to wrong address in data memory (addr>>1)
    - Write to wrong address in data memory (addr<<1)
    - Allow write to instruction memory
    - Write enable function inverted
    - Always write without checking on write enable

4.2.4 Stimulus

A set of 5 programs with each 10K instructions are used to execute on the design with single bug activated at a time. The ratio of the type of instructions is set to be ALU (40%), Control (20%) & Load/Store (40%). Each of these 5 programs are generated to randomize the following:

- Source registers/Memory addr
- Destination registers/Memory addr
- ALU operation to be performed
- Jump/branch target addresses

4.2.5 Bug Injection Framework

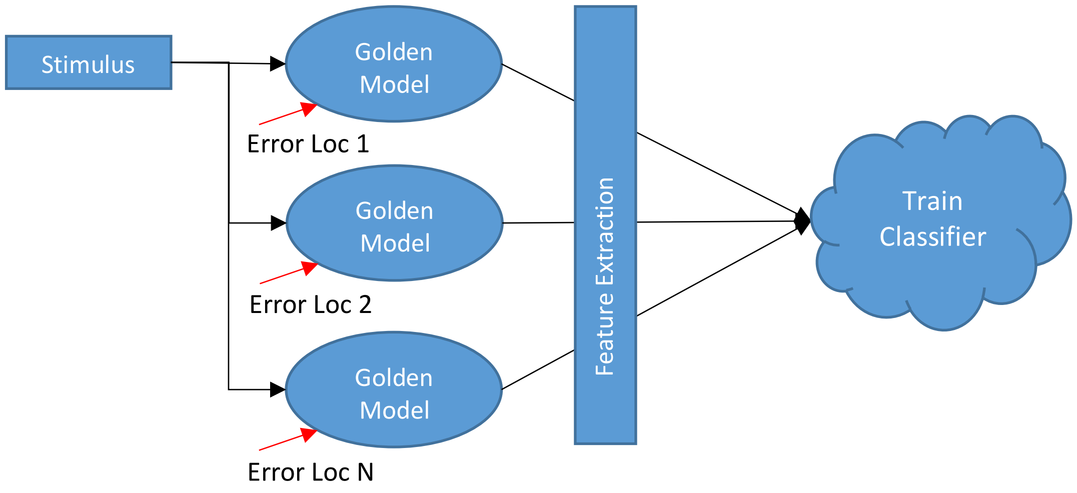


Figure 16: Error Injection Framework

The machine learning model is built on data collected by injecting errors into an error free design. During the early stages of verification, the RTL wouldn't be mature enough and it would be logically wrong to assume that RTL would be error free. However, mature RTL designs from previous generations can be used quite well to an extent if there are minor changes with respect to current design. But for a generic case, a software model of the correct design needs to be prepared by the verification engineer. Since the model would be coded in a simpler language when compared to HDLs, they would be relatively easier to model them since they don't have to deal with the semantic complications like concurrent executions, non-blocking statements, etc. Once this is ready, a variety of bugs can be injected into different units of the model to create a database of bugs. This helps to selectively train and validate the machine learning classifier's performance. This process is captured in Figure 16. For this work, the whole design is modeled in Python. The choice of Python is made because it is easier to collect & manipulate data and also provides very good integration with well-known machine learning packages[7][8].

4.2.6 Error Signatures

The error signatures are extracted from the data mismatches from the golden model and the DUV for all the state-level checkers and internal signal checkers enabled. The mismatches are grouped as follows:

a) Register Value Mismatch

b) Register Address Mismatch

c) Memory Value Mismatch

d) Memory Address Mismatch

e) PC Value Mismatch

f) Internal Signal Checker Output Mismatch

Each of these mismatches are coupled with the type of Instruction that is executed which is one of the following:

a) ALU

b) Load

c) Store

d) Branch/Jump

As already discussed, for every mismatch in the state checkers, we readjust the DUV's state to prevent cascading failures. The simulation termination reasons would give valuable information and hence are noted. They are labelled under one of the following:

a) No failures

b) Data Memory Out of Bounds

c) Instruction Memory Out of Bounds

d) Unknown Instruction Opcode

The difference and Hamming distance for every mismatch are compressed by using mean and standard deviation of those respective distributions, for e.g., consider sample mismatches throughout a single test as shown in Table 6.

No.	Instruction Type	Mismatch type	Model Value	DUV Value
1	ALU	Register Value	64	32
2	ALU	Register Value	128	32
3	ALU	Register Addr	2	4

Table 6: Sample Mismatches

The distribution in this table is converted into a compressed form as shown in Table 7, which uses the mean & standard deviation of the arithmetic difference and the Hamming distance between the mismatched values. These 4 values are used as features to capture any errors resulting from register value mismatches. These are further divided based on which instruction is simulated when the error manifested. Hence, there are 16 such features for the 4 types of instructions as discussed earlier. Along with this, the fraction of number of mismatches that belonged to this category is also captured. When combined, there are a total of 127 features.

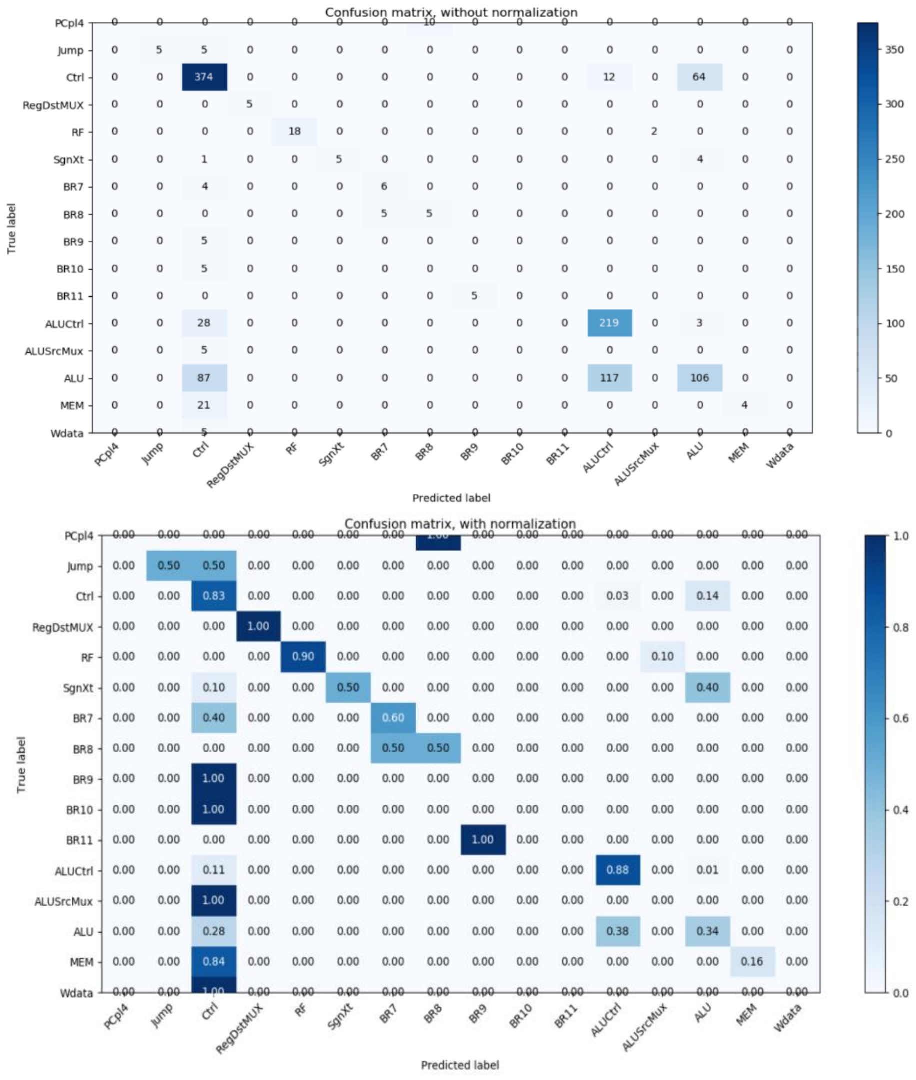
Register Value Features				
Mean		Standard Deviation		Fraction
Difference	Hamming Distance	Difference	Hamming Distance	
64	2	32	0	2/3

Table 7: Compressed Error Signature

5 EXPERIMENTAL RESULTS

This section discusses the results obtained for the proposed solution & provides analysis on how the results change when the choice of the internal signals monitored are changed. Training-testing split of 2:1 is used. Three main classifiers are used, Random Forest (100 trees), XGBoost (100 trees) & Neural Network (hidden layer size=100, #layers=1).

5.1 Testing with BugMD [3] environmental features



	PCpl4	Jump	Ctrl	RegDstMUX	RF	SgnXt	BR7	BR8	BR9	BR10	BR11	ALUCtrl	ALUSrcMux	ALU	MEM	Wdata
PCpl4	0	0	0	0	0	0	0	10	0	0	0	0	0	0	0	0
Jump	0	5	5	0	0	0	0	0	0	0	0	0	0	0	0	0
Ctrl	0	0	374	0	0	0	0	0	0	0	0	12	0	64	0	0
RegDstMUX	0	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0
RF	0	0	0	0	18	0	0	0	0	0	0	0	2	0	0	0
SgnXt	0	0	1	0	0	5	0	0	0	0	0	0	0	4	0	0
BR7	0	0	4	0	0	0	6	0	0	0	0	0	0	0	0	0
BR8	0	0	0	0	0	0	5	5	0	0	0	0	0	0	0	0
BR9	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
BR10	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
BR11	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0
ALUCtrl	0	0	28	0	0	0	0	0	0	0	0	219	0	3	0	0
ALUSrcMux	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0
ALU	0	0	87	0	0	0	0	0	0	0	0	117	0	106	0	0
MEM	0	0	21	0	0	0	0	0	0	0	0	0	0	0	4	0
Wdata	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0

	PCpl4	Jump	Ctrl	RegDstMUX	RF	SgnXt	BR7	BR8	BR9	BR10	BR11	ALUCtrl	ALUSrcMux	ALU	MEM	Wdata
PCpl4	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Jump	0.00	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Ctrl	0.00	0.00	0.83	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.14	0.00	0.00
RegDstMUX	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RF	0.00	0.00	0.00	0.00	0.90	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.00
SgnXt	0.00	0.00	0.10	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00	0.00
BR7	0.00	0.00	0.40	0.00	0.00	0.00	0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BR8	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BR9	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BR10	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
BR11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ALUCtrl	0.00	0.00	0.11	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.88	0.00	0.01	0.00	0.00
ALUSrcMux	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ALU	0.00	0.00	0.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.38	0.00	0.34	0.00	0.00
MEM	0.00	0.00	0.84	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.16	0.00
Wdata	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 17: XGBoost BugMD Confusion Matrix

Algorithm	**Macro**			**Micro**
	PRE	**REC**	**F1**	**PRE = REC = F1**
XGBoost	48.1	38.8	**40.3**	**65.8**
Random Forest	42.9	35.9	**37.8**	**65.3**
Neural Network	35.07	30.93	**32.8**	**41.41**

Table 8: BugMD Results

5.2 Testing with added Observation of internal signals of ALU

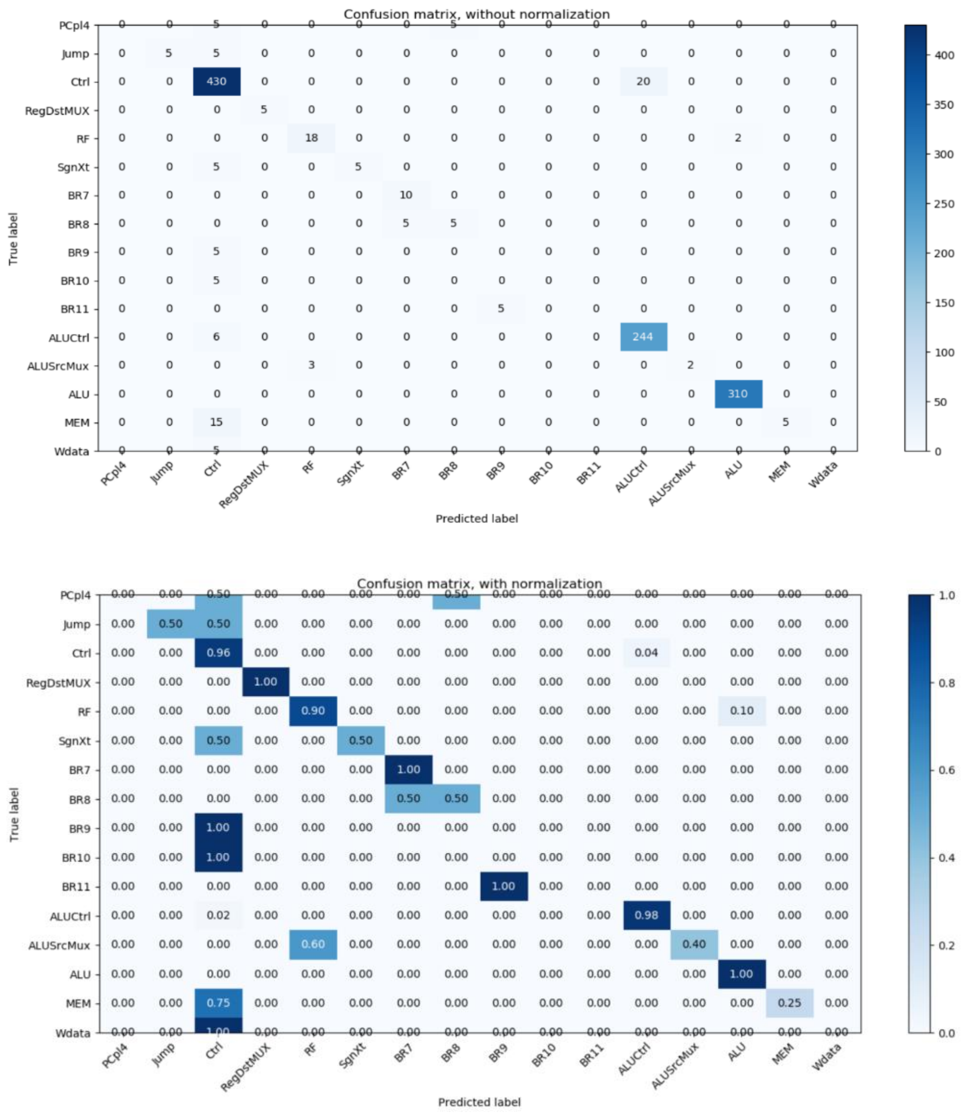
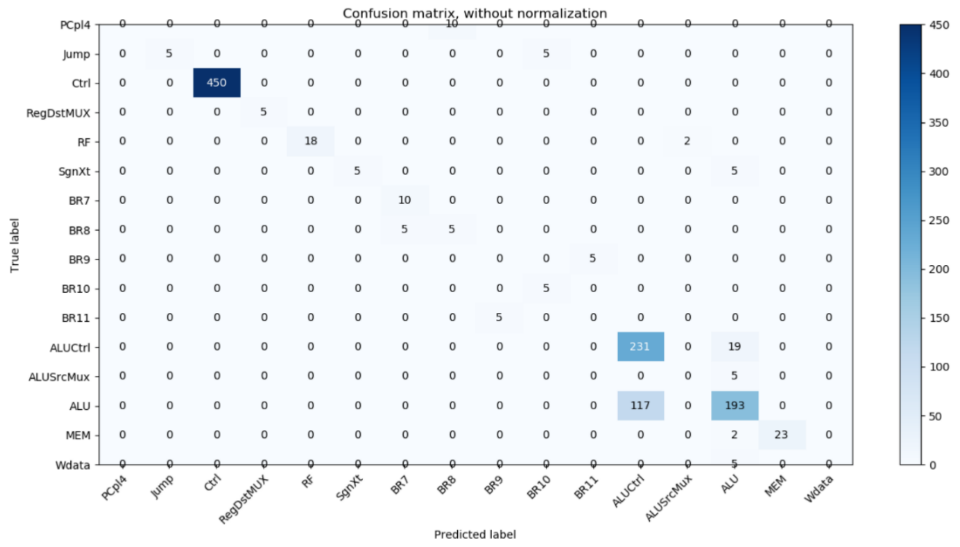


Figure 18: XGBoost Confusion Matrix of added ALU Observation

Algorithm	Macro			Micro
	PRE	REC	F1	PRE = REC = F1
XGBoost	54.1	43.7	**45.8**	**89.25**
Random Forest	47.2	41.7	**43.4**	**89.4**
Neural Network	26.5	18.9	**22.06**	**40.79**

Table 9: Results of Observing ALU

5.3 Testing with added Observation of internal signals of Control Unit



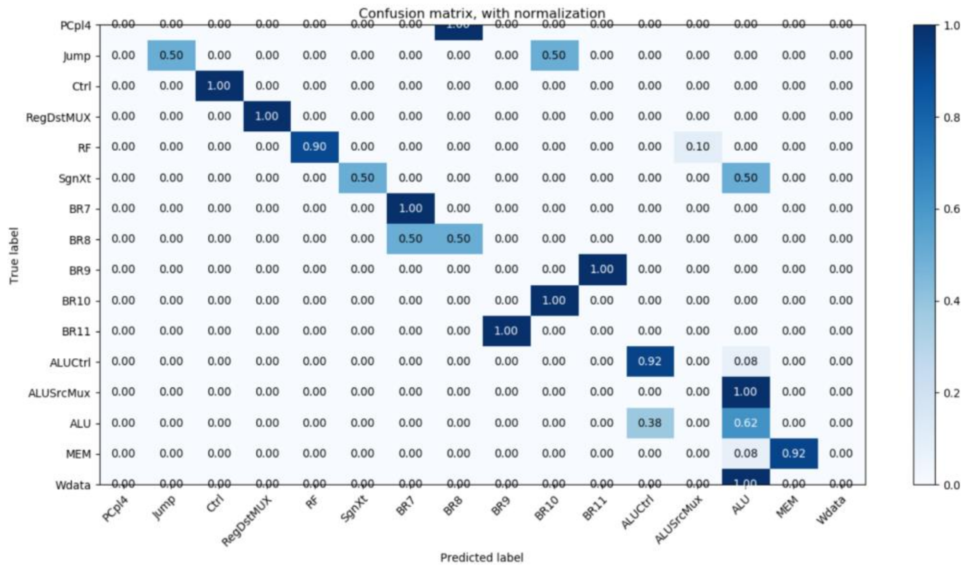


Figure 19: XGBoost Confusion Matrix of added Control unit Observation

Algorithm	**Macro**			**Micro**
	PRE	**REC**	**F1**	**PRE = REC = F1**
XGBoost	56.3	55.4	**53.7**	**83.7**
Random Forest	55.9	50.8	**50.35**	**81.9**
Neural Network	47.37	37.9	**42.11**	**43.08**

Table 10: Results of Observing Control Unit

5.4 Testing with added Observation of internal signals of ALU_Ctrl Unit

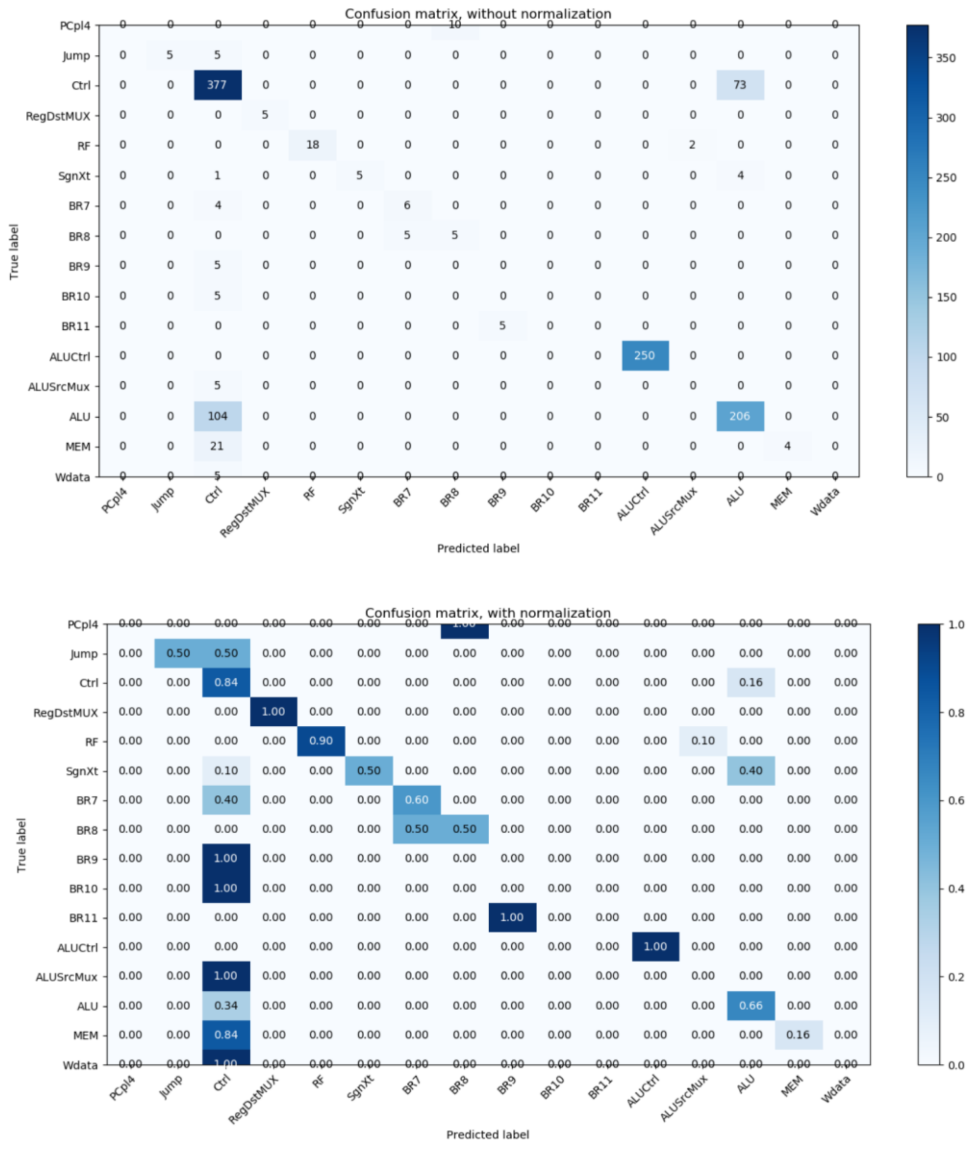


Figure 20: XGBoost Confusion Matrix of added ALU_Ctrl unit Observation

Algorithm	**Macro**			**Micro**
	PRE	**REC**	**F1**	**PRE = REC = F1**
XGBoost	51.9	41.6	**43.7**	**77.6**
Random Forest	46.5	39.8	**42**	**76.7**
Neural Network	29.64	22.08	**25.3**	**42.4**

Table 11: Results of Observing ALU_Ctrl Unit

5.5 Results Summary

To summarize the results, the 2 best classifiers that provided the highest classification accuracy were decision tree-based algorithms, XGBoost and Random forest. It is also noted that choosing to observe the control unit checker was more helpful in macro average F1 than any other units that were tested. There seems a logical explanation to the same. This is because the macro average performance depends on the classification accuracy of the individual classes rather than the overall performances of the system. When we look at the normalized confusion matrix of the vanilla BugMD approach, we see that there are many misclassifications happening from other units into control unit (observe the highly populated 3rd column). But when the control unit checker is turned on, this inherently reduces the misclassification happening into control unit class. This could be one of the reasons why observing control unit seemed to provide maximal improvement of the macro classification performance when compared to the other two units. On the other hand, ALU unit provides maximal improvement in the micro average F1 score, i.e., it improves overall accuracy maximally. This could be mainly due to the position of the ALU unit's checker when compared to ctrl unit or ALU_Ctrl Unit. The data flowing through ALU Unit has gone through the other 2 units already. This means that it has more valuable data about all its previous units' behavior which could be helping to obtain the better accuracy. Since micro F1 score only depends on the accuracy of the overall system, choosing ALU checker increases it by the maximum amount.

Design	**Macro**			**Micro**
	PRE	**REC**	**F1**	**PRE = REC = F1**
BugMD	48.1	38.8	**40.3**	**65.8**
ALU_Chkr	54.1	43.7	**45.8**	**89.25**
Ctrl_Chkr	56.3	55.4	**53.7**	**83.7**
ALUCtrl_Chkr	51.9	41.6	**43.7**	**77.6**
Ctrl_ALU_ALUCtrl_Chkr	57.8	61.8	**58.4**	**96.3**

Table 12: Summary of Results for XGBoost algorithm

Design	**Macro**			**Micro**
	PRE	**REC**	**F1**	**PRE = REC = F1**
BugMD	42.9	35.9	**37.8**	**65.3**
ALU_Chkr	47.2	41.7	**43.4**	**89.4**
Ctrl_Chkr	55.9	50.8	**50.35**	**81.9**
ALUCtrl_Chkr	46.5	39.8	**42**	**76.7**
Ctrl_ALU_ALUCtrl_Chkr	57.5	56.8	**55.1**	**94.7**

Table 13: Summary of Results for Random Forest algorithm

Design	**Macro**			**Micro**
	PRE	**REC**	**F1**	**PRE = REC = F1**
BugMD	35.07	30.93	**32.8**	**41.41**
ALU_Chkr	26.5	18.9	**22.06**	**40.79**
Ctrl_Chkr	47.37	37.9	**42.11**	**43.08**
ALUCtrl_Chkr	29.64	22.08	**25.3**	**42.4**
Ctrl_ALU_ALUCtrl_Chkr	40.27	28.7	**33.51**	**42.7**

Table 14: Summary of Results for Neural Network

6 CONCLUSION

In this thesis, we have seen that use of machine learning in design verification planning could reduce the effort involved in localizing design bugs which would eventually expedite finding the root cause of the failure. There is an added effort from the verification engineer to model the error injection and collect the data samples for the failing cases. But, across generations, many of the designs require only incremental changes. Hence, the initial effort could prove to be worthy because for every incremental change in the design, the model could potentially be reused within an error margin. With the growing complexity of ASIC functions, there is a high demand to reduce the verification effort in both time and engineer resources. For future work, implementing the same idea on more complex and industry standard designs (like RISC V) & experimenting with more complicated bugs would be a good exploration to understand the abilities or limitation of this work. This thesis work explored an idea where the resources required for debugging a failure could be reduced by the use of a bug localization technique based on machine learning.

REFERENCES

[1] S. B. Park, S. Mitra, et al., “BLoG: Post-Silicon Bug Localization in Processors using Bug Localization Graphs”, Proc. Design Automation Conf, 2010.

[2] T. Hong, D. Lin, et al., “QED: Quick error detection tests for effective post-silicon validation”, Proc. IEEE Int. Test Conf., 2010.

[3] B. Mammo, V. Bertacco, et al., “BugMD: Automatic mismatch diagnosis for bug triaging”, Proc. IEEE/ACM Int. Conf. Computer Aided Design, 2016.

[4] A. DeOrio, V. Bertacco, et al., “Machine learning-based anomaly detection for post-silicon bug diagnosis”, Proc. Design Autom. Test Europe, 2013.

[5] Y. Ma, et al., “High Performance Graph Convolutional Networks with Applications in Testability Analysis”, Proc. Design Automation Conference, 2019.

[6] S. Vasudevan, D. Johnson, et al., “GoldMine: Automatic assertion generation using data mining and static analysis”, Proc. Conf. Design Automation Test Europe, 2010.

[7] Fabian, et al., “Scikit-learn: Machine learning in python,” Journal of Machine Learning Research, vol. 12, 2011.

[8] A. Liaw and M. Wiener, “Classification and regression by random forest,” R News, vol. 2/3, December 2002.

[9] Machine Learning Basics, Terminologies & Definitions, Random Forest, XGBoost https://towardsdatascience.com.

[10] MIPS Architecture Wikipedia https://en.wikipedia.org/wiki/MIPS_architecture.

[11] ECEN 651 Lab Syllabus, MIPS Design, Texas A&M University, Spring 2019.