

Advancing Hardware Implementation: A Novel Integration of Machine Learning for Error Verification and Correction

Omri Steinberg-Tatman
EEC 283
College of Engineering
Davis, CA
ost@ucdavis.edu

Jackson Vaughn
EEC 283
College of Engineering
Davis, CA
jvaughn@ucdavis.edu

Abstract—This paper serves as an exploration of the usage of machine learning in both the verification and correction of various hardware architectures. Throughout, machine learning models are utilized to process and produce the same outputs as combinational and sequential logical circuits, covering a substantial range in size. Through careful processes, the machine learning models were able to simulate all the provided circuits with 100% accuracy. By overfitting and using finite datasets of the circuit outputs, the models could be trained to function approximately as effectively as physical hardware. This paper illustrates how hardware systems can be accurately emulated using a machine learning model and have the possibility to be more applicable than general hardware in many fields.

Index Terms—machine learning, hardware verification, hardware correction

I. OBJECTIVES

Starting at the earliest days of hardware design and implementation, faults have been a concern. From simple computational devices such as calculators to complex technological systems such as rocket launches, correct (and predictable) hardware behavior is essential for the functioning and safety of nearly all systems. Hardware can be checked ahead of time with manual fault testing and output verification, but sometimes hardware performance is important enough to be actively checked as well. These active verification processes already take place; however, machine learning is also applicable in terms of advancing this area of verification.

With the world entering a new era of machine learning and artificial intelligence-assisted computing, the advent of machine learning-assisted hardware design, verification, and active correction is not far behind. As the cost of training and processing technology becomes cheaper, machine learning models are slowly becoming more viable and cost-effective options for in-the-field verification and testing—but also the execution of pre-programmed circuits instead of the usual Field Programmable Gate Array's (FPGAs). This paper aims to show the viability of machine learning use in active verification and correction of hardware systems.

The objectives of this project are to:

- 1) Show the viability of machine learning in active hardware verification.
- 2) Extend the usage of machine learning to active large circuit verification.
- 3) Show the viability of machine learning in active hardware correction.
- 4) Extend the usage of machine learning to active large circuit correction.
- 5) Apply verification and correction models to as wide a range of circuits as possible.

II. INTRODUCTION

Verifiably correct hardware performance has been an important part of functioning computational systems since the beginning of computers; the first computers ever made had to be manually checked to ensure that the fist-sized vacuum tubes all outputted the correct signal. Now, hardware is checked by another digital system rather than someone looking at each piece by hand. Hardware has been actively checked with a smaller separate digital system for decades. For example, on the Saturn V rocket, redundant logic and a majority voting system were in place to check for issues within the hardware. That is, there were three sets of identical hardware for important systems, and those were passed into a voting system that took in all three inputs and "chose" an output that agreed with a majority of them. This simple design ensured that if one of the systems were to fail during an important operation—such as takeoff or landing—the rocket could still be kept online with the other identical hardware. A visualization of this process can be seen below in Figure 1. In this case, the correct output can be seen as a "1". If one of the systems were to fail, the voter would still have a correct output. This specific hardware construction is referred to as triple modular redundancy, but for the purposes of this paper, it will simply be referred to as redundant hardware.

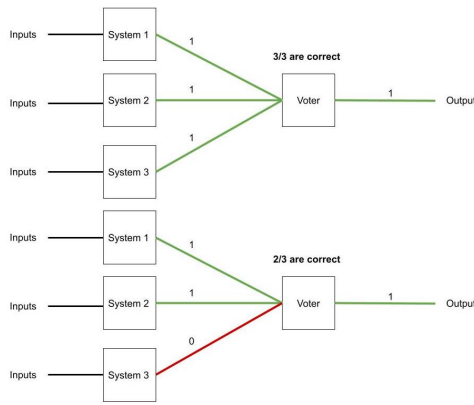


Fig. 1. Redundant Voter System

One major downfall of having multiple sets of the same hardware is not only the cost of design and construction but also the power needed to run all of the hardware simultaneously. These downfalls can be partially mitigated by having all three hardware systems running together only at critical times—such as takeoff or landing—and having only one hardware system running when the rocket is in regular use. An example of this is shown in Figure 2.

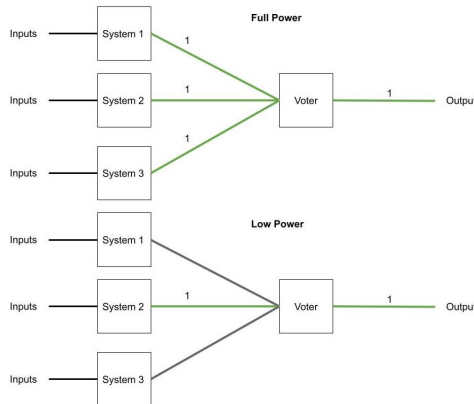


Fig. 2. Redundant Voter System With Power-Saving

The voter used with multiple redundant sets of hardware does not have any actual ability to reason what is the correct behavior of a circuit—it only has the ability to decide the majority vote of the other identical hardware. If a machine learning (ML) model instead were to be trained on the correct functioning of the circuit (i.e. how it should and should not behave) an output could be designed that verifies the implementation and tells us whether or not there is an error; this will provide information as to whether or not our hardware is working properly. Assuming the model is perfectly functional, this would save significant amounts of power and cost of redundant hardware. An example is shown in Figure 3.

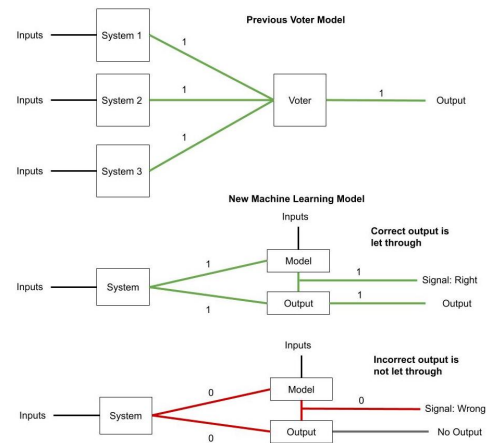


Fig. 3. Single Machine Learning Model System With Verified Output

This new type of hardware with a pre-trained machine learning model would allow for a much lower overall power draw and the ability to constantly work at peak efficiency. A verified output is extremely useful in determining the correct functioning of the circuit. The output signal can be a large help in testing hardware and making sure that any incorrect outputs are not allowed to propagate further. This type of model would have a much simpler implementation than the original redundant voter system and be significantly more open to in-mission change. However, this model only allows for the output to be correct—in the case where the output is incorrect, there is simply no output. To mitigate this issue, the model can instead take the output of the hardware and incorporate the verified decision into what the output should be. This new model is shown in Figure 4.

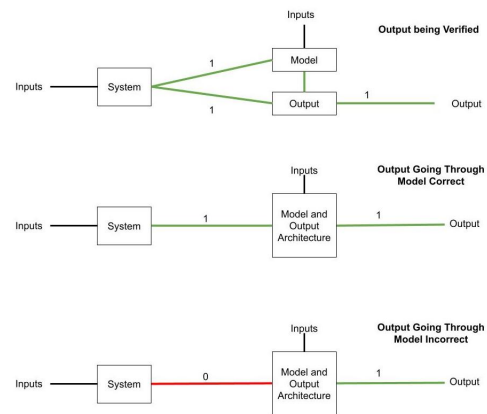


Fig. 4. Machine Learning Model With Correction Of Output

There appears to be one source of inefficiency in the above design, being that the machine learning model not only takes in the original inputs but also the output of the system itself. However, assuming that the model relates what output maps to what inputs, the hardware system could be removed in its entirety. The redesign integrates all inputs and outputs into one single model and output architecture. This would

allow for all interactions within the model to be simplified; additionally, with sufficient training, this model could even become more consistent than a hardware model. An example of this newly proposed model can be seen in Figure 5.

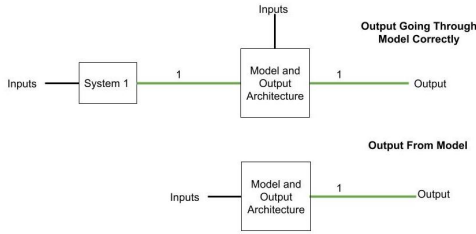


Fig. 5. Machine Learning Model Handling Inputs And Outputs

With this updated version, the application of a machine learning model is now in place of the original hardware. Though the type of model depicted in Figure 5 has yet to be deeply studied, it has been tested in the study below. Throughout this paper, the many different hardware tests that have been run with these models will be discussed for both verification and correction—one point that must be considered is whether these machine learning designs are applicable.

Currently, machine learning models have not been widely studied or used for these use cases, yet in the future they very well could apply. While a full simulation is not necessary for single or few gate circuits, a circuit can be exponentially larger; moreover, the costs of a machine learning model stay nearly constant while keeping the accuracy constant. This cost only increases during the initial training of the model, but is still significantly less than the cost of extra hardware. Ideally, future projects would expand more on machine learning’s specific use cases in larger models. As time progresses, the hardware used to run machine learning models will also improve, thus the costs will surely decrease.

The rest of this paper will discuss the uses of machine learning in hardware verification and correction; what hardware was simulated; what models were used; what types of faults were tested; and the usefulness of this in real-world scenarios.

III. WHAT IS MACHINE LEARNING?

A. Teaching a Machine

Machine learning is the process of a model changing its behavior over time in reaction to a desired set of inputs and outputs to be able to emulate those behaviors. In a way, a machine learning model “learns” how to take in and process each set of input and what outputs should correspond. In many cases, this can then be extrapolated to inputs that were not given in training but can be very accurately guessed in testing.

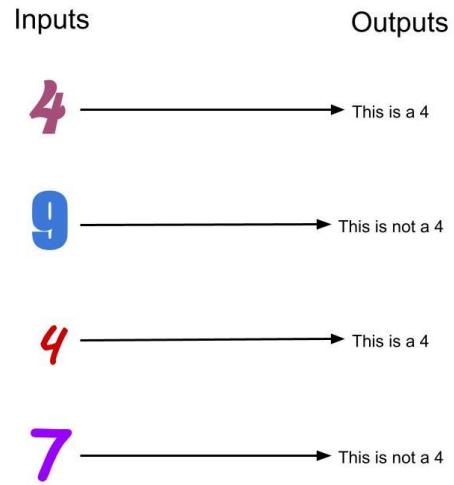


Fig. 6. Set of Inputs and Outputs

In the dataset in Figure 6, the inputs are an assortment of numbers—in various fonts and colors—and the output is whether or not the input is a 4. Once there is an established dataset, the next step is to design the machine learning model. A machine learning model is made up of layers, almost like a sandwich; the input and output layers are like the bread on each side of the sandwich. The input layer receives your inputs, and the output layer outputs what the model guesses the outputs should be. The sandwich also has to have a filling, which can be set up in countless different ways inside the machine learning model.

For the example, depicted in Figure 7, is a simple model with two dense layers of ten nodes. A node is an object with two traits: weight and bias. For the purposes of this example, the actual numbers don’t actually matter; instead, it is the inside layers that matter. These insides are made up of numbers that can change whenever necessary.

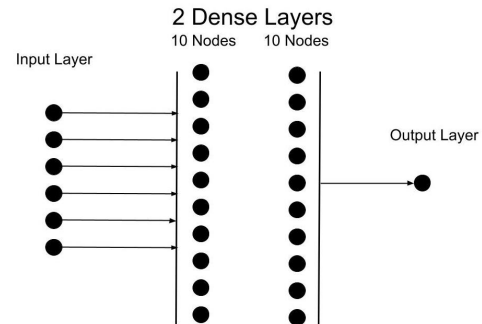


Fig. 7. Proposed Machine Learning Model Design

The input layer is indicated on the left side of Figure 7. The input layer can change in size to match the size of the input the model is given. The output layer is indicated on the right side of Figure 7. Sometimes the output layer can be more than one node if allowed by the dataset, but in this case, the

output layer is a single node. The output node will tell us one thing—whether or not the model thinks the input image given is a “4”. In between the input and output layers, there are two “Dense” layers of nodes. Now that the model has been designed, it can be trained.

A dataset, weights, and biases are necessary in order to train a model—the dataset contains the inputs and desired outputs, while weights and biases are utilized by the model to make decisions. When a model begins to train, the weights and biases of the nodes are set randomly. The model is then tested over and over again until it gets the correct output. The weights and biases influence how the model makes decisions, although the exact process is beyond the scope of this paper. In the process of testing, the machine learning model is given inputs—as shown in Figure 6—and provides outputs. At first, the machine learning model guesses whether the input is a “4” or not. Given the dataset in Figure 6, this entails a fifty percent chance of correctly guessing if the inputs are “4.” However, as time progresses the model will become more and more accurate.

B. Loss vs. Accuracy

The machine learning model makes decisions by first deciding a percentage chance it thinks the inputs are of each class—in this example being “4” or “not 4”—then choosing the class with the highest probability as the output. A theoretical understanding of how the model would behave can be seen as follows: when looking at the first image with no training, the model can only make a guess, and that guess will come in the form of two values; these values will be the two classes decided upon earlier {this is a 4, this is not a 4}. For the first image, theoretically, the model might guess {0.55, 0.45}, meaning it guesses there is a 55% chance that the image is a “4”, and a 45% chance that the image is not a “4.” Given these odds, the model would then predict that the image is a 4 even though it is not entirely sure.

An important concept to understand is the difference between accuracy and loss. Accuracy is the percentage of correct guesses that the model gets—that is, the number of correct guesses divided by the number of total guesses. In the earlier example, if the model guessed that every image is a “4,” it had a 50% accuracy. This is because 2 of the model’s 4 guesses were a “4”. If the model was trained over a longer period of time, it would continue guessing and automatically changing the node traits, slowly improving upon itself. This happens in an epoch— a set of tests. Figure 8 shows an example of how a set of epochs for Figure 6 may run, along with the guesses for each class.





Inputs	Epoch 1	Epoch 2	Epoch 3	Epoch 4	Epoch 5	Final Test
	{0.55,0.45}	{0.63,0.37}	{0.78,0.22}	{0.90,0.10}	{0.98,0.02}	This is a 4
	{0.55,0.45}	{0.40,0.60}	{0.37,0.63}	{0.22,0.78}	{0.01,0.99}	This is not 4
	{0.55,0.45}	{0.64,0.36}	{0.80,0.20}	{0.88,0.12}	{0.97,0.03}	This is a 4
	{0.55,0.45}	{0.41,0.59}	{0.16,0.84}	{0.11,0.89}	{0.06,0.94}	This is not 4

Fig. 8. Training the Model Over Time

In the figure above, it can be seen that the model’s guesses at the content of the image improve over time. At the start, the model has no idea what is going on, yet by the second epoch, the model already has 100% accuracy. This is where loss comes in as an important variable.

Loss is the difference between the percentage of guesses and what the percentage of a perfect guess would be. For example, the results of epoch 1 of the first row are {0.55,0.45}. This example has a 100% accurate guess as the model correctly thinks the image is a “4,” but it has a very bad loss as it is very uncertain with the answer. However, looking at the same row’s epoch 5, the results can be seen as {0.98, 0.02}. The loss of this epoch is very low, as a perfect guess for the image would be {1.0,0.0}, and loss is measured against a perfect guess. The actual value of loss doesn’t hold as much direct meaning as the value of accuracy, but the change of loss over time is important. It can be seen that as the model trains, the loss decreases as the guesses improve. This is desired, but there is a point at which the guesses will be “too good”. Deciding this threshold of good will decide how to fit a model.

C. Fitting a Model

A model can be retrained many times because it is just a set of nodes and values. Depending on whether the model is performing as desired, the training can continue uninterrupted, or—if the model is being trained incorrectly—the training can be reset, and the node values reset to random guesses. However, at some point, a model has to stop training and be used—at this point, it must be set up to satisfy the needs of the project.

Training a machine learning model is susceptible to two main issues in the length of training: overfitting and underfitting. This is best shown with a 2D graph of an input, demonstrating the pattern the model has developed given a dataset. Instead of the input set with images of things that are “4” or are not “4,” imagine a set of items that can be described on two number lines that can be configured as the X and Y values of a graph. These items also have been separated into two different classes, “X” and “O”. Figure 9 shows some possible fittings.

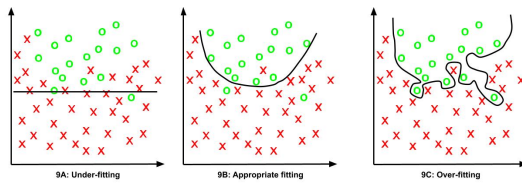


Fig. 9. Fitting A Machine Learning Model

Figure 9 shows the progression of a line fitting of a specified dataset. In machine learning, the goal is for the machine learning model to find a pattern in the dataset. Each set of data will have variations within it that are specific to that dataset; however, the model should be searching for patterns in the data that may be applied to other sets of data. Figure 9A demonstrates underfitting—when the model has created a pattern without correlation. In this case, the model has not been run long enough, and thus is unable to catch any variation within the data. This is why the model is described as “under-fitted.” An underfit model needs to be changed or trained more, and can usually be fixed. The underfitting may also vary in severity, and in the case of Figure 9A the model has extracted a bit of the pattern but not nearly enough of it.

In contrast, Figure 9B has an appropriate amount of fitting for the data. The model fits the general patterns in the data, but it does not succumb to the noise—or outliers—of the dataset. This line would be a good guess, and if new points were provided as inputs, the model would have a good chance of classing them correctly, assuming the given points are not outliers.

If the model is overtrained to account for specific noise of the dataset, it will not work well for separate sets of data—this type of outcome is demonstrated in Figure 9C. The data has been overfitted, so an actual understanding of the underlying pattern has been lost; the model ends up just fitting around every data point. Instead of looking for patterns, the model looks for specific points and makes it’s choices around them. If new data were to be provided as input, the model would decide how it compares only to the original dataset, and not the pattern of the dataset.

The main issue with overfitting is that the machine learning model, in almost all cases, will not be able to encompass every single piece of data within the possible training dataset. Returning to the example of a model deciding whether or not an image is a “4”, overfitting would require the model to be provided with every image in existence of what is or is not a “4”, which is not feasible. As a result, most models need to be trained for an underlying pattern in the data, rather than for 100% accuracy.

D. Machine Learning Applications in this Research

In the case of this research, machine learning can be used without worry of overfitting. Since the data used in this paper comes from digital logic gates and circuits—items that can be modeled out with truth tables—a machine learning model can achieve 100% accuracy. It is worth noting that a truth

table is a way to represent the inputs and outputs of a piece of digital logic, an example of which is shown in Figure 10.

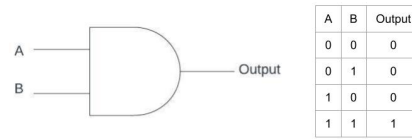


Fig. 10. And Gate

The logic gate seen here is an AND gate. It has two inputs, A and B, and one output—labeled simply as such. The inputs can be either a logical 0 or a logical 1. The output is 1 only when both A and B are 1, and is otherwise 0. The truth table in Figure 10 summarizes the possible inputs and outputs. Since the first row has A and B as 0, the output is 0. The second and third rows each have one input as a 1, and the other as a 0, so both have an output of 0. Only the last row—the case where both A and B are 1—produces an output of 1. Every logic gate function can be reduced to this formatting, with some sort of equation for the output, and thus can be represented in a truth table. Additionally, logic gates can be connected together to create larger circuits, which can also be represented using truth tables. These truth tables encompass all the possible inputs and outputs to the logic gate(s)—thus, overfitting the model to the data is the desired outcome.

To understand why overfitting works with digital logic but not for other datasets, imagine the difference between the inputs of a truth table and the example “4” image dataset. Once the “4” model is trained, it will be presented with images that may or may not have been part of the original dataset, as it is impossible to account for every image in existence. Conversely, for the truth table model, the input data encompasses every feasible input. Further, during testing or usage of the model trained on a truth table, the model will know what to output given any set of inputs, thus overfitting ensures the correct output is always produced. Looking at Figure 9C (overfitting graph), this is the desired behavior of our specific model. Various combinations of gates were modeled for the testing of the machine learning model and—using a large dataset—every input was able to be captured. Consequently, overfitting was used to allow the model to perfectly interpret the data.

There are other tools for finding the output given an input. One such tool is a look-up table, which is able to search the inputs to give the correct corresponding output. As the lookup table increases in size, it takes longer to parse. In turn, as a circuit gets larger it takes a greater amount of effort to find the answer. Although a large initial investment to design and train a machine-learning model is required, once the model is trained, it functions with the same run-time for a circuit with one AND gate or a circuit with a thousand AND gates.

This paper aims to show the viability of machine learning

to emulate circuits and provide a different path for circuit simulation than what is currently available. As discussed in the introduction, this approach could possibly be integrated into multiple hardware systems, reducing the power needed to function properly, i.e., instead of having multiple sets of hardware, a spaceship could have all of the logic centralized on two or three concurrent machine learning models—without hardware needed to produce the output. However, it should be noted that this last point is simply the next logical step and that this paper does not currently prove this capability.

IV. MODEL AND DATA IMPLEMENTATION

A. Verification and Correction

With an understanding of machine learning, the two models used in this paper can now be discussed. The first model will be referred to as the “verification model”. This model bears similarity to the model discussed in Figure 3. It will be trained on both the inputs and outputs of the hardware architecture, yielding a single output that indicates whether the data is deemed correct or incorrect. The output could then be utilized similarly to ensure that incorrect behavior does not propagate through a circuit and cannot cause further errors. The second model will be referred to as the “correction model”. Similar to the model proposed in Figure 5, this model will be trained on the hardware inputs and optimized to generate the outputs the hardware would have produced. To reiterate, the purpose of this approach is to eliminate the need for hardware entirely.

B. Combinational and Sequential Circuits

In digital logic, there exist numerous types of logical circuits. However, for the scope of this paper, only two types will be focused on: combinational and sequential circuits. Until now, only single gates have been discussed, but going forward, many tests will be conducted on circuits consisting of multiple gates. A combinational circuit comprises interconnected gates, where the inputs propagate outward to the outputs. Examples of combinational circuits are shown in Figure 11.

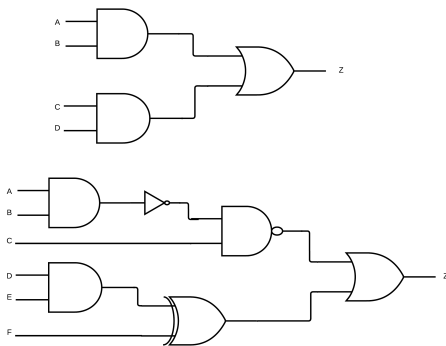


Fig. 11. Examples of Combinational Circuits

The specific functionality of these circuits is not particularly relevant. What matters is to demonstrate that the output of one gate serves as the input for the next gate, without any of the outputs being used as inputs in preceding gates. On the other hand, a sequential circuit utilizes some or all of its outputs as inputs in previous stages. This enables the circuit to incorporate not only new inputs but also the previous “state” of the circuit. Examples of sequential circuits are shown in Figure 12.

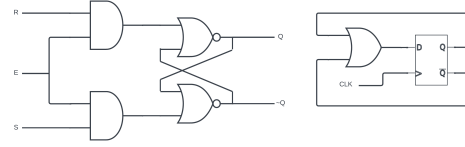


Fig. 12. Examples of Sequential Circuits

This paper will focus on testing both combinational and sequential circuits. It is important to note that due to their behavioral differences, the data must be handled differently. In the case of fault injection, which is discussed below, sequential circuit data must be handled with caution to ensure that it is not corrupted.

C. Errors and Fault Injection

To initiate the training of the correctional circuit data, no errors were introduced initially. In other words, a completely accurate list of inputs and outputs was provided to the machine. This approach was necessary to train the correction model to exhibit the correct behavior. In contrast, the verification model required the introduction of errors into the dataset. This was essential as the purpose of the verification model is to discern between correct and incorrect behaviors.

The modification of the accurate dataset is known as fault injection. This process involves taking a correct dataset and randomly altering the behavior for specific inputs. There are various types of faults that can occur in a logical circuit. However, the only aspect that is incorrect is the output. Therefore, when injecting faults into the dataset, the only component that required manipulation was the output. While many simple circuit datasets could be randomly modified, the larger combinational circuits posed a greater challenge. Manually injecting numerous errors was necessary. Checks had to be done after the error injection to make sure that the errors injected were detectable.

D. Single and Multiple Output Models

Finally, before delving into the model itself, the distinction between single and multiple output models must be addressed. A single output model resembles an example such as: {this is a 4, this is not a 4}. In this case, there are two classes, and the model only needs to select one of them. The verification model follows a single output model structure, as its sole output indicates whether the circuit’s output is correct or not.

Unfortunately, the correction model is not as straightforward. The correction model entails multiple outputs, with each individual output representing a distinct class. An illustration of a multiple-output model is shown in Figure 13.

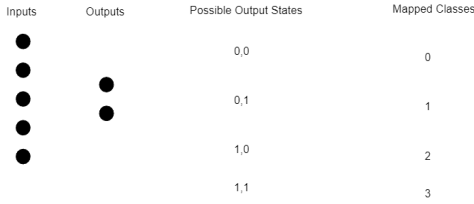


Fig. 13. Multiple Output Classes

The multiple output model in Figure 13 has 5 inputs and 2 outputs. The outputs can only assume four possible forms, as shown under "Possible Output States". Each of these forms must be associated with a corresponding class value, as shown in Figure 13 on the right. Once this mapping is completed, each input set can be processed, and the resulting output can be assigned to a specific class. Though it may appear simple, it is a complex process. Once all the data is organized into the appropriate classes, the processing can commence.

E. Design of the Model

Each of the models, the verification model and the correction model, shares a common foundational design that serves as a basis for all other circuits. The basic version of the verification model is shown in the code presented in Figure 14.

```
# define the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(100, input_shape=(n_features,), activation="relu"))
model.add(tf.keras.layers.Dense(80, activation="relu"))
model.add(tf.keras.layers.Dense(80, activation="relu"))
model.add(tf.keras.layers.Dense(30, activation="relu"))
model.add(tf.keras.layers.Dense(30, activation="relu"))
model.add(tf.keras.layers.Dense(30, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="relu"))
model.add(tf.keras.layers.Dense(5, activation="relu"))
model.add(tf.keras.layers.Dense(5, activation="relu"))
model.add(tf.keras.layers.Dense(5, activation="relu"))
model.add(tf.keras.layers.Dense(1, activation="sigmoid"))

opt = SGD(lr=0.01)

model.compile(optimizer=opt, loss="binary_crossentropy", metrics=['accuracy'])
```

Fig. 14. Verification Model

The first line of code simply sets up the model and is not particularly noteworthy. The second line of code defines the input layer based on the size of the inputs, automatically configuring it accordingly. The last two lines of code are also not particularly significant. The crucial aspect of this model lies in the dense layers. When additional dense layers and nodes are added, the model becomes more prone to overfitting, as discussed earlier. However, for the purposes of the verification model, numerous nodes and layers have been incorporated. This is because, as mentioned previously, overfitting is beneficial in this specific use case. Purposefully, a substantial number of nodes and layers have been included,

in addition to running the model for an extended duration. One last crucial point to note is that the last dense layer contains only one output. This single output indicates whether the circuit's output is correct or incorrect. The activations of each node do not matter and are outside the reach of this paper.

The correction model closely resembles the verification model, with the main distinction being the "shape" of the output. In this scenario, there are four outputs corresponding to four output classes. Although the original circuit had only two outputs, as depicted in Figure 13, they ultimately led to four classes. The correction model is shown in Figure 15.

```
# define the model
model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(100, input_shape=(n_features,), activation="relu"))
model.add(tf.keras.layers.Dense(80, activation="relu"))
model.add(tf.keras.layers.Dense(80, activation="relu"))
model.add(tf.keras.layers.Dense(30, activation="relu"))
model.add(tf.keras.layers.Dense(30, activation="relu"))
model.add(tf.keras.layers.Dense(30, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="relu"))
model.add(tf.keras.layers.Dense(10, activation="relu"))
model.add(tf.keras.layers.Dense(5, activation="relu"))
model.add(tf.keras.layers.Dense(5, activation="relu"))
model.add(tf.keras.layers.Dense(5, activation="relu"))
model.add(tf.keras.layers.Dense(4, activation="softmax"))

opt = SGD(lr=0.01)

model.compile(optimizer=opt, loss="categorical_crossentropy", metrics=['accuracy'])
```

Fig. 15. Correction Model

V. DATA GENERATION

A. Designing Modules

The first crucial step in generating the necessary data for training and testing the machine learning model involved designing the modules used for data collection. Verilog and ModelSim, two digital design software tools, were employed for module design and testing, enabling the generation of the test data. When selecting the modules to utilize, it was necessary to include a wide range of both combinational and sequential circuits.

Initially, basic logic gates such as AND, OR, and XOR were chosen. Subsequently, more intricate combinational components were implemented, as will be elaborated below. Additionally, to ensure thorough testing of various operations, an Arithmetic Logic Unit (ALU) was incorporated. The ALU is capable of performing mathematical operations, shifts, logic operations, rotations, and more.

While the aforementioned selection covers a diverse range of combinational circuits, it was also important to include a variety of sequential circuits in the testing process. However, the testing of sequential models was not as extensive as that of combinational models. Nonetheless, the selected models for testing sequential circuits included a T Flip Flop and a D Flip Flop, both of which will be discussed later.

B. Generating Data

After creating all the modules, ModelSim was utilized to generate the required data. The input format for the machine learning model was in the form of a CSV file. To generate the data, Verilog test benches were employed to test all

the previously mentioned modules for every possible input combination. A test bench is essentially a set of code utilized to test Verilog modules. By simulating the test benches using ModelSim, the resulting transcript text output could be utilized as the CSV data (with minor aesthetic adjustments).

It is important to note that the machine learning model should not merely memorize the order of the CSV lines but rather understand the intended functionality of each module. To address this concern, a simple Python script was employed to "shuffle" the lines of the CSV file. Each CSV file consisted of approximately 100,000 lines, ensuring thorough training of the model.

VI. COMBINATIONAL CIRCUIT TESTING

A. Proof of Concept (High/Low)

The modeling work began with a set of data to be used for a proof of concept. This proof of concept was of an output that would be either a "1" or a "0", which is high or low. That model was trained on the idea that the output of a "1" was correct and that the output of a "0" was incorrect. An example of the input data is shown Figure 16.



Fig. 16. High/Low Model Input

The model was given a randomized set of data points, with the inputs being either a "0" or a "1", simulating the output of another piece of hardware in series with the model. After very little training, the model was able to get to 100% accuracy with essentially no loss. The final loss and accuracy during training, along with a semi-standardized output test (also used in later examples), is shown in Figure 17.

```
Class 0:
  Total: 4904
  Correct: 4904
  True 0: 4904
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 5095
  Correct: 5095
  True 1: 5095
  False 0: 0
  Accuracy: 1.0

loss: 6.6736e-04 - accuracy: 1.0000
```

Fig. 17. Final Output Statistics of High/Low Model

B. Single Gate

Single gates serve as the building blocks of larger circuits and are an ideal starting point for individual testing. In the

case of the single gates depicted below, data generation was performed by utilizing Verilog modules to simulate the expected correct output. The models designed for these single gates exhibited satisfactory performance and trained rapidly, achieving 100% accuracy within the initial few epochs. Once the basic model was established and operational, the verification and correction processes proceeded smoothly without encountering any errors or issues.

1) *AND Gate*: The first single-gate circuit chosen was an AND gate.

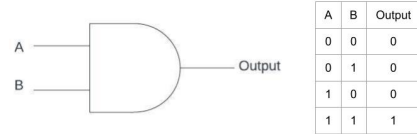


Fig. 18. And Gate

The complete functionality of an AND gate can be summarized using a truth table, as mentioned earlier. By utilizing the AND gate Verilog module, 100,000 data points were generated, comprising random combinations of the inputs $\{0,0\}$, $\{0,1\}$, $\{1,0\}$, $\{1,1\}$. These data points encompass all possible input-output pairs, and the models are trained using this dataset. The initial test conducted was for verification, which yielded rapid results. The inputs for the verification model included the two inputs of the AND gate, as well as its corresponding output. The model's output was a single class, denoted as "0" if the output was incorrect, or "1" if the output was correct. The training process progressed swiftly and achieved 100% accuracy within four epochs. Figure 19 presents the statistics of the verification output.

```
Class 0:
  Total: 4954
  Correct: 4954
  True 0: 4954
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 5045
  Correct: 5045
  True 1: 5045
  False 0: 0
  Accuracy: 1.0

loss: 1.0585e-06 - accuracy: 1.0000
```

Fig. 19. Final Output Statistics For AND Gate Verification Model

Additionally, a correction model was run. For the correction model, the inputs were the two inputs of the AND gate, and the correct output was the desired output of the AND gate. In this case, there is only one output node, so no extra classes were needed, and the model still had a binary output, either a "0" or a "1". The final output statistics of the correction

model are shown in Figure 20.

```

Class 0:
  Total: 7477
  Correct: 7477
  True 0: 7477
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 2522
  Correct: 2522
  True 1: 2522
  False 0: 0
  Accuracy: 1.0

loss: 1.6351e-04 - accuracy: 1.0000

```

Fig. 20. Final Output Statistics For AND Gate Correction Model

It can be noted for the correction model that only 10,000 data points were used for testing, and that is because the rest of the data points were used in training. In theory, for this gate, every input and output combination could be shown in only 4 data points, so if the testing or training set was to be shortened, it would function the same.

2) *OR Gate*: The second single-gate circuit that was chosen was an OR gate. This is a gate that has an output of “1” if either the A input or the B input is “1”. Otherwise, the output is a “0”. An example of an OR gate is shown in Figure 21.



Fig. 21. OR Gate

Similarly to the AND gate, the functionality of the OR gate can be effectively captured in the truth table shown in Figure 21. By utilizing the OR gate Verilog module, it was possible to generate 100,000 data points with randomized input sets $\{\{0,0\}, \{0,1\}, \{1,0\}, \{1,1\}\}$. These data points encompassed all possible combinations of inputs and outputs, serving as the training dataset for the models. The inputs for the verification model were structured similarly to the AND gate, consisting of the two inputs to the OR gate along with its corresponding output. The model’s output was represented as a single class, with “0” denoting an incorrect output and “1” indicating a correct output. The training process for the OR gate model was also rapid, achieving 100% accuracy within five epochs. Figure 22 displays the statistics of the verification output.

```

Class 0:
  Total: 5048
  Correct: 5048
  True 0: 5048
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 4951
  Correct: 4951
  True 1: 4951
  False 0: 0
  Accuracy: 1.0

loss: 1.6073e-04 - accuracy: 1.0000

```

Fig. 22. Final Output Statistics For OR Gate Verification Model

The correction model was also run, with the same binary “1” or “0” output as the verification model. The final output statistics of the correction model is shown in Figure 23.

```

Class 0:
  Total: 2518
  Correct: 2518
  True 0: 2518
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 7481
  Correct: 7481
  True 1: 7481
  False 0: 0
  Accuracy: 1.0

loss: 1.6268e-04 - accuracy: 1.0000

```

Fig. 23. Final Output Statistics For OR Gate Correction Model

The same mixing of data points exists here as in the AND gate model. The losses and accuracies for the OR gate indicate a precise model.

3) *XOR Gate*: The final single-gate circuit that was chosen was an XOR gate. This is a gate that has an output of “1” if the A input and the B input are opposites, such as in the sets $\{\{0,1\}, \{1,0\}\}$. Otherwise, the output is a “0” (if the A input and the B input are the same). An example of the XOR gate is shown in Figure 24.

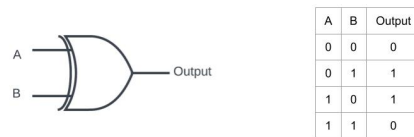


Fig. 24. XOR Gate

The XOR gate presents a slightly more intricate scenario compared to the other two gates. The model must comprehend that the inputs must differ and not merely be present. Similarly to the other single gates, the XOR gate’s behavior can be effectively captured using a concise truth table, enabling the model to learn rapidly. In the case of the verification model, the inputs were structured akin to the AND gate, incorporating the two inputs to the XOR gate alongside its output. The model’s output was represented as a single class, where “0” indicated an incorrect output and “1” denoted a

correct output. The training process for the XOR gate model also proceeded swiftly, achieving 100% accuracy within eight epochs. Figure 25 showcases the statistics of the verification output.

```

Class 0:
  Total: 4933
  Correct: 4933
  True 0: 4933
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 5066
  Correct: 5066
  True 1: 5066
  False 0: 0
  Accuracy: 1.0
loss: 1.2419e-06 - accuracy: 1.0000

```

Fig. 25. Final Output Statistics For XOR Gate Verification Model

The correction model was also run with the same binary “1” or “0” output as the verification model. The final output statistics of the correction model are shown in Figure 26.

```

Class 0:
  Total: 4918
  Correct: 4918
  True 0: 4918
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 5081
  Correct: 5081
  True 1: 5081
  False 0: 0
  Accuracy: 1.0
loss: 1.0376e-06 - accuracy: 1.0000

```

Fig. 26. Final Output Statistics For XOR Gate Correction Model

It can be seen that through the testing of single circuits, the model is able to very quickly understand the meaning and pattern of the different gates and implement the pattern within good reason. The next step was to create these models for larger and more complex combinational circuits.

C. Multiple Gate Circuits

Many multiple-gate circuits perform a real function and not just a smaller abstract idea of AND and OR circuits. Additionally, many circuits have multiple outputs and have to have all of the details set into different classes for the output. These models, in general, took a lot longer to train and were sometimes more difficult to get working as expected. In general, as the circuit became more complex in functionality, the model required more epochs to reach 100% accuracy.

1) *C17*: C17 is a benchmark combinational circuit that is made up of 6 NAND gates. C17 is a basic circuit that has been used in many types of hardware error and fault testing over the years. A diagram of C17 is shown in Figure 27.

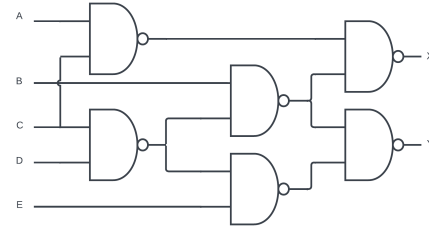


Fig. 27. C17 combinational circuit

The C17 circuit presents the first instance of multiple outputs. Injecting errors and faults into the verification model proved slightly more challenging. Although the model still had a single output for verification, caution had to be exercised when introducing errors. A random set of outputs was generated to correspond to each set of inputs that were deemed to contain an error. It was necessary to verify that the randomly generated set of outputs differed from the original set of outputs. If they were equivalent, the outputs had to be regenerated until a different output was achieved. This issue became more pronounced in subsequent circuits and posed a significant debugging step to ensure the proper functioning of data generation. Figure 28 provides the output statistics of the verification model.

```

Class 0:
  Total: 11583
  Correct: 11583
  True 0: 11583
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 11748
  Correct: 11748
  True 1: 11748
  False 0: 0
  Accuracy: 1.0
loss: 8.8454e-07 - accuracy: 1.0000

```

Fig. 28. Final Output Statistics For C17 Verification Model

The C17 correction model was the first model to include multiple output classes. Similarly to Figure 13, four output classes had to be assigned to have the model read properly. Figure 29 shows the output statistics for the correction model.

```

Class 0:
  Total: 6571
  Correct: 6571
  True 0: 6571
  False 1: 0
  False 2: 0
  False 3: 0
  Accuracy: 1.0
Class 1:
  Total: 3602
  Correct: 3602
  True 1: 3602
  False 0: 0
  False 2: 0
  False 3: 0
  Accuracy: 1.0
Class 2:
  Total: 3597
  Correct: 3597
  True 2: 3597
  False 0: 0
  False 1: 0
  False 3: 0
  Accuracy: 1.0
Class 3:
  Total: 9561
  Correct: 9561
  True 3: 9561
  False 0: 0
  False 1: 0
  False 2: 0
  Accuracy: 1.0
loss: 7.1704e-07 - accuracy: 1.0000

```

Fig. 29. Final Output Statistics For C17 Correction Model

The output in Figure 29 shows that the correction model worked very well and is a great example of functionality for larger circuits.

2) *Multiplexer*: Next, the multiplexer gates were tested with our models. A multiplexer takes in many inputs and only has one output, and depending on the "select pins," will pass one of its inputs to the output. The truth table and diagram for a multiplexer, specifically an 8 to 1 multiplexer, is shown in Figure 30 below.

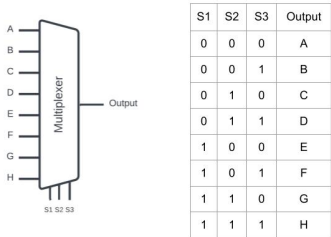


Fig. 30. 8 to 1 Multiplexer

The verification data for the multiplexer was fairly simple but provided an interesting experiment for testing the ability of the model to understand. Because there was only one output, the verification model had to recognize the pattern of the selected input going to the output. Like the other larger circuits, the verification model trained quickly. The models' ability to train quickly despite the seeming complexity of the component proved interesting. The output of the verification model is shown in Figure 31.

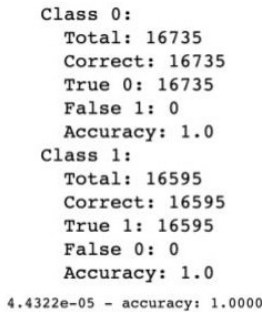


Fig. 31. Final Output Statistics For 8 to 1 Multiplexer Verification Model

The multiplexer correction model also had a single output, which made it significantly easier than the C17 circuit to generate the data. The output of the correction model is shown in Figure 32.

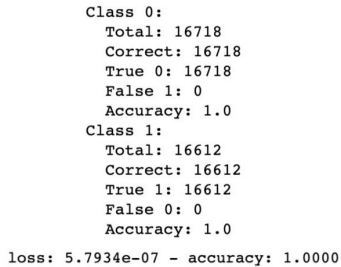


Fig. 32. Final Output Statistics For 8 to 1 Multiplexer Correction Model

3) *Demultiplexer*: The results obtained with the multiplexer instilled confidence that the process would proceed smoothly with the demultiplexer. However, challenges persisted concerning the correction data. Unlike the multiplexer, the demultiplexer functions in the opposite manner, featuring one input and eight outputs. An output is chosen by utilizing select pins, and the input is routed to the selected output. The truth table and diagram for a demultiplexer, specifically a 1-to-8 demultiplexer, are depicted in Figure 33.

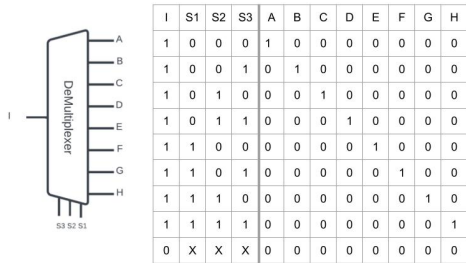


Fig. 33. 8 to 1 Demultiplexer

The demultiplexer verification data was fairly simple, with the only issue being the fact that all false outputs had to be checked to confirm that they were, in fact, false. The output results of the verification model are shown in Figure 34.

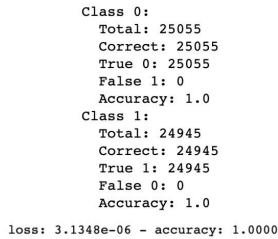


Fig. 34. Final Output Statistics For 1 to 8 Demultiplexer Verification Model

The correction data for the demultiplexer was fairly complicated. The data had to be separated into 9 different classes, as shown in Figure 33. This took a while to accomplish and had some issues initially, but once all the data was corrected, the model trained quickly. The output of the correction model is shown in Figure 35.

```

Class 0:
Total: 16613
Correct: 16613
True 0: 16613
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 1:
Total: 2097
Correct: 2097
True 1: 2097
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 2:
Total: 2117
Correct: 2117
True 2: 2117
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 3:
Total: 2048
Correct: 2048
True 3: 2048
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 4:
Total: 2047
Correct: 2047
True 4: 2047
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 5:
Total: 2112
Correct: 2112
True 5: 2112
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 6:
Total: 2079
Correct: 2079
True 6: 2079
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0
Class 7:
Total: 2107
Correct: 2107
True 7: 2107
False: (0, 0, 0, 0, 0, 0, 0, 0)
Accuracy: 1.0

```

loss: 4.6966e-07 - accuracy: 1.0000

Fig. 35. Final Output Statistics For 1 to 8 Demultiplexer Correction Model

4) *ALU*: All of the previous circuits solely execute a single function. However, for the purpose of this paper, it is crucial to demonstrate that a model can be utilized for combinational circuits that encompass a range of functions. This is where the Arithmetic Logic Unit (ALU) proves its utility. The ALU designed for this paper is a 16-function ALU capable of performing various computational operations. These operations include basic functions such as addition, subtraction, multiplication, and division, as well as rotations, shifts, and other useful operations. By employing this ALU, it becomes evident that our models can be applied to diverse functions accurately with minimal adjustments. A description of the ALU is shown in Figure 36.

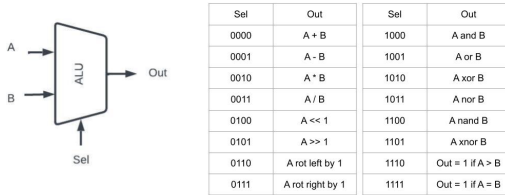


Fig. 36. ALU

For the verification model, corrupting specific data points proved to be quite intricate. Randomizing the output required it to be distinct from the correct output while still being a valid output for the ALU. Initially, this presented numerous challenges and it took several iterations to generate a valid set of incorrect data for the ALU. The process involved introducing various types of errors in the inputs and outputs, ultimately resulting in incorrect outputs. Figure 37 displays the output statistics for the verification model.

```

Class 0:
Total: 82028
Correct: 82028
True 0: 82028
False 1: 0
Accuracy: 1.0
Class 1:
Total: 81812
Correct: 81812
True 1: 81812
False 0: 0
Accuracy: 1.0

```

loss: 4.7324e-06 - accuracy: 1.0000

Fig. 37. ALU Verification Model

Interestingly, the model took nearly 1000 epochs to reach 100 % accuracy. In contrast, many other circuits achieved perfect accuracy within 10 epochs. This discrepancy can be attributed to the size and complexity of the data, as well as the model's dimensions, which align with the objective of overfitting. The output statistics for the correction model are shown in Figure 38.

```

Class 0: Accuracy: 1.0
Class 1: Accuracy: 1.0
Class 2: Accuracy: 1.0
Class 3: Accuracy: 1.0
Class 4: Accuracy: 1.0
Class 5: Accuracy: 1.0
Class 6: Accuracy: 1.0
Class 7: Accuracy: 1.0
Class 8: Accuracy: 1.0
Class 9: Accuracy: 1.0
Class 10: Accuracy: 1.0
Class 11: Accuracy: 1.0
Class 12: Accuracy: 1.0
Class 13: Accuracy: 1.0
Class 14: Accuracy: 1.0
Class 15: Accuracy: 1.0
Class 16: Accuracy: 1.0
Class 17: Accuracy: 1.0
Class 18: Accuracy: 1.0
Class 19: Accuracy: 1.0
Class 20: Accuracy: 1.0
Class 21: Accuracy: 1.0
Class 22: Accuracy: 1.0
Class 23: Accuracy: 1.0
Class 24: Accuracy: 1.0
Class 25: Accuracy: 1.0
Class 26: Accuracy: 1.0
Class 27: Accuracy: 1.0
Class 28: Accuracy: 1.0
Class 29: Accuracy: 1.0
Class 30: Accuracy: 1.0
Class 31: Accuracy: 1.0
Class 32: Accuracy: 1.0
Class 33: Accuracy: 1.0
Class 34: Accuracy: 1.0
Class 35: Accuracy: 1.0
Class 36: Accuracy: 1.0
Class 37: Accuracy: 1.0
Class 38: Accuracy: 1.0
Class 39: Accuracy: 1.0
Class 40: Accuracy: 1.0
Class 41: Accuracy: 1.0
Class 42: Accuracy: 1.0
Class 43: Accuracy: 1.0
Class 44: Accuracy: 1.0
Class 45: Accuracy: 1.0
Class 46: Accuracy: 1.0
Class 47: Accuracy: 1.0
Class 48: Accuracy: 1.0
Class 49: Accuracy: 1.0
Class 50: Accuracy: 1.0
Class 51: Accuracy: 1.0
Class 52: Accuracy: 1.0
Class 53: Accuracy: 1.0
Class 54: Accuracy: 1.0
Class 55: Accuracy: 1.0
Class 56: Accuracy: 1.0
Class 57: Accuracy: 1.0
Class 58: Accuracy: 1.0
Class 59: Accuracy: 1.0
Class 60: Accuracy: 1.0
Class 61: Accuracy: 1.0
Class 62: Accuracy: 1.0
Class 63: Accuracy: 1.0
Class 64: Accuracy: 1.0

```

loss: 8.2208e-04 - accuracy: 1.0000

Fig. 38. ALU Correction Model

D. Sequential Circuits

As mentioned earlier, sequential circuits utilize their output as an input to themselves. This adds complexity when applying a machine learning model to their dataset, as including only the external circuit's inputs (such as the clock and enable) is insufficient. It is also crucial to incorporate the previous output as an input. This aspect posed significant challenges during the initial design of the models and data. Multiple iterations were necessary before the model could properly comprehend the data, and during training, the model often experienced accuracy loss and reverted to random guessing.

1) *D Flip Flop*: The D Flip Flop serves as the initial test for sequential circuits. It is a circuit capable of retaining its previous state and performing an action based on the input,

but only on the positive edge of the clock. The specific operation performed is inconsequential; what matters is that the action occurs when the clock input transitions from "0" to "1". When the clock remains unchanged or transitions from "1" to "0", there are no changes in the output. The truth table and diagram for a D Flip Flop are shown in Figure 39.

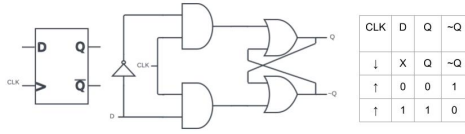


Fig. 39. D Flip-Flop

Setting up and utilizing this particular dataset proved to be quite complex as well. Due to the inherent randomness in testing the inputs for training the model, tracking the previous state of the circuit was challenging. In the initial iterations of training with the input data, the model struggled to make any progress. After extensive testing, it was discovered that the pattern of the rising edge of the clock needed to be included in the data, along with the previous state of the circuit. Once the setup was adjusted accordingly, the verification and correction models proceeded smoothly. The verification model statistics are shown in Figure 40.

```

Class 0:
  Total: 16638
  Correct: 16638
  True 0: 16638
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 16692
  Correct: 16692
  True 1: 16692
  False 0: 0
  Accuracy: 1.0
loss: 5.2936e-06 - accuracy: 1.0000

```

Fig. 40. Final Output Statistics For D Flip-Flop Verification Model

The correction model was fairly simple once the correct input data was set up. The model only had one output, and it was a simple binary class decision. The final output statistics for the correction model are shown in Figure 41.

```

Class 0:
  Total: 5559
  Correct: 5559
  True 0: 5559
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 27771
  Correct: 27771
  True 1: 27771
  False 0: 0
  Accuracy: 1.0
loss: 1.0865e-06 - accuracy: 1.0000

```

Fig. 41. Final Output Statistics For D Flip-Flop Correction Model

2) *T Flip Flop*: The T Flip Flop is similar to the D Flip Flop and performs an action on the previous state, depending on the input. This had the same input data issue as the D Flip Flop, but once it was deciphered, the testing worked well. The truth table and diagram of a T Flip-Flop are shown in Figure 42.

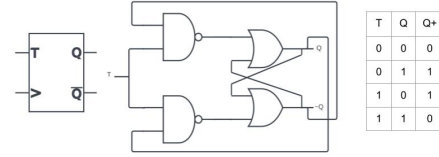


Fig. 42. T Flip-Flop

This model trained well, similar to the D Flip Flop, with few differences in output. The verification model statistics are shown in Figure 43.

```

Class 0:
  Total: 16648
  Correct: 16648
  True 0: 16648
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 16682
  Correct: 16682
  True 1: 16682
  False 0: 0
  Accuracy: 1.0
loss: 6.0118e-07 - accuracy: 1.0000

```

Fig. 43. Final Output Statistics For T Flip-Flop Verification Model

The correction model statistics are shown in Figure 44.

```

Class 0:
  Total: 16496
  Correct: 16496
  True 0: 16496
  False 1: 0
  Accuracy: 1.0
Class 1:
  Total: 16834
  Correct: 16834
  True 1: 16834
  False 0: 0
  Accuracy: 1.0
loss: 1.0268e-06 - accuracy: 1.0000

```

Fig. 44. Final Output Statistics For T Flip-Flop Correction Model

VII. ISSUES

While many of these models have achieved the desired output, there are still several factors that have not been taken into account in terms of their practical application. Although these models were assumed to be perfect, they rely on hardware such as processors and GPUs to function, which cannot be bypassed. This brings into question when machine learning is more efficient than hardware.

Unlike hardware, machine learning does not guarantee perfectly accurate results, even when trained to an optimal state. Machine learning models are still subject to errors, despite being trained with vast amounts of data. On the other hand, well-built and tested hardware can sometimes provide more reliable output. However, based on our research, it appears that the machine learning models created achieved 100% accuracy.

Another concern is the high initial cost of training a machine learning model. It involves not only running the model itself but also the manual processing of input data. Many applications require specialized engineers or even teams of engineers to ensure their proper functioning. Furthermore, all the models tested in this context were relatively small and simple, requiring much more extensive testing before they could be considered usable in a consumer environment.

VIII. CONCLUSIONS

Machine learning is a powerful tool in many research sectors and activities and was shown to be applicable in the correction and verification of hardware. By leveraging overfitting in training, machine learning models can accurately process combinational and sequential circuits. These models successfully analyzed combinational and sequential circuits of different sizes, demonstrating that machine learning can serve as both a complementary and standalone verification tool for hardware systems. Moreover, machine learning has the potential to correct hardware systems. Machine learning models have shown to be a viable choice for simulating and implementing various types of hardware that would otherwise exist physically.

IX. IMPLEMENTATION IN THE REAL WORLD

By employing machine learning techniques, hardware can be trained, retrained, and reset as needed. The centralization of hardware on models offers several benefits. For instance, predictions can be made with a specific input power requirement, potentially resulting in improved power consumption, particularly in scenarios involving significant amounts of hardware. It is important to note that this technology extends beyond the previous spaceship example and can be applied to various domains. As machine learning hardware becomes more affordable and compact, it can be utilized in data centers, graphics cards, and even smaller electronic devices like smartphones and laptops.

One aspect that this paper does not address in depth is determining the circuit size at which using a model becomes more advantageous than physically building the hardware. Future research should explore the power consumption threshold at which models become applicable and assess the feasibility of implementing machine learning models in small-scale devices such as laptops and smartphones. Additionally, experiments could be conducted to investigate whether a machine learning model can transition from partial input and output data to

comprehending and interpreting a complete working model. Furthermore, comparing machine learning models with other simulation-driven hardware, such as field-programmable gate arrays (FPGAs), could shed light on the potential advantages of machine learning models in running designed circuits. This paper has laid the groundwork for further research in this field and offers a solid foundation for future investigations.

ACKNOWLEDGMENT

Without Professor Hussain Al-Asaad for his mentorship and educational prowess, this paper would not have been possible. The paper would not have been complete without the assistance of Gali Steinberg-Tatman, Hans O'Flaherty, Aman Ganapathy, Bella Doherty, and Alexander Gardner for their amazing proofreading, commenting, and editing of this paper. We would also like to thank Marji K. LeGrand for her incomparable advice, wisdom, and help.

REFERENCES

- [1] Dinu, A.; Ogrutan, P.L. "Reinforcement Learning Made Affordable for Hardware Verification Engineers". *Micromachines* 2022, 13, 1887. <https://doi.org/10.3390/mi13111887>
- [2] P. Gaur, S. S. Rout and S. Deb, "Efficient Hardware Verification Using Machine Learning Approach," 2019 IEEE International Symposium on Smart Electronic Systems (iSES) (Formerly iNiS), Rourkela, India, 2019, pp. 168-171, doi: 10.1109/iSES47678.2019.00045.
- [3] Dave, Abhilasha Harsukhbhai. "Application of Machine Learning in Digital Logic Circuit Design Verification and Testing", Scholarworks Aug. 2018, scholarworks.calstate.edu/.