The Thesis Committee for Yuxin Wang
certifies that this is the approved version of the following Thesis:

Extending Capability of Formal Tools: Applying Semiformal Verification on Large Design

APPROVED BY

SUPERVISING COMMITTEE:

Jacob A. Abraham, Supervisor

Andreas M. Gerstlauer

Extending Capability of Formal Tools: Applying Semiformal Verification on Large Design

by

Yuxin Wang

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN ENGINEERING

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2019

Dedicated to my family and friends.

Acknowledgments

I would like to thank my supervisor Prof. Jacob Abraham for providing me direction in this interesting domain of research and bringing me inspiration when I encountered difficulties. Also, I want to thank him for offering me the opportunity to be his teaching assistant, which helped me bolster my understanding of digital design.

I appreciate it very much that Dr. Jacob Chang, Mr. Clifford Wolf and Sharukh Shaikh provide kind help for me during my research. I also want to specially thank my parents and my boyfriend Zhengping Yu for their continuous love and support throughout my master study.

I would like to take this opportunity to specially thank Prof. Andreas Gerstlauer for taking out his valuable time to be the reader for my thesis.

Extending Capability of Formal Tools: Applying Semiformal Verification on Large Design

Yuxin Wang, M.S.E.
The University of Texas at Austin, 2019

Supervisor: Jacob A. Abraham

Simulation and formal verification (FV) are the two most commonly used techniques for verifying a digital design described at the Register-Transfer Level (RTL). Compared to simulation, formal verification shows an advantage in terms of exhaustive design coverage. However, due to state-space explosion, it is limited in size of designs that can be analyzed, and this capacity problem remains a big issue for application in large designs, such as processors.

In this thesis, a waypoint-based semiformal verification (SFV) method is proposed in order to extend formal tool capacity for large designs. Our algorithm involves formal engines to explore traces to hit waypoints, reducing the computation time and memory required to reach a desired state. In addition, an automatic waypoint generation tool is developed. Criteria are developed to identify important flip-flops in the design to generate the waypoints, based on information from the synthesized netlist. A neural network is trained to score all the flip-flops in the target design. Based on the predicted scores, we set a threashold to select the critical flip-flops and then generate waypoint guides for RTL verification.

The process is first studied using a small FIFO example. Then an expandable end-to-end ISA verification framework designed around a RISC-V core is evaluated with the proposed SFV techniques. The results show that waypoint-based SFV and the automatic waypoint generation algorithm have great potential in RTL verification. SFV can save a substantial amount of the time and memory required to cover all important scenarios, compared to direct application of FV.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Classic Verification Techniques

There is a desperate need for practical solutions to the problem of performing design verification using today's computing systems, since most modern digital designs are far too large for traditional tests to guarantee their correctness and safety. Two main approaches are applied in RTL verification: dynamic simulation and formal verification (FV) [2].

Simulation-based verification has been applied for a long time. This approach requires a testbench with manually designed or constrained random test vectors to propagate logic values through blocks [36]. It is acknowledged that the traditional simulation method offers the benefits of simplicity and scalability; however, it is impractical for simulation to keep up with the increasing design complexity [17], and its reliability depends on how comprehensive the stimuli are. For example, if the design under verification (DUV) state machine has 64 state variables, then the total time required to simulate vectors to cover all states will be nearly 600,000 years if each vector takes 1 ms to simulate.

Formal methods are increasingly applied in the verification of complex digital designs for their ability to detect subtle bugs [41]. The idea of formal verification is derived from reasoning based on manipulation of formulas and logic, that is, the proof is mathematical instead of experimental [24]. Generally, the name formal can refer to many sub-topics, such as logic equivalence checking [38], model checking [9, 13] (also known as functional formal analysis), and theorem proving [27]. In this thesis, we will only be dealing with formal methods using model checking.

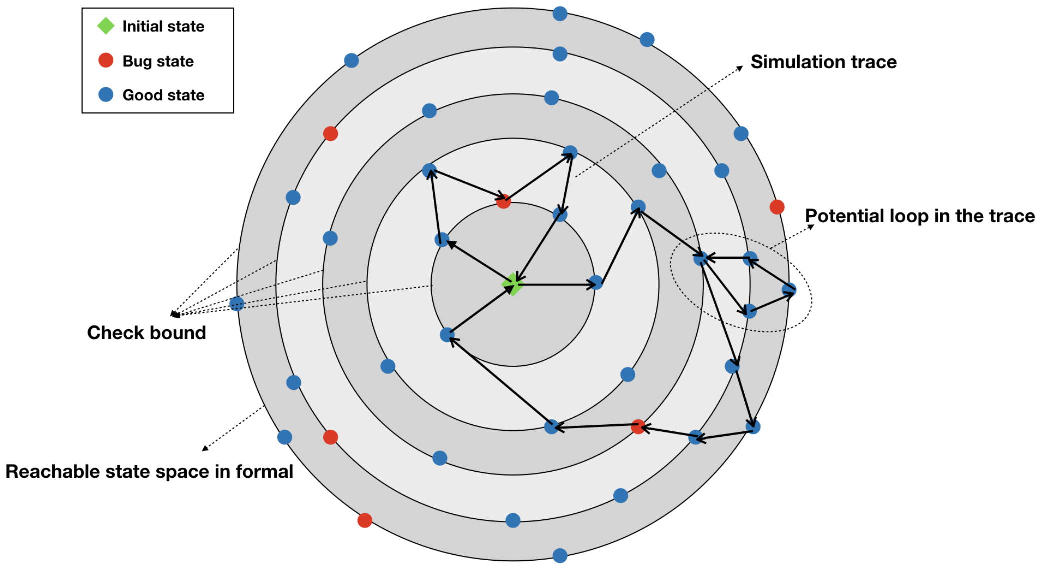


Figure 1.1: Comparison between simulation-based and formal verification

Formal model checking technique can verify Register-Transfer Level (RTL) functional correctness with a set of assertion properties without involving a testbench. Figure 1.1 shows the fundamental difference between how simulation-based verification and formal verification explore states, where each circle represents one transition step, or check bound, in formal analysis, and the simulation trace is represented by the black arrows. Unlike simulation, where the inputs (random or targeted sequences) result in the target transitioning through different states, formal verification starts from a given initial state and analyzes target behavior under user-defined constraints to verify the assertions. Therefore it can improve verification quality by exposing corner case bugs or exhaustively proving the correctness of one property, enabling shorter design and verification time. Meanwhile, with the emergence of language standards such as SystemVerilog Assertions (SVA) to define properties, formal verification has lately become a mainstream verification method. Development of more efficient algorithms, as well as the performance improvement of CPU and memory units, also push formal tools to produce meaningful results in practical time frames.

Compared to simulation, formal techniques, such as model checking, are much more comprehensive in detecting corner case bugs because they can search for all reachable states in each step. However, formal does not scale well with large designs [17], which makes it still just one part of the solution to the massive industrial verification problems. This capacity issue is the biggest reason that prevents formal from directly replacing any simulation-based methodologies or tools. Being exhaustive, formal verification can easily run into state explosion, especially for the enormous state spaces of modern designs. In addition, it is difficult for verification engineers to evaluate the results when the formal tool cannot present determined outcomes before running out of memory or exceeding CPU time limits. These issues, which usually occur in verification of large designs, require more effective and innovative techniques to extend the capability of formal engines while maintaining relative satisfying coverage.

1.2 Verification Challenges in Modern Processors

The increasing complexity of hardware designs, especially modern processors, have been posing a lot more challenges in verification. The industry has been putting more and more resources into verification than design, yet still many projects with promising new features are dropped due to the inability to fully verify the functionality within a reasonable time period.

Despite decades of research, the exponential growth in the complexity of modern processor designs over the last decades has continued to bring huge challenges into the verification fields. There are many optimizations to improve the performance of these microprocessors such as pipelining, forwarding, branch prediction, out-of-order instruction, etc. [28]. Even with the same instruction set architecture (ISA), especially for the open source ones, such as RISC-V, vendors tend to have various implementations that focus on different aspects, such as high performance or low power consumption. All the optimizations and implementation diversities stated above have introduced extra corner cases for the design, which makes verification a critical task to guarantee system function correctness and safety.

The ultimate goal is to verify the correctness of each instruction that can cover all critical internal states of the design. The most common tests conducted for such a purpose is at the assembly level, where a sequence of instructions is fed to the core, and values of memory and registers are checked. However, all of these optimizations or different micro-architectures should not be visible to the programmer in general, which means that the overall instructions should be processed one at a time in the designated order in accordance with the ISA specifications [45]. In this case, such assembly level tests cannot locate the bugs precisely in RTL related to micro-architectures, but classic RTL verification requires a deep understanding of what internal signals in the DUV behaviors are considered as legal, which brings a lot of unnecessary work to verification teams to ensure the correctness of complex designs. This major requirement shows the importance of establishing a dependable verification method to perform an RTL end-to-end ISA verification for each design which includes all the micro-architecture behavior invisible to the programmer.

On one hand, traditional simulation-based verification requires a deep understanding of a particular implementation in order to generate the stimuli to reach those corner cases. Creating such testbench is not only hard and unreliable, but also expensive in terms of reusability, as the tests would be specifically designed for one micro-architecture [45]. Furthermore, the state space of modern processors is huge, especially for 64-bit machines; therefore it is impractical to obtain full coverage by so-called exhaustive tests. One purpose of this thesis is to come up with a verification method that is able to verify the correctness of the instruction operations with higher coverage and can also be applied to any third party implementations with relatively low effort.

On the other hand, FV usually uses assertions to find violations of specification, which is suitable to describe correct behavior for the end-to-end approach. However, the formal engine has to go through nearly all the states in the design thus resulting in state explosion, which would overload the formal tool. Some corner case bugs for complex state machines can be very deep for formal techniques to catch. A better approach is to deal with the urgent need to solve the capacity issue in FV, which will be discussed in the next section.

There exist some tools that can perform automatic formal verification on the architectural or transaction level models such as FISACo in [39]. However, such tools still require human effort in defining informal specifications and the architecture models, which limits their application. In addition, the machine-generated properties are usually not human-readable, thus one completely depends on the soundness of the tools and it can be extremely difficult to debug if the properties themselves are buggy. In contrast, experienced verification engineers can write human-readable proofs and provide effective guidelines. Our intention is to achieve a balance between automation in the verification process and high-level user involvement.

1.3 Semiformal Verification: Motivation and Related Work

An integration of simulation with FV, generally referred to as semiformal verification (SFV) [1, 2, 17, 58], can have particularly preferable performance in situations where bugs are fairly deep and require thousands of cycles to execute. Semiformal verification combines the completeness of formal techniques and the capacity and ease-of-use of dynamic verification, which leverages FV in a resource-bounded approach, and thus is a key to scale FV to larger, more complex designs.

Two general semiformal techniques have been frequently investigated in the verification area. The first is referred to as augmenting simulation. In this category, dynamic simulation is first applied to reach certain intermediate state and then formal engines take over to exhaustively verify the remaining reachable states. However, formal search, in this case, would only be effective if it is triggered fairly close to the failure state. In order to relieve this limitation, some approaches of augmenting simulation include rarity-guided simulation [21] that ranks the states for later search iterations, or the automatically generated lighthouses (important intermediate states) [56] as guideposts towards the target state.

Another semiformal technique can be referred to as guided simulation. Abstraction plays a critical role in this type of methods. It is the preliminary step for formal engines to perform an exhaustive search, which partitions the reachable states into “rings” as shown in Figure 1.2 based on the depth from the target. These “rings” can be used to guide simulation towards the inner rings until it reaches the target states. However, the abstraction can sometimes be too coarse and the existence of mapping paths back to the concrete design must be taken into consideration to make sure its validity. To solve the dead-

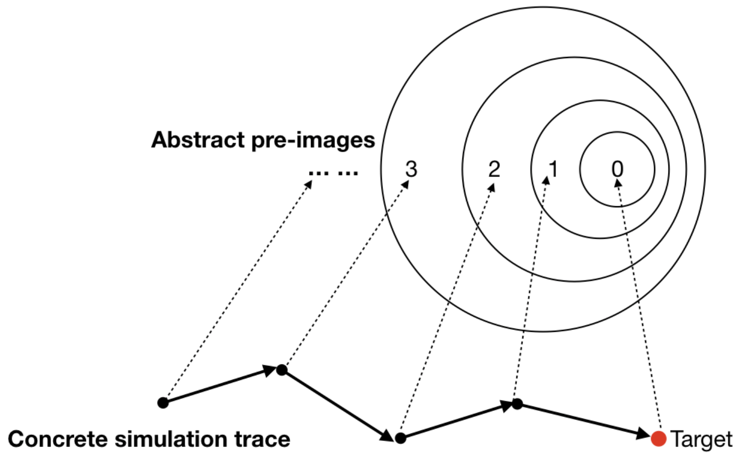


Figure 1.2: Guided simulation using abstraction

end states in a coarse abstraction, many research efforts have been put into optimization of the abstraction methodology. One solution is to keep a record of multiples states and maintain a balance between greed and relaxation [16]. Involving domain knowledge can also be effective [15], but it requires extra efforts from engineers to manually abstract the designs. Data mining has also been investigated for example in [43] but few applications were demonstrated on industrial designs and verification testbenches. Meanwhile, localization can be used to automatically refine the abstraction [18] but the abstracted models tend to grow very quickly. One promising way to solve this explosion issue is to abstract away the data-path signals to retain only control-path registers [50, 58]. Since most bugs are bundled with control issues, verification processes that focus on exploring control states increase the likelihood to find deep bugs.

Semiformal verification can be very useful in verifying industry designs due to its ability to expand the bug hunting capability of FV to detect deeper state failures in large designs. Although SFV is not as exhaustive as FV, it does achieve much larger coverage than simulation. In the next chapter, we will elaborate on the methodology explored in this thesis to apply semiformal techniques using waypoints as intermediate guides towards the target in complex designs.

1.4 Summary and Chapter Outline

This thesis studies an SFV approach that manages to alleviate the capacity issue of formal verification for complex modern designs. Unlike some previous research that only use simple RTL modules as study cases [2], this thesis investigated an end-to-end verification framework using RISC-V cores as targets. A waypoint-based SFV algorithm with no testbench required is proposed where waypoints are important intermediate states that are selected based on specific criteria. They will serve as the "lighthouses" in the formal verification process in order to shorten the proof or counterexample exploration time, thus expanding the capacity of formal tools. The main contribution of this work is to design a new automatic waypoint generation tool using machine learning techniques, which is useful for complex DUVs as it can serve as a complement to manual waypoint selection. In addition, the data set built for automatic waypoint generation part is open source, which identifies the importance of the flip-flops based on synthesized netlist information. Both manual and automatic methods will be evaluated specifically in this work for different design cases.

This chapter gave a brief introduction to related work on semiformal verification. The rest of this thesis is structured as follows. Chapter 2 illustrates the semiformal techniques adopted and the algorithm studied in this work using waypoint-based SFV methods. Chapter 3 introduces an automatic waypoint generation tool designed by applying neural networks to analyze the extracted control models. The RISC-V study cases and the corresponding experimental results are presented in Chapter 4. Conclusions and future work follow in Chapter 5.

Chapter 2

Hybrid Semiformal Verification Methods

This chapter elaborates on the waypoint-based SFV technique that is adopted and modified in this thesis to extend the formal tool capability.

2.1 Formal Verification Waypoints Definition

It has been illustrated in the previous chapter that formal verification is likely to run into capacity issues when the target state is too deep to reach within the limited amount of time and memory resources available. Corner cases usually require difficult scenarios that can only cover them when part of the design has reached a certain state, which can take a lot of cycles from the initial state. For example, in microprocessors, many interesting cases are likely to happen after the cache becomes active, and perhaps full, but it takes many cycles to write the configuration registers to enable the specific bits and many cycles to generate the memory address sequences to fill the cache.

In conventional FV, the particular state from which formal model checking starts is typically the state where the reset signal drops to be inactive. In the aforementioned example, in order to hit the interesting scenarios, the formal tool has to go through the initial sequence and then enable and fill the cache. To shorten the number of cycles necessary to get to our interesting states, we could define the intermediate state where configuration is complete as the start state for model checking. These hypothesized intermediate states are defined as "waypoints" or "lighthouses", serving as new start states, from which formal tool can search deeper until reaching the target state.

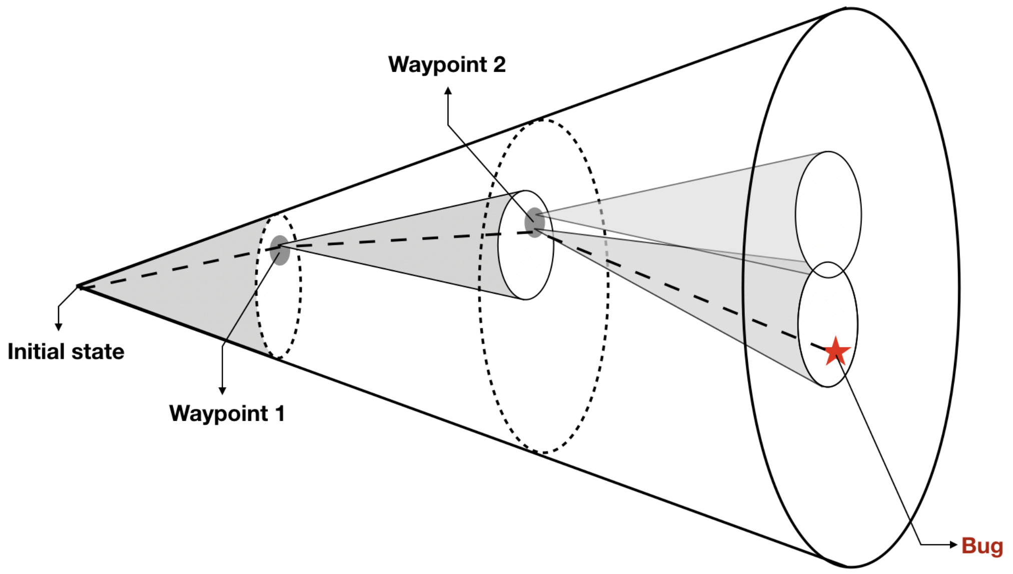


Figure 2.1: Multiple waypoints in formal model checking

Figure 2.1 illustrates the general idea of how waypoints serve as the intermediate states in SFV to find a bug. Instead of directly checking through all states for the assertions, the formal tool will only hit all or a subset of the pre-selected waypoints based on some specific search algorithms. In this scenario, two waypoints reached using simulation are used for the formal tool to re-start in order to hit the final bug state. When a certain waypoint, such as waypoint 2 in Figure 2.1, is reasonably close to the assertion failure state, it would be much easier for the tool to hit that failure if it starts from that waypoint. Without these intermediate states, the state explosion in formal verification may make our targets unreachable. By doing so, we are able to reduce the Cone of Influence (COI) at each step. Even in the cases where no bug exists, the proof time for properties are expected to be largely reduced with the premise that these multiple COIs at each waypoint can keep appropriate coverage.

One thing requiring our attention is that the reachable state set depends on the initial state from where model checking starts. Therefore, it is important to ensure that the selected waypoints are traceable from the reset state, otherwise the SFV result could run into false negatives.

2.2 Semiformal Technique Using Waypoints

The fundamental idea of applying waypoints is to use the witnesses of such state as a guide that can lead the formal proof towards the deep target state. A straightforward example of using waypoints would be a first-in-first-out queue (FIFO): if our check target is FIFO overflow, the waypoints can be set to "FIFO is 3/4 full". Or even, multiple waypoints can be selected to guide the formal tool towards the target, i.e. "FIFO is 1/4, 1/2, 3/4 full" respectively. What we have to feed the formal tool is the correct input sequence that both obeys the design specification and is able to guide the DUV towards all the waypoints.

2.2.1 Finding Proper Waypoints

Waypoints can be either explicitly selected by the design or verification engineers, or automatically generated by machines using specific algorithms, for example, based on proximity metrics. From the view of the designer, user-defined waypoints are more effective and often comes naturally. It is suggested that more users tend to select waypoints manually [2].

Meanwhile, the advantage of machine-generated waypoints lies in the fact that they do not require human intervention and detailed understanding of the design specification. One intuitive method is generate waypoints that makes the pre-condition of assert properties to be true. For example, if the property looks like *cond* | => *seq* ##1 *res*, the automatically generated waypoints would be when the antecedent *cond* is *true*, or more strictly with *seq* being *true* in the successive cycle. Other machine generated methods may include, for instance, selecting waypoints that are at a specified distance from the target states from an abstract design, or based on an architecture model of the design. The main disadvantage of the automatically generated waypoints is that they are usually less efficient than their user-defined counterparts. What's more, the generated waypoints are usually large in number, which could result in expensive computation effort in ordering and reaching each waypoint, making the later process more complicated or even unfeasible. This would unnecessarily increase the verification time and effort.

One can also combine the two, conducting a mixed strategy: some explicit waypoints can be provided by user guidance whereas the others are generated by algorithms.

2.2.2 Traversal Policy

The traversal policy of the waypoints also remains to be an interesting topic for discussion. It defines the order in which the waypoints are witnessed in the verification flow, which has a large influence on when the failure state of the assertion will be hit.

The most straightforward policy could traverse the waypoints in a specific order. More sophisticated policies usually involve regression in order to hit the closest waypoint to the current state or failure state. In the latter category policy, the search engine will need to continuously check whether current traverse meets the requirement for a certain period, otherwise, it will restart from the previous point and insert the latter path into a "blacklist" to avoid repeating the same path.

2.2.3 Waypoint Propagation Strategies

The major categories of propagation strategies for reaching new waypoints are simulation-based and FV-based propagation. The former can be further divided into random simulation and dynamic simulation. This section will elaborate and compare the three strategies in detail.

In the random simulation policy, randomly generated stimuli are fed to DUV in order to hit the next waypoints [31]. This method is popular because it does not require the user to create simulation testbenches manually. The external signals of the DUV are constrained with System Verilog assumptions for formal checks, and the randomly generated stimuli should respect all the assumptions. It is critical to make the correct assumptions to constrain the behavior in order to avoid false negatives. Another issue is that this policy requires consistent models in simulation and formal environments to integrate the two successfully. However, this is usually not the case. In order to improve the verification efficiency, models used in simulation and formal engines are likely to be significantly different. For instance, FV tools tend to replace parts of the logic with abstractions to fit their capacity. These difficult issues, especially in complex designs, limits the application of the random simulation policy in our case.

In the dynamic simulation policy, both the simulation environment and the simulation testbench should be involved alongside the formal verification environment. The simulation testbench needs to be run first, and then waypoints are selected on the simulation traces manually or automatically based on specific rules. Within the set of selected waypoints, each one serves as a new initial state for formal verification, from which formal engine starts to search the violation of assertions. The idea of the dynamic simulation policy is shown in Figure 2.2. From the coverage point of view, the simulation trace looks “thicker”, which suggests better coverage than using classic simulation methods alone. The advantage of the dynamic simulation policy is that it can model practical scenarios by designed test vectors. However, one problem remains that the FV checks are always in proximity of simulation traces, which are generally very shallow explorations. In addition, the consistent model requirement mentioned in the context of the random simulation policy also applies here. These issues make dynamic simulation policy infeasible for verifying complex hardware designs.

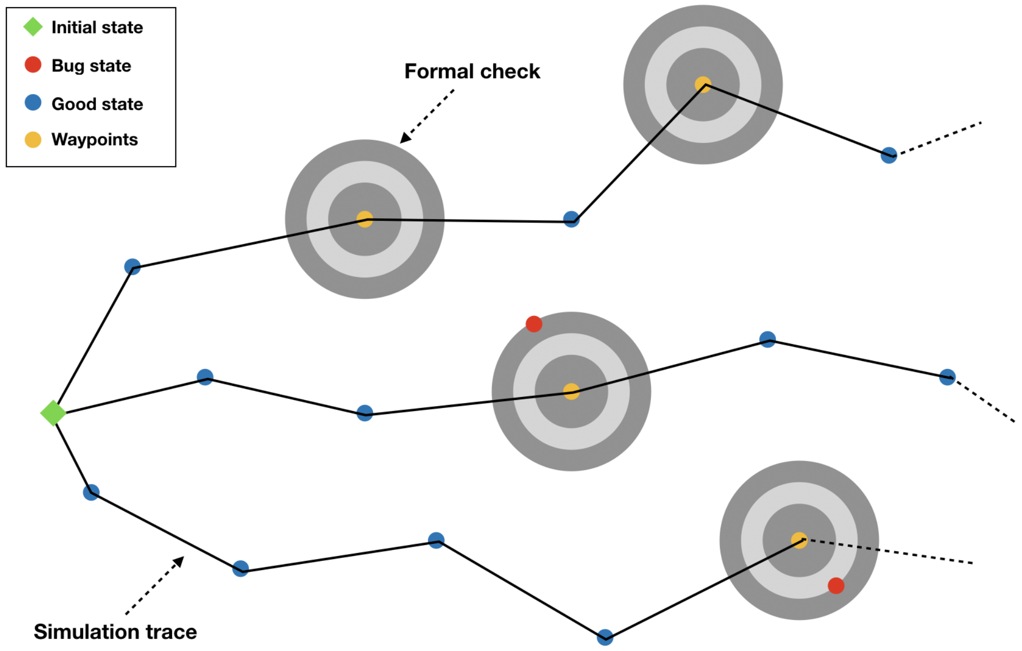


Figure 2.2: Dynamic simulation

In the FV-based propagation policy, waypoints are selected manually or automatically before starting the verification flow. Those waypoints are then treated as assertions. For example, “FIFO being 3/4 full” will be represented in SVA as an assertion of “FIFO is never 3/4 full”. The FV engines will search exhaustively within bounded steps to find a violation of the waypoints. The counterexample found is called a “waypoint witness”, which corresponds to the violation trace that can be used as input sequence to reach this witness point later. The biggest advantage of the FV-based propagation policy is that the verification environment required is exactly the same as that of classical FV except for the multiple re-runs based on the waypoints specification [2]. Therefore, there is no issue with model consistency, and no limitation on specified assumptions. The disadvantage for this policy is that the adjacent waypoints should be relatively close to each other, otherwise it will raise problems due to capacity issues of formal engines. There are commercial tools [44] already implemented certain engines to accelerate the proof by dynamically using proved assertions as waypoints for other property checks.

2.2.4 Algorithmic Waypoint-based SFV Flow

As aforementioned, one can either automate the waypoint search or use manually selected waypoints. The algorithm we will discuss in this section is based on high-level directions provided by users, but it also works for automatic process by simply modifying the first waypoint selection step. This guidance assists the SFV search towards the desired state by encoding the waypoints with SVA as either cover or assert properties (in our case, we use assertions). After discussing the algorithm, we are going to elaborate on its effectiveness using a synchronous FIFO as an example in the next section.

The outline flow chart of the SFV algorithm is presented in Figure 2.3. The sources of waypoints are the fundamental preconditions for the SFV flow. Given a set of waypoints $C_1, C_2, ..., C_n$ and the target property P we

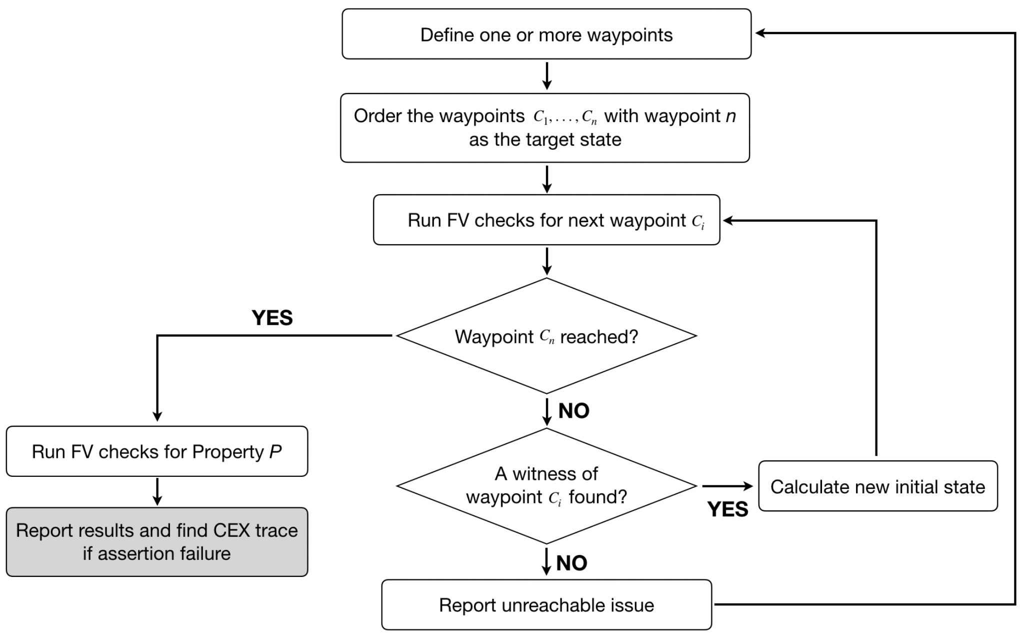

Define one or more waypoints
Order the waypoints $C_1, \ldots, C_n$ with waypoint n as the target state
Run FV checks for next waypoint C_i
Waypoint C_n reached?
YES
NO
Run FV checks for Property P
Report results and find CEX trace if assertion failure
A witness of waypoint C_i found?
YES
Calculate new initial state
NO
Report unreachable issue

Figure 2.3: Algorithmic waypoints-based SFV flow

plan to verify, the manual waypoint traversal policy will be applied. All waypoints should be pre-ordered in sequence before starting witness checks. For each waypoint C_i, the formal engine will perform a bounded model check and search for a witness of this waypoint using a corresponding assertion A_i. An assertion "failure", in this case, should be treated as a successful witness, and the counterexample trace is generated to serve as preliminaries for calculating the new arbitrary starting state for the next run. Intermediate new initial state calculation is a critical step in this algorithm because it must satisfy all the constraints for the system, otherwise, we could end up having false failures that are unreachable or not able to hit the corner case bugs. To make sure the new initial state is valid and compatible with the DUV, the counterexample trace is saved after each run to serve as the next input sequence to lead the design to the previously checked waypoint. Since the first counterexample trace is generated from the reset initial state, it is guaranteed that C_i is reachable if counterexamples are found. This input sequence is iteratively built upon successive waypoints witnesses, ensuring that all waypoints including the final target state are reachable from the original reset state. This algorithm rules out the possibility of bogus witnesses and counterexamples.

If a certain waypoint cannot be reached within a bounded time, the unreachability issue should be reported, and the waypoints will be re-ordered or re-selected. It is also possible to only roll back one step, using heuristic methods to specify the new waypoint. These issues will not be included in the discussion of this work but are worth studying.

The search steps are repeated as the subsequent waypoints to guide the FV engine towards deeper design behaviors. When the last waypoint C_n, also the target state condition, is reached, we run FV checks for the target property P and report the result.

Another thing that needs to be mentioned here is that multiple threads running in parallel would result in better performance in the search. In addition, depending on how deep the target state is, single or multiple waypoints can be selected to guide the formal check to reach states that traditional FV can never reach. As shown in Figure 2.4, the black curve shows the runtime increase versus check bound for isolated FV, while the blue curve models the

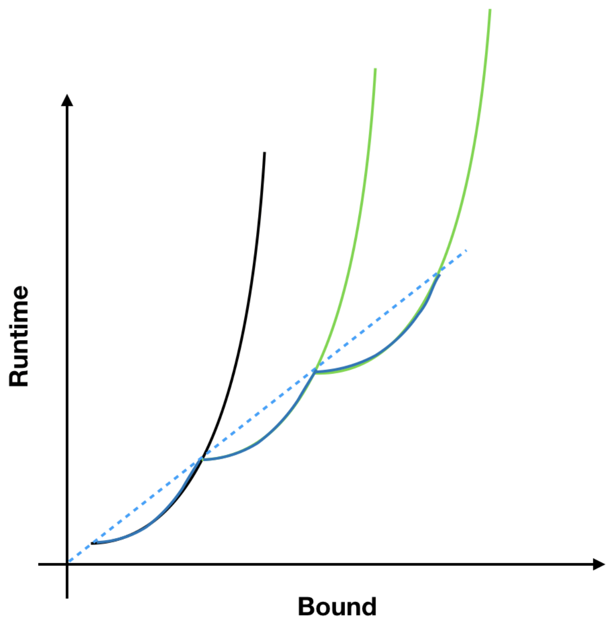


Figure 2.4: Ideal runtime reduction with waypoints-based SFV

performance of waypoint-based SFV. It is clear that SFV runtime is growing linearly, while FV runtime increases exponentially, which indicates that using waypoints can greatly improve the capacity of the formal engines.

2.3 A Toy Example: Synchronized FIFO

A simple synchronized First-in-first-out (FIFO) module (source code in Appendix C.1) is introduced here to verify the concepts elaborated above. The DUV interface is presented in Table 2.1, where FIFO_DEPTH will be gradually increased in order to touch the boundary of FV capacity in our specific verification environment: running Cadence JasperGold on a server with four Intel E5-2690 CPUs and a memory size of 8GB. In this example, Jasper Engine, Hp, Ht are used for coverage checks and Engines N, B for assertion checks.

Table 2.1: Interface of Synchronized-FIFO under verification

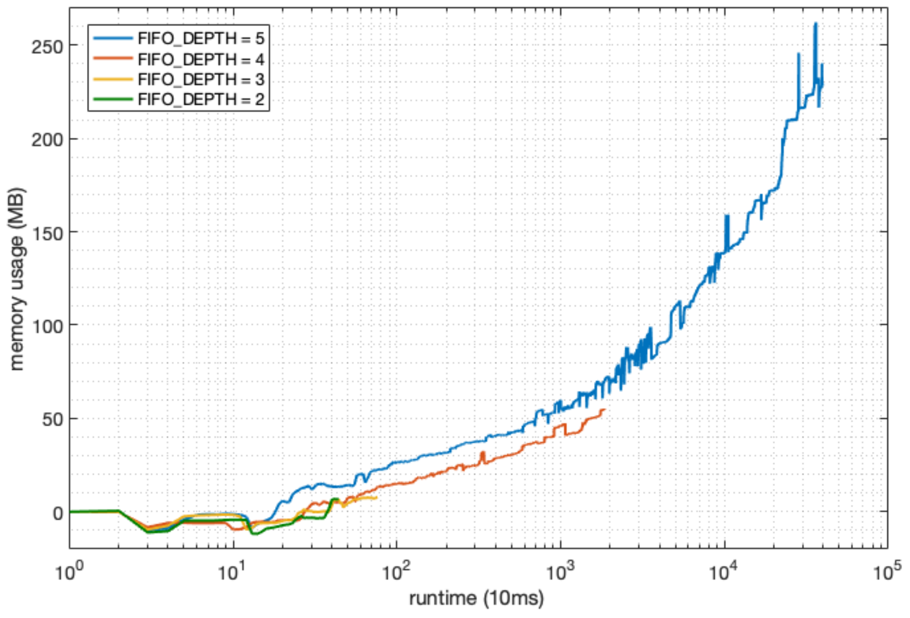
Name	**Type**	**Length**	**Description**
clock	input	1	system clock
reset	input	1	high active asynchronize reset
wr	input	1	write enable, rising edge active
rd	input	1	read enable, rising edge active
empty	output	1	high active FIFO empty flag
full	output	1	high active FIFO full flag
din	input	DATA_WIDTH	data for write
dout	output	DATA_WIDTH	data for read
DATA_WIDTH	parameter	any integer	the width data stored in FIFO
FIFO_DEPTH	parameter	any integer	2^{FIFO_DEPTH} being the number of data can fit in FIFO

To make the comparison clearer, our experiments are conducted with different values of FIFO_DEPTH and waypoint selection. Each experimental group only has one variable different from the control group. Note that although all other tasks are killed before the tests to make sure the amount of resources available is relatively stable, it is possible some resources will be occupied by other users from time to time. In order to generate results with more accuracy, multiple tests (> 3) are conducted and the results presented are the averages from these runs.

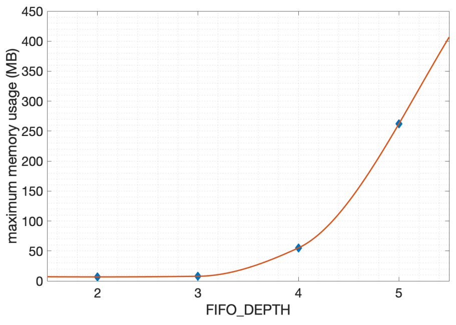
Table 2.2: SVA properties checked with Synchronized-FIFO

Name	**Type**	**Description**
ck_empty_correct	assert	check the correctness of condition for empty flag being active
ck_full_correct	assert	check the correctness of condition for full flag being active
ck_empty_once	cover	cover the case that empty flag is raised at least once
ck_full_once	cover	cover the case that full flag is raised at least once
ck_empty_to_full	cover	cover the case that empty flag is raised and then full flag is raised after certain period
ck_full_to_empty	cover	cover the case that full flag is raised and then empty flag is raised after certain period
ck_wr_num	cover	check whether write operations are executed from zero up to an upper bound (set to be three times more than the FIFO_DEPTH)
ck_rd_num	cover	check whether read operations are executed from zero up to an upper bound (set to be three times more than the FIFO_DEPTH)
ck_all_used	cover	check whether all the positions in FIFO are used at least once

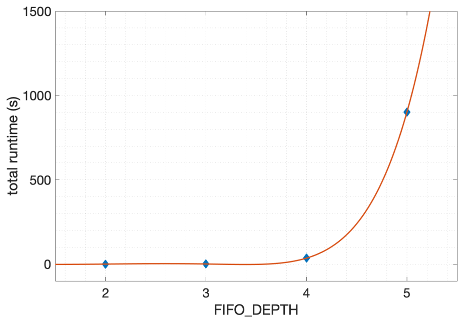
The assert and cover properties included in the evaluation of this experiment are shown in Table 2.2, where all data are raw without post-processing. Since our goal for this test is to prove the concept that waypoints can guide large designs to reach deeper states, coverage for each spot in the FIFO is the primary check on which we are focusing. In addition, assertions for correct "full" and "empty" behaviors are also included. A list of properties written for this experiment is presented in Table 2.2, the implementation of which uses auxiliary code to ensure the efficiency of SVA in the formal engine.



(a) Memory usage growing with formal check runtime



(b) Maximum memory usage



(c) Total runtime

Figure 2.5: Comparison of memory usage and runtime of synchronized FIFO with various FIFO_DEPTH

First of all, in order to illustrate the FV capacity issue more clearly, Figure 2.5 presents the fundamental idea that both memory consumption and runtime required for FV grows exponentially as FIFO_DEPTH increases. One bit added to FIFO_DEPTH shows that the number of data fit in the FIFO doubles. In the first test, all properties listed in Table 2.2 are enabled. When the size is relatively small, a formal tool can cover all properties very fast. However, if the size keeps increasing, the resources needed for full coverage are going to explode very quickly. As a matter of fact, simply increasing the FIFO_DEPTH to 8 will result in a large number of non-deterministic cover points: our experiment shows that after a 24 hour TIMEOUT limit, only 102 out of total 256 spots in the FIFO are covered with approximately 6GB maximum memory required for the computation.

Since this example is to give the readers a general idea of how waypoint-based SFV can effectively extend the FV tool capability, the experimental group will use a medium size (FIFO_DEPTH = 5) FIFO as our DUV. We only focus on pushing the FIFO to the full status because the FV tool will find the shortest path automatically for each property and those properties listed in Table 2.2 do not always share the same effective waypoints. This way, we only need to check *ck_all_used* and *ck_full_once* thereby simplifying the procedure to generate input sequence to hit the waypoints.

The result is presented in Figure 2.6 – 2.8. Three conditions are tested for evaluation.

1. Classic FV without waypoints;
2. SFV with single waypoint close to target state;

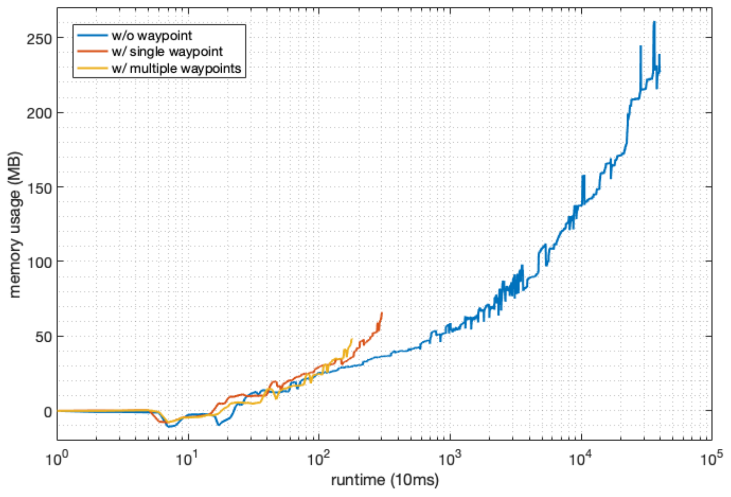


Figure 2.6: Comparison of memory consumption versus runtime among FV, SFV with single and multiple waypoints

3. SFV with multiple waypoints spread out in the design.

To validate the queue logic in stressed "full" state, waypoints are selected manually, with "3/4 full" for case 2 and "1/4 full", "1/2 full", "3/4 full" for case 3.

To clarify, the data collected includes both the process to hit the waypoints and the subsequent runs using the input sequence to cover the rest states from there. It is clear that using waypoint-based SFV can reduce the memory consumption enormously, as well as the check time for each step with the FV tool going deeper. Meanwhile, the cover properties are proved to be covered much faster using SFV than classic FV. It can also be concluded that the deeper the FV tool goes, the longer the time it needs to complete the current check. This feature implies that our waypoint-based SFV methods can

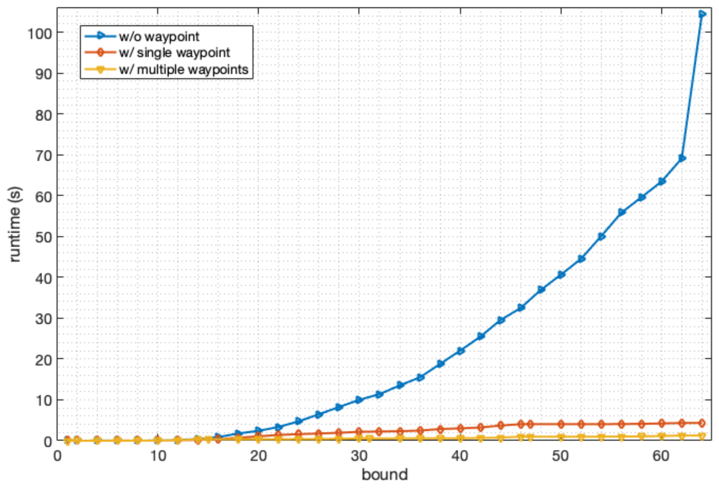


Figure 2.7: Comparison of runtime versus check bound among FV, SFV with single and multiple waypoints

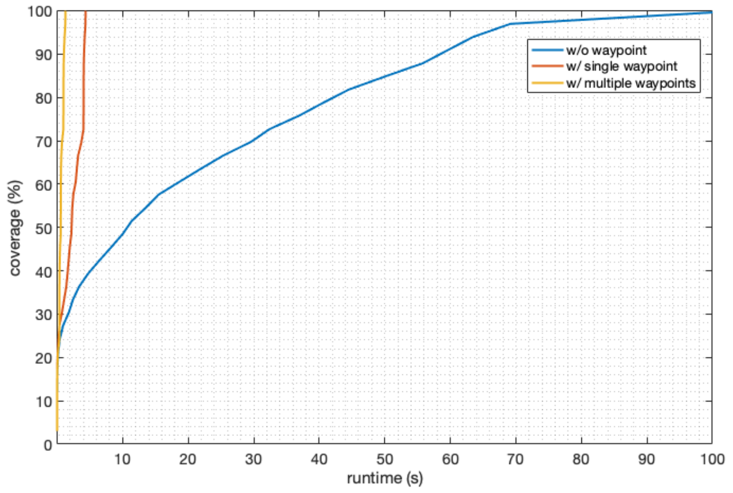


Figure 2.8: Comparison of coverage percentage versus runtime among FV, SFV with single and multiple waypoints

be especially beneficial in very large designs.

Apart from the benefits of using waypoints, it can be seen from Figure 2.6 – 2.8 that multiple waypoints can result in faster check and lower memory consumption to cover all states. However, this improvement from single waypoint to multiple ones is not as fruitful as "from 0 to 1". Considering the fact that multiple waypoint SFV requires much more effort from the user to define the intermediate states and find the correct input sequences, it is probably more effective to only involve a single waypoint if it works with the target design, whereas multiple waypoints can be applied to extremely large state machines to exploit the benefits.

Now we can draw a conclusion that waypoint-guided SFV can largely increase the capacity of the formal tools in our toy example. Even though it is a very simple FIFO module, the capacity issue easily emerges with reasonably increased FIFO depth. Considering the common designs in real applications we will be dealing with are much larger, this issue will be a severe obstacle for applying formal verification to achieve an exhaustive check. Therefore, the waypoint-guided method can be very promising in terms of its considerable coverage and capacity. In the next chapter, we will discuss the implementation of machine learning techniques to generate guidance for waypoints to assist in situations where it is difficult to define them with manageable human effort.

Chapter 3

Automatic Generation of Guide Waypoints

3.1 Introduction to Neural Networks

Neural networks have been the research hot-spots in the field of artificial intelligence since the 1980s. A neural network abstracts the neural network in the human brain from the perspective of information processing, establishes a simple model, and can be used to generate different networks according to different connection methods. It has been proved that a neural network can represent arbitrary linear or non-linear functions [35], which is usually an approximation of a certain algorithm or function in nature. In the past ten years, many research efforts have been put into artificial neural networks. Neural networks have successfully solved many practical problems in the fields of pattern recognition, intelligent robots, computer vision, speech processing, etc., that are difficult for traditional algorithms to resolve [29].

The network consists of a large number of nodes (or neurons) that are massively interconnected [49]. Each node represents a specific output function called an excitation function. A weight is assigned to each connection between every two nodes to represent the connection strength. The output of the neuron depends on the input variables, the weight value, and the excitation

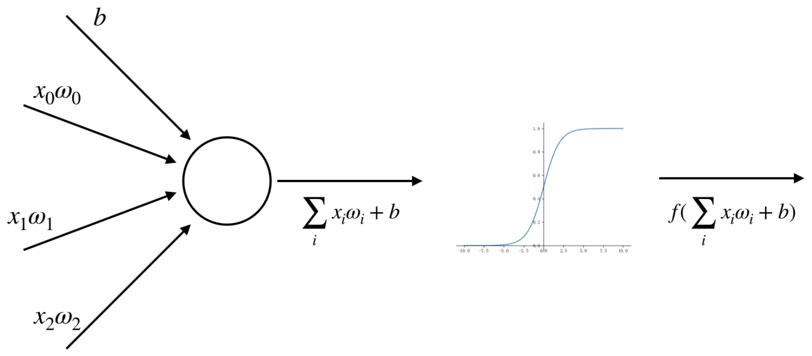


Figure 3.1: Neuron structure diagram with activation functions

function. In addition, the output of the neural network is also affected by the connection pattern of the network [49].

In the neural network, the neurons in each layer are associated with functions generally referred to as "activation functions", which decide whether this neuron will be fired or activated [48]. Figure 3.1 shows the neural network structure with each neuron associated with the corresponding activation function, where x is the input vector and ω is the weight vector from other neurons. The connection of a typical neural network will be illustrated in later section. Note that the activation functions used in this thesis are non-linear, which helps the de-linearization process of transformation in each neuron.

The neural network can be trained to learn the weight matrix that controls how variables propagate through the neurons. The optimization goal of the training is to make the prediction values as close as possible to the true value, which can be evaluated by loss function. In a regression problem, the loss function is usually measured by mean squared error (MSE), which is the average value of the sum of the squared errors between the predicted values and the true values [22].

Gradient descent (GD) is a common neural network optimization algorithm. By calculating the gradient, the parameters in the network move towards the negative direction of the gradient, so that the loss function is able to reach the minimum value locally. However, traditional GD needs to minimize the loss function on all training data. When the neural network needs to train a huge data set, the amount of computation required is very large. A compromise is to calculate the gradient of the loss function with only a subset of all the training data each time. This small amount of data is usually called a "batch", so this partition calculation method is usually referred to as batch gradient descent (BGD) [46]. BGD can greatly reduce the training time without affecting the convergence result, thus has become the mainstream to optimize parameters.

The target of our training is to minimize MSE by optimizing parameters in each layer using BGD in order to move the training results closer towards the true values.

3.2 Why We Need Machine Generated Waypoints

Chapter 2 presented a waypoint-based SFV algorithm to extend the capacity of formal verification of microprocessors. Despite many advantages of high-level user-selected waypoints mentioned above, it is necessary to investigate machine generated ones due to the growing number of large complex designs which are difficult to analyze by verification engineers without design details. In addition, many high-level hardware description languages, such as Chisel [7] are becoming more and more popular. In such designs, low level RTL design languages, i.e., Verilog or VHDL, are given up in exchange for a more convenient and better-structured design methodology. Therefore, it is impossible to apply high-level waypoint selection on large designs, especially those generated from higher level languages with machine-generated names for internal signals.

It is difficult to find an obvious relationship between the information that we can get from Verilog source code and whether this state is critical to be our guide waypoints. As introduced in the previous section, neural networks can extract hidden features in higher dimensions, thereby being more suitable for solving problems with hard-to-describe relationship between features and results like this.

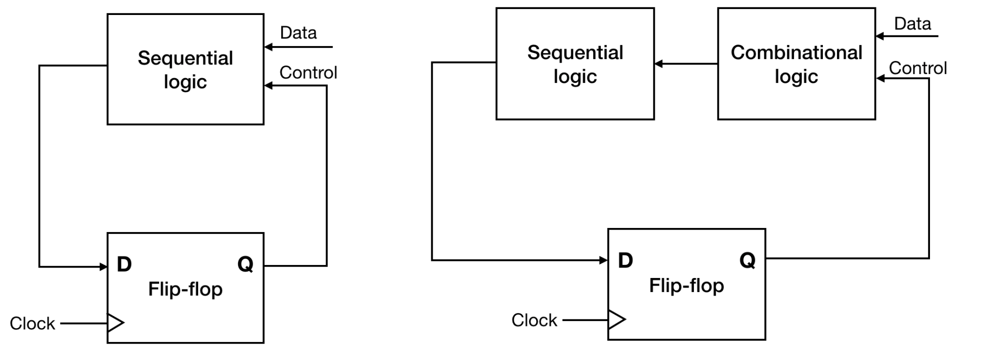
In order to provide guidance of waypoint selection in such situations, a tool based on combined Python and Tcl is implemented to perform the automatic generation of the critical flip-flops. These critical flip-flops are then taken as guides for corresponding target states in the verification plans. This will involve a Verilog syntax analyzer and machine learning techniques. The detailed algorithm will be discussed in detail in the later sections.

3.3 Automation Procedure

3.3.1 Extraction of Control Models

Control signals are usually dominant in system behavior. In addition, control logic designs are often built around finite state machines (FSMs), in which some of the states are "idle states" that keep waiting for certain trigger signals, causing the design to be very deep and complex. Therefore, extracting control models can be the core step to solve the reachability problems in complex design verification. Meanwhile, those control signals can help to narrow down the target design size, facilitating system analysis by formal tools.

Two simple models [25, 50] shown in Figure 3.2 are elaborated here to assist the heuristic algorithm in order to make a decisions on whether a signal belongs to the control set or the data set.



(a) Pure sequential logic (b) Mixed combinational and sequential logic

Figure 3.2: Typical control path model

We first consider the Verilog module in Figure 3.2a, where some flip-flops would store their own history values, whereas, for others, their current value is computed every cycle without directly referring to their previous states. The former will be referred to as critical state registers and the latter as data registers. Usually, pure sequential logic corresponds to “always” blocks with signals sensitive to the system clock in the RTL code.

Another common situation would be the case shown in Figure 3.2b, where the output value of the flip-flop affects the sequential logic indirectly by going through combinational logic first. In this case, we need to continue exploring the assignment values in combinational logic until we reach a clocking event to determine whether this flip-flop is within the critical control flow or not.

3.3.2 Training Data Set Acquisition

In order to acquire data that are related to information in the control model, we need information from the synthesized netlist, which can be found, for example, using Cadence RTL Compiler (RC). This tool can perform generic synthesis and provide timing path and connection information for inputs, outputs and registers for the design system regardless of how they are implemented in the modules. In this work, only internal flip-flops are analyzed since the interface inputs and outputs usually have clear definitions based on the design specification, thus being easy to figure out whether they are critical control signals.

There are nine features for each flip-flop we obtain from the tool as shown in Table 3.1. Some features are extracted directly from the RC tool while the others have been processed with simple arithmetic to get values in proportion.

Table 3.1: Feature vectors of the data set

Notation	**Description**
Rin	The rate of input ports connected to the flip-flop over the total number of input ports
Rout	The rate of output ports connected to the flip-flop over the total number of output ports
$>> R$	The number of pins, ports or cells that exist in the fanin cone of the specified flip-flop (must be timing start points)
$R <<$	The number of pins, ports or cells that exist in the fanout cone of the specified flip-flop (must be timing end points)
$>> Rv$	The number of all pins, ports or cells that exist in the fanin cone of the specified flip-flop
$Rv <<$	The number of all pins, ports or cells that exist in the fanout cone of the specified flip-flop
Loop	Whether the specified flip-flop (or a group of flip-flops serves different bits of the same signal) affect its own value in the subsequent cycle
Len	The number of bits for specified signal (two-dimensional memory storage will only be reduced to one-dimensional signal)
Hier	specify the level of submodule, with 0 being the top module

After data acquisition, the next step is classifying flip-flops to make up the training set. Although we would love to have two clearly distinguished categories for the signals, it is sometimes hard to completely distinguish between control and data signals. An example would be the instructions fetched in microprocessors: on one hand, they are data from memory or cache, on the other hand, they decide how the following pipeline stages work, which is more like control signal functions. Therefore, we will label each flip-flop in the training set with a score between 0 and 1 instead of directly classifying them as critical or non-critical. The continuous score is based on the control model illustrated in the previous section, but real situations are much more complex so we also involve some heuristics based on the actual behavior of the design.

Table 3.2: Design information for neural network training data set

Design name	**Number of FFs**
a25_write_back	44
ahb2wb	84
alu_with_selected_input_output	83
arbiter	4
asynchronized_spi	62
axis_master	15
axis_arb_mux_4	37
axis_slave	5
axis_switch_4	184
SSP_fifo	53
ftdi_wb_bridge	145
hpdmc	216
i2c_controller	159
i2c_master	150
jtag	70
lock	3
o8_controller	53
reed_solomon_decoder	9228
sdc_fifo	771
sequence_detector	3
SSP	134
wishbone	72

To gather sufficient data for training, feature extraction is performed on 22 designs, the number of flip-flops of which is listed in Table 3.2. They are either from `https://opencores.org` or implemented by the author; all the designs have been uploaded to the GitHub repository mentioned in Appendix A.3. The script used with the RC tool is given in Appendix A.2.

3.3.3 Finding the Critical Flip-flops

According to the aforementioned fundamental ideas, the neural network constructed for waypoint generation is illustrated in Figure 3.3. The details of each layer are as follows.

- Input layer: there are 9 nodes representing 9 features illustrated in section 3.3.2
- Hidden layer: full connection is implemented to connect the hidden layer with input layer nodes. Only single layer is used with 100 nodes. This can be extended to get more comprehensive training result. We choose the most commmonly used ReLu function to be our excitation function to perform de-linearization task.
- Raw output layer: full connection is implemented to connect the hidden layer with raw output layer. There is only one node in the raw output layer, representing the raw score from the neural network.
- Map layer: using the Sigmoid function to map the raw score $(-\infty, +\infty)$ to the (0,1) distribution.

- Output layer: final score.

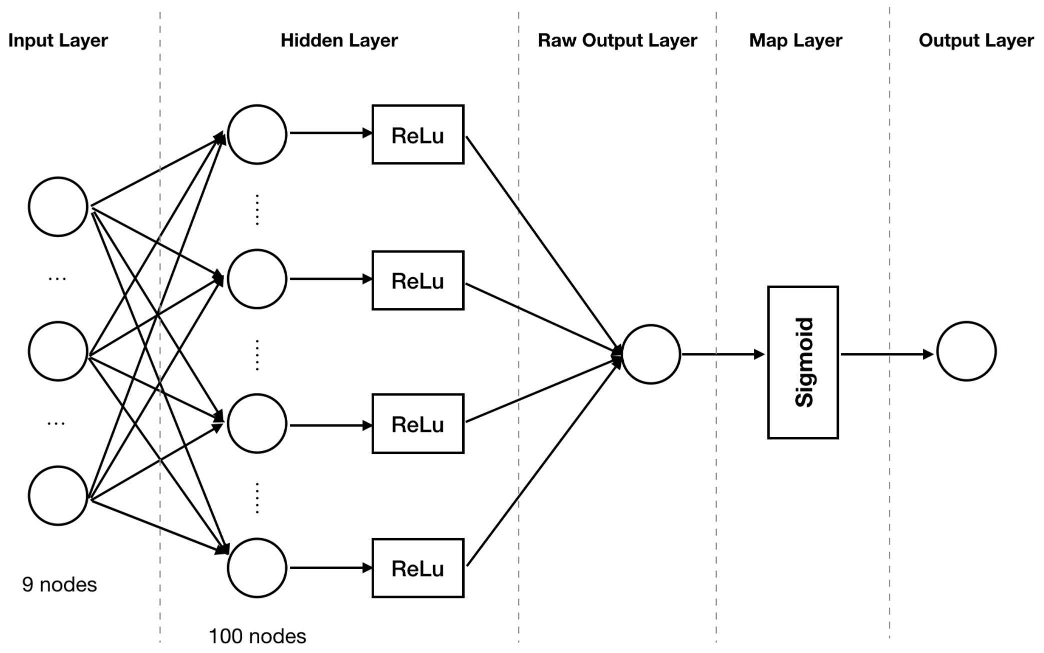


Figure 3.3: Algorithm structure of the neural network

In the training process of the model, MSE is used as the loss function, and the BGD algorithm is applied as the parameter optimization method. The equation for MSE is given in Eq.3.1, where N is the sample number in one batch, y_i is the true value, and $\hat{y}$ is the predicted value from the neural network. The specific parameters after tuning are presented in Table 3.3. Note that the regularization coefficient is set to zero because the results imply there is no over-fitting problem. In the future, more data needs to be collected to expand the data set and further reduce the loss function.

$$MSE(y) = \frac{1}{N}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{3.1}$$

Table 3.3: Neural network training parameters

Parameter	**Value**
Batch	100
Base Learning Rate	0.8
Learning Rate Attenuation	0.99
Regularization Coefficient	0
Training Round	2500
Average Moving Attenuation Rate	0.99

The entire neural network is based on the *Keras* framework (Tensorflow as backend) [12]. The parameters listed in Table 3.3 are tentative results and might not represent optimal solutions.

With the trained model, test data collected from the DUV are fed to it to get the prediction results. After retrieving the score for each flip-flop, we can set a threshold to select the critical ones for later use. This threshold depends on the requirements for practical application. We will elaborate on this in later sections.

3.3.4 Generating Guide Waypoints

We now have the score of all Flip-flops in the target design. Based on the scores, we can set a threshold to select single or multiple flip-flops as the the critical ones, which serve as a database for the waypoint generation step.

In this section, we will discuss the algorithm for single waypoint generation. Multiple waypoints generation would be similar, but with more rounds of iteration and regression in later verification because there is a higher chance that the generated waypoints are out of order or not reachable.

We will first define some notation before proceeding with our illustration of the generation algorithm. Suppose we have a model M for the FSM of our target design, we will have:

$$M = (S, I, s, \rightarrow)$$

Where S is the state variables, I stands for input variables, s is the initial state assignment, and $\rightarrow$ represents the state transition. Note that all reachable S are correlated with at least one transition $\rightarrow$.

A one-step state transition can be represented with $[S_n] \times [I] \rightarrow [S_m]$. If we call a certain S_m "CurImg", the states S_n that can enter S_m in one clock cycle with corresponding I is defined as "PreImg"; if S_n is "CurImg", the states S_m that the "CurImg" can transfer to in one clock cycle with corresponding "I" is defined as "NxtImg".

Before starting the analysis of the Verilog source files, we need to determine the CFF pool for critical flip-flops out of all the flip-flops that we are interested in. From this stage, all operations are conducted for the flip-flops in the CFF pool.

For a specified state $[S_i]$, we can define the current state as "CurImg" according to the FSM transition graph. We define its "PreImg" as those states that can directly reach CurImg state without passing through an intermediate state, and "NxtImg" as those can be directly transited from CurImg via no intermediate state. All CFF should store some values in the specified CurImg, and those flip-flop values can be calculated by parsing the information of RTL source code in the "CurImg", "PreImg" and "NxtImg".

As for the RTL code, we will use $Cond$ to represent a condition statement, RV for right value of the assignment and LV for the left value of the assignment. In addition, we need to pay attention to NOT logic, i.e., ! and $\sim$ symbols, of the interesting flip-flops since the reverse logic affects how we pair those values in the assignment. This feature will be used for waypoints automatically generated in the next step.

With the notation clarified above, we can provide the Algorithm 1 in pseudo code for waypoint generation based on the selected CFF and the parsed RTL code.

Assume CFF is not an empty set, a new flip-flop list X is initialized with all elements in CFF. Besides, $Waypoint$ set is initialized as an empty set for later appending. The generation loop (Line 3) will evaluate each element x in the list X in sequence. For each x, its "Nickname" is first searched through the RTL code based on two criteria: 1) when x is the only RV in the assign statement, the corresponding LV is defined as a Nickname of x; 2) when x is directly connected to an input or output port of other modules, that port is defined as a Nickname of x. The Nicknames should be appended to list X for the exact same search in later looping starting Line 10 and Line

Algorithm 1 Waypoints Generation

```
Require: $CFF \neq \emptyset$
 1: $X \leftarrow \forall x \in CFF$
 2: $Waypoints \leftarrow \emptyset$
 3: for $x \in X$ do
 4:     if $x$ found as the only* $RV$ in combinational assignment then
 5:         append $LV$ to $X$
 6:     end if
 7:     if $x$ is directly connected to submodule ports then
 8:         append the connected port to $X$
 9:     end if
10:     for $\forall x \subset$ Combinational Logic do
11:         if $x \in \forall Cond$ then
12:             append $(Cond)$ to $Waypoints$
13:         end if
14:     end for
15:     for $\forall x \subset$ Sequential Logic do
16:         if $x \in \forall Cond$ then
17:             append $(Cond)$ to $Waypoints$
18:         end if
19:         if $(x \in \forall RV) \cup$ (corresponding $LV \in CFF$) then
20:             append (RV ##1 LV) to $Waypoints$
21:         end if
22:     end for
23: end for
24: return $Waypoints$
```

15. The next step is analyzing the RTL code to generate waypoints. First, we go through both combinational logic (=) and sequential logic (<=) to check whether x belongs to the conditional statement, such as *if* and *case* (Line 11 and Line 16). If yes, the entire condition logic will be added to the *Waypoints* set. Besides, there is one more case for sequential logic where we can select waypoints: if x belongs to any RV, and the corresponding LV is

also within CFF set (Nickname excluded), then the sequence RV ##1 LV should be added to the $Waypoints$ set (Line 19). When all elements in list X are analyzed, the loop ends and the $Waypoints$ set is returned. In addition, the algorithm description with symbol * can be altered to meet specific needs. We will give an example in Chapter 4.

Note that in an application of the generated waypoints, we need to add '!' if we use SVA to find the trace. No such reverse logic is needed if we use coverage properties.

It is worth mentioning that another important issue in the automatic process is how to order the generated waypoints. This order process is completed manually in this work. We suggest that future work can apply heuristic regression referring to fanin and fanout metrics to further automate this step.

3.3.5 Result Evaluation

To evaluate the proposed automatic waypoint generation methods, we apply the same synchronized FIFO example used in Chapter 2.

First, the neural network model is trained with the collected data set illustrated in Section 3.3.2 using *Keras*. This model is then used to score each flip-flop, which we refer to as "machine-predicted score". Meanwhile, we manually score each flip-flop as a ground-truth, which is referred to as "user-defined score". Figure 3.4 shows the squared errors between corresponding user-defined score and machine-predicted score. The machine scores increase alongside the horizontal axis from left to right, but the coordinate values are not evenly distributed because we want to show each spot clearly. As shown in the figure, the squared errors at both ends are quite small while the median ones are relatively large. This result is acceptable because our goal is to find the critical flip-flops with a very large score. Those flip-flops belong to data path have small scores in both user-defined and machine generated cases, which are ensured to be eliminated. The only concern remaining is the middle part with large squared errors. However, scoring itself is actually ambiguous, especially for these hard-to-define flip-flops with median scores. Since even humans cannot give a clear classification, we can take them out of our consideration. The original data is attached in appendix B.2.

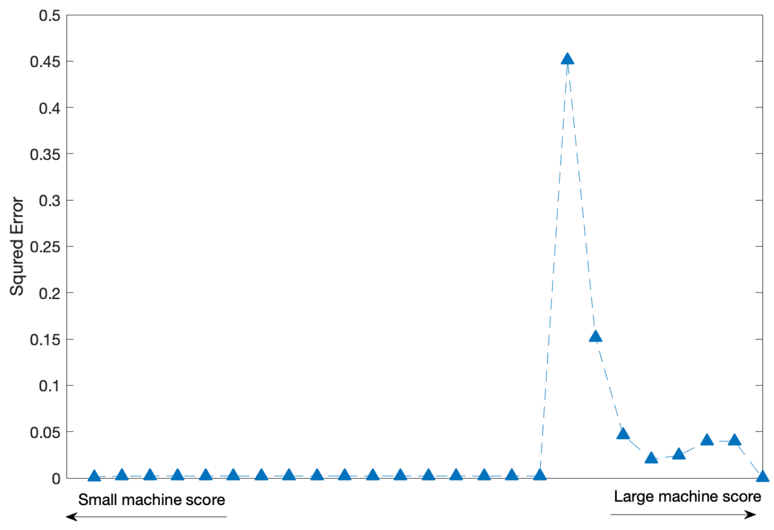


Figure 3.4: Squared error between user-defined scores and machine-predicted scores

A threshold should be set manually to pick the CFF set for the next step. The decision on threshold depends on practical application, but we suggest a basic rule that selected CFF should not include those with large squared errors. A fundamental way to judge whether the predicted scores are rational is to evaluate the selected CFF based on the original RTL design.

In this example, we set 0.8 as the threshold so that:

$$\text{CFF} = \{\text{dffw1, dffw2, full\_reg, wr\_reg}\}$$

The detail of these signals can be referred to Appendix C.1. This result is consistent with our arbitrary definition of control signals in the synchronized FIFO module. These critical flip-flops will serve as the preliminary dataset for the waypoint guide generation in the next step.

We develop a Python script to perform the waypoint guidance generation for each flop in the CFF set. It also returns the interesting line in the RTL code in case users may want to check the design and introduce some human intervention in deciding waypoints. The result is presented in Table 3.4.

Table 3.4: Automatic generated waypoints guide

CFF	Waypoints	line
dffw1	(dffw1) ##1 (dffw2)	21
dffw2	null	null
wr_reg	(rd_succ == wr_reg)	91
full_reg	(~full_reg)	99

The next step is applying the one or multiple waypoints from Table 3.4 to SFV flow. However, can be noticed that these waypoints could guide the system a different direction which may not include our target states. In addition, the memory resources and time required to automatically select and order multiple waypoints can be too high. Therefore, combining automatic generation results and artificial analysis may be more effective to reach the target state.

The test conducted applies a more effective direction which lets the user select and order the waypoints for our target from the guidance list. In addition, higher hierarchical conditions can be added to the waypoints to specify them in more detail. This is also a good example of the reason why human intervention is preferred even with the generated guidance list. Therefore, the waypoint can be selected as *(∼ empty && (wr_reg == rd_succ))* to help to hit the target state faster. Besides, the same environment, running Cadence JasperGold engine Ht & Hp on the server with four Intel E5-2690 CPUs and memory size of 8GB, is used in this test. To obtain better comparison results, we take the same coverage property, *ck_full_once* and *ck_all_used* in Table 2.2, as our target states and test the amount of the time and memory needed for the formal engine to reach them.

Table 3.5: Comparison between FV and SFV with automatic generated waypoints

	total runtime /s	memory consumption /MB
w/o waypoint	149	251.67
w/ manual waypoint	4.35	65.96
w/ automatic waypoint	78	65.68

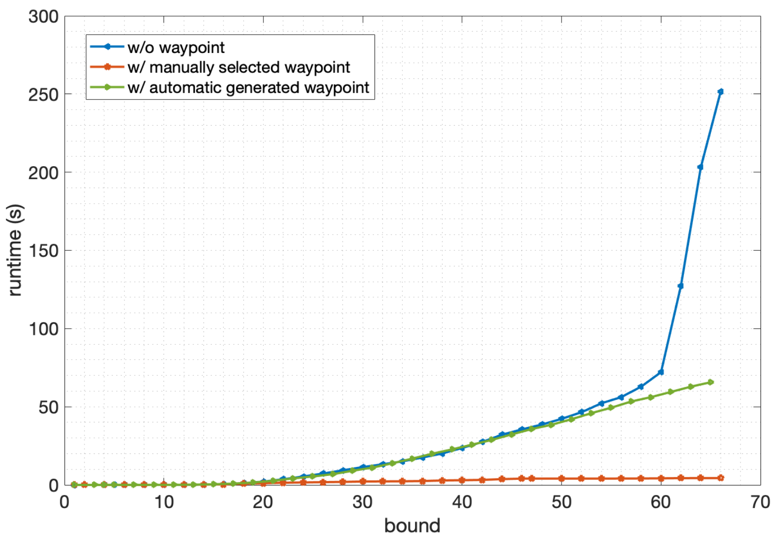


Figure 3.5: Runtime growth versus bound of FV, SFV with manually selected, and automatic generated waypoints

We compare run times between FV and SFV with automatically generated waypoints using the synchronized FIFO with FIFO_DEPTH of 5. The result is presented in Figure 3.5 and Table 3.5, where total runtime and max memory consumption using automatic generated waypoints are significantly improved by 47.7% and 73.9% respectively compared to isolated FV. The reduction in runtime using SFV with waypoints generated by the proposed algorithm appears to be from nearly exponential to linear growth. However, it is undeniable that the manually selected results presented in Chapter 2 appear to be more effective than this automation algorithm, as shown in Figure 3.5 in terms of runtime growth. However, in the cases where the DUV design details are ambiguous or too complex, this automation method can still be of great benefit.

3.4 Other note: Coverage Metrics

Besides waypoint generation automation, there are many other applications for the identified critical flip-flops, such as fault tolerance and coverage. In this section, we will discuss the application from the aspect of coverage metrics based on the critical flip-flops found from the aforementioned algorithm.

Evaluation of whether verification results give sufficient confidence regarding the correctness of the design depends heavily on the coverage metrics. Most of the mainstream verification flows use coverage metrics such as line coverage, signal coverage, branch coverage, etc., which can be generally classified into two categories: code coverage and function coverage [3]. However, many of these are ambiguous and incomplete for situations where multiple decisions on state transitions are made together. The thesis research in [25] designed an automatic coverage directives generation tool by analyzing RTL written in Verilog HDL. However, the coverage properties written in SystemVerilog bear the advantages that they can be integrated with most simulators and formal tools easily. One potential issue is that the properties generated from large designs require a huge amount of computation. The main drawback is that this process may take an impractical time to run. Even it can generate all properties within a tolerable time, the possibility of having unreachable coverage properties increases due to the large number of properties generated, and it is hard for humans to distinguish whether the unreachable coverage is due to the problem of automation tool or the deficiencies in the verification plan. Therefore, we could use the method illustrated in this chapter to find the CFF as a subset of all registers, which helps the coverage directives generation tool to focus only on the critical control flow. Even excluding the automation tool, these critical flip-flops can still serve as a good guide for manual coverage metric design.

In addition, other similar algorithms can also be applied here to automatically generate the coverage metrics guides. The only difference lies in how to select the CFF pool. The number of flip-flops selected can be very flexible since we provide a continuous scoring prediction instead of an absolute classification. Designers can choose any threshold based on their own project requirements.

Chapter 4

Experiments and Results

4.1 Application Guidelines

Before looking into the study cases, we will first specify the cases where the application of waypoint-based SFV flow would be more suitable and beneficial.

The SFV flow proposed in this work is able to largely improve the performance of verification in large design, where classic FV easily run into complexity issues. The keypoint in our SFV flow is to select the proper waypoints. For example, in microprocessor verification, we could bypass the long initialization sequence through the peripheral bus to configure all the architectural registers. Another case, also for microprocessor verification, is that we can select waypoints based on the pipeline stages in order to reduce the size of the COI. For instance, if we take the state right after decoding as waypoint, we can focus on the specific instruction for verification.

In this Chapter, experiments on RISC-V cores will be conducted and the corresponding performance will be evaluated.

4.2 RISC-V Verification Setup

Chapters 2 and 3 introduced the SFV algorithm with manually selected and automatically generated waypoints in detail and demonstrates their effectiveness with a simple example of a synchronous FIFO. Since the capacity issue that we are trying to solve is more likely to appear in large complex systems, we will take RISC-V processors as our test objects in order to verify the effectiveness of the proposed algorithm in such a real application. In this section, we will present the verification flow based on the proposed SFV method and the corresponding results of the study cases.

In order to verify the RISC-V cores, it is better to build an expandable formal verification framework based on the RISC-V specification to conduct classic FV as a control group. The framework used in this thesis is based on the work of [54], the structure of which is presented in Figure 4.1. The RISC-V Formal Interface (RVFI) can serve as the communication ports between the DUV and the SVA checkers. The *rvfi_wrapper* is connected directly to RVFI, which provides not only the standardized wrapper, but also the input constraints that can mimic the correct bus behavior of the design. These input constraints will change based on a specific design. Otherwise, incorrect input constraints can result in weird illegal behavior and false negatives in verification, which is also an important research topic but will not be discussed in this thesis. The primary verification targets are the instructions based on the supported RISC-V ISA specification. The *rvfi_insn_check* module performs this task by having an expandable interface that can be connected to various modules designed specifically for describing valid behaviors of each instruction. The formal verification top module *rvfi_testbench* integrates all the SystemVerilog (SV) modules that have property checkers or auxiliary codes. Note that this *rvfi_testbench* can also connect to other checkers with different configuration macros. These macros must be defined before loading any RVFI files correctly. In addition, this framework can be set up for bounded model checks or unbounded verification depending on factors such as the overall complexity of the core and verification requirements. In our test cases, we configure framework for bounded model checks by having a variable name "check". This check depth is determined by the number of cycles needed for each instruction based on the specific pipelining design. Only when the check is asserted will the tool start the checking process.

Since this framework is written in SV, it can accommodate any formal tools that support SV verification with minor changes. The experiments conducted in this thesis uses Cadence JasperGold Formal Property Verification APP [44] on a large server. However, similar to the issue mentioned in experiments on the synchronized FIFO, the server we use is shared so the data collected may have some variance due to multiple uses on the shared system. Therefore, multiple tests have been conducted to present the results on an average base to make the outcoming data more reliable.

In our work, we will concentrate on the verification of the functional correctness of the instruction set. The primary assertions checked for each instruction are listed in Table 4.1. To illustrate, property 3, checking the

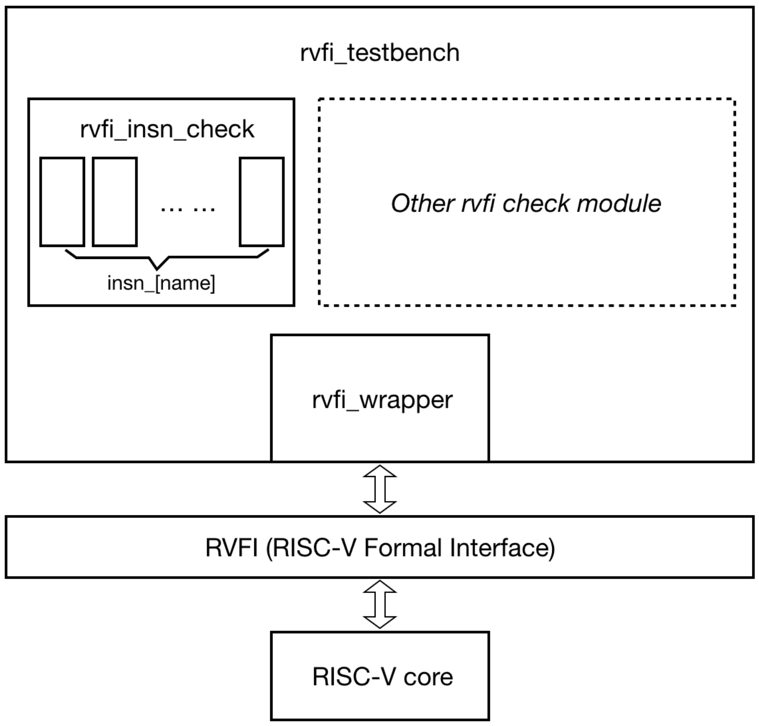


Figure 4.1: RISC-V formal verification framework diagram

correctness of R_d, write data is under the condition where data in R_{s1} and R_{s2} are unconstrained. Note that not all of the listed assertions would be covered in each instruction check.

Due to space limitations, we would only present our detailed elaboration on one RISC-V implementation, PicoRV32.

Table 4.1: Assertions checked in formal verification on RISC-V instructions

NO.	Description
1	Conditions for entering TRAP vectors
2	R_d address
3	R_d write data
4	Next PC address
5	R_{s1} address
6	R_{s2} address
7	Data correctness in memory read access
8	Data correctness in memory write access
9	Correct alignment in compact ISAs

4.3 RISC-V Core: PicoRV32

4.3.1 Introduction to Experiment

PicoRV32 is a CPU core that implements the RISC-V RV32IMC ISA. It can be configured as RV32E (embedded), or any combination of RV32I (integer), RV32C (compact) and RV32M (multiplication) [53], and an optional configuration to support IRQ using a simple customized ISA [55].

The average cycles per instruction (CPI) varies among different instructions, but is usually around 4. The CPI numbers for the individual instructions can be found in Table 4.2 with the register file configured in dual-port mode. Based on such information, the check depth is set as 20 so that the processor is guaranteed to retire the current instruction (except the multiplication and division) under check.

The verification experiment steps are listed as follows.

1. Make a detailed verification plan on what assert and cover properties

Table 4.2: CPI numbers for the individual instructions

Instruction	CPI
direct jump (jal)	3
ALU reg + immediate	3
ALU reg + reg	3
branch (not taken)	3
memory load	5
memory store	5
branch (taken)	5
indirect jump (jalr)	6
shift operations	4 - 14
multiplication	40 - 72
division	40

should be checked.

2. Instantiate the formal verification framework to accommodate for PicoRV32.

3. Specify waypoint(s) for the DUV (manually or automatically generated)

4. Run FV on the instruction set with waypoints as assertions. Collect data from FV checks, and find a trace to each waypoint.

5. Run SFV on the same instruction set with customized input sequence to hit waypoints. If this is the last waypoint – our target state, collect data from SFV and compare with FV results; otherwise, go back to step 4.

4.3.2 Verification Ability Improvement

The experiment results for PicoRV32 study case are presented in this section. Both manually selected and automatic generated waypoints will be assessed specifically.

In order to quantify the computing performance improvement, we will use memory and runtime as our performance metrics. Many studies only focus on one aspect while sacrificing the other. Instead, neither would be ignored in our study cases and each parameter will be presented in detail. If less runtime and memory consumption is observed, it suggests this method has more capacity than isolated FV, or in another word, cover more states within limited resources. In our experiment, we exclude the situation where stored values can get flipped by interference such as electromagnetic interference (EMI), that is to say, our verification only focuses on reachable states starting from the reset state in the FSM.

4.3.2.1 User-defined Waypoints

In this section, the waypoint is manually selected as the state where the instruction under check has retired, indicating the state where all computation is completed.

The same verification environment as the FIFO example in section 2.3 was first used with four Intel E5-2690 CPU and memory size of 8GB. However, JasperGold is unable to provide determinate FV results without waypoints with the limited resources on this server. This result indicates that classic FV

Table 4.3: Verification result of PicoRV32 AND instruction on server S

	Runtime/s	Memory Usage/GB	Line Coverage/%
FV result	386.7 +	page fault	-
SFV result	88	1.02	87.41

checks requires too many resources since the tool reports a page fault issue due to low memory. Table 4.3 presents the results of the check for the AND instruction using both FV and SFV methods. The '+' sign in the FV runtime entry shows the time that the tool reports a "page fault" when the check has not been completed yet. Although we are not able to statistically analyze such results, it is obvious that our waypoint-based SFV is superior to classic FV in terms of both runtime and memory consumption.

To better quantify the comparison of verification ability between FV and SFV, we move to the "big" server with 32 Intel Xeon E5-2690 CPUs and memory size of 378.47GB available. In this condition, we are able to complete both FV and SFV experiments for all instructions listed on RV32ICM specifications [53]. Both FV and waypoint-based SFV are applied to the PicoRV32 core to verify the functional correctness as indicated in Table 4.1 of each instruction. Note that only the control paths are verified for RV32M, which will be elaborated on later.

Comparison in terms of runtime and memory consumption is presented in Figure 4.2. As can be seen from these figures, the improvement is significant: it is obvious that both runtime and memory usage applying the SFV method is much less than those applying the FV method. In fact, our statistical results show that the average improvement of runtime from FV to SFV is **79%**, and the reduction of memory usage is **94%**. At the same time, the line coverage and signal coverage [3] provided by JasperGold remain the same for the two methods, indicating that introducing traces to hit waypoints does not have a significant effect in basic coverage measurements.

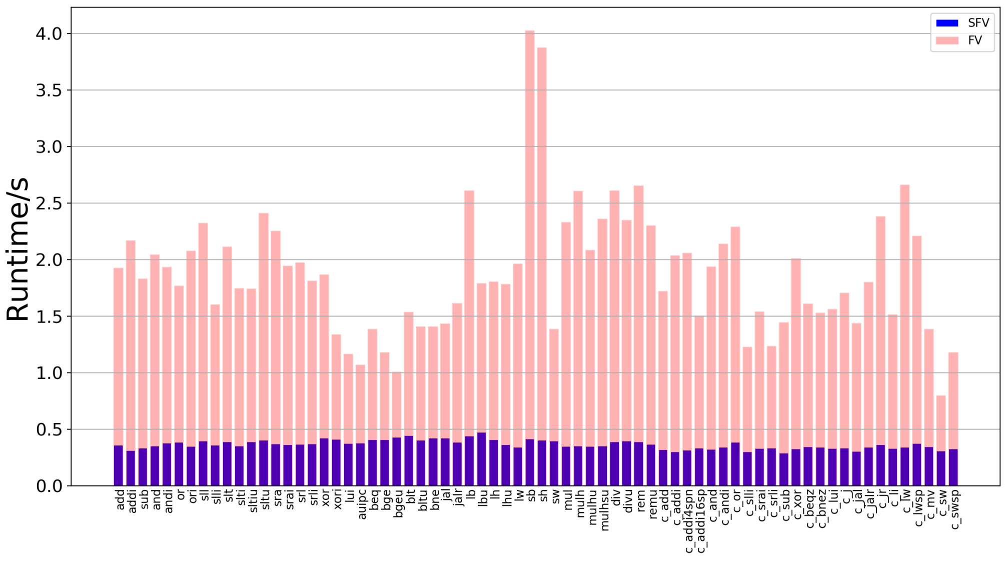
We will now discuss verification for RV32M specifically. Due to its complex arithmetic operations, It will take a lot longer runtime if we include the data path in verification. To solve this issue, we assume the MUL/DIV calculation is correct, similar to the idea of black-boxing, and only check the addresses of R_{s1}, R_{s2} and R_d (source and destination registers), and the control signals send to the control unit to establish that the instruction has been correctly executed. If more comprehensive verification is required, the function unit can be verified separately.

To conclude, the proposed SFV method is proved to be very promising in expanding formal verification capacity on such large designs.

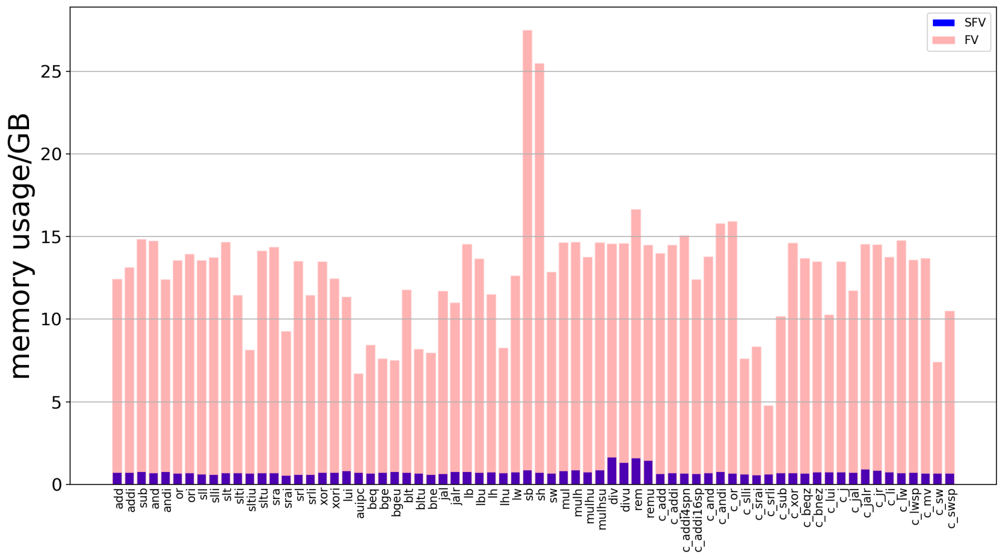
The original data is given in Appendix B.1 as a reference.

4.3.2.2 Automatically Generated Waypoints

The previous section shows that applying high-level user-defined waypoints in our SFV method can bring us large improvements in verification ability compared with classic FV. However, though the manually selected waypoints are effective, they require a large amount of time and effort to analyze



(a) Runtime



(b) Memory Usage

Figure 4.2: Comparison of memory usage and runtime of RISC-V ISAs

the system and select the proper intermediate states. In this section, the automatically generated waypoints will be assessed with the PicoRV32 core in order to prove the effectiveness of the automatic waypoint generation algorithm proposed in Chapter 3. The results in this experiment will be compared with the case using manually selected waypoints.

With the model already trained by the neural network and the data gathered from the synthesized PicoRV32 netlist, we follow the same routine introduced in Chapter 3 and get the score predictions of flip-flops in the PicoRV32 core. The original data is given in Table B.3 attached in Appendix B.2. Based on the score set, the threshold for CFF is set to 0.8, which delimits 5 critical flip-flops in total. These CFFs are then passed to the waypoint generation tool. The guide report for each CFF is presented in Appendix B.3, where the number after "L" is the corresponding line in the Verilog code analyzed by the tool, and the logic expressions after them are the generated waypoints.

As stated in Chapter 3, human intervention is involved to select the suitable waypoints based on this guide. In this experiment, the waypoint below is used to guide the formal engine (no hierarchical information added):

(resetn && cpuregs_write && latched_rd)

With the same environment as metioned in the previous section, we run formal checks in JasperGold using 1) FV without waypoints, 2) SFV with the manually selected waypoint, and 3) SFV with the automatically generated waypoint. In order to compare the performance of these three cases, we visualize the increase in the memory usage as a function of the runtime of the formal tool pass in Figure 4.3 with the *LW* instruction as an example. Noted that the data is not consistent with that in Appendix B.1 because this is from one experiment result, while the original table shows the average values.

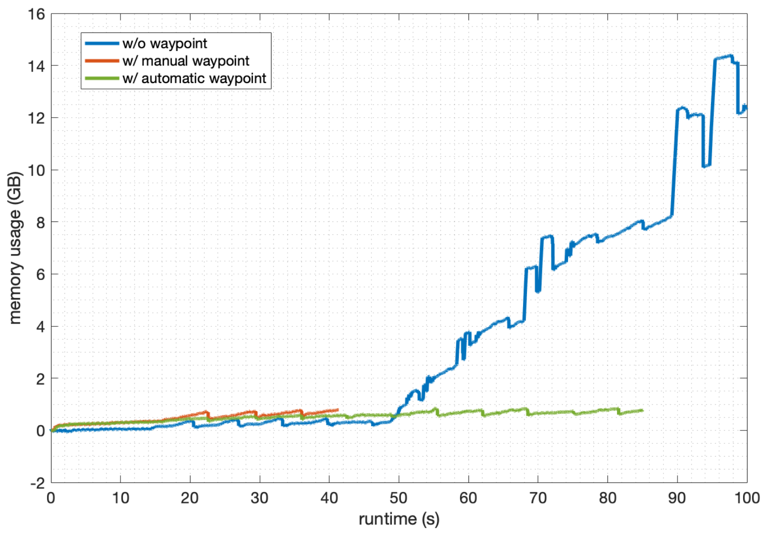


Figure 4.3: Memory usage versus runtime of LW instruction comparison

The results show that using the FV method, the memory usage is nearly exponential with the increase in runtime, and that of SFV is close to linear. Besides, the result is similar to the synchronized FIFO example elaborated in Chapter 3: though both are able to improve the verification performance, the user-defined waypoint is more effective than its automatic generated counterpart. Due to space limitations, corresponding curves of other instructions, which all show similar performance changes, will not be presented here. However, one problem that has not been resolved is the SFV runs of the DIV set of instructions. This may be due to the fact that the waypoint we defined is not necessarily on the path towards the target state; this is an area for future experiments and analysis. In addition, since this DUV is bug-free, we intentionally added some bugs related to instruction functions. Those bugs have been captured with our formal verification framework, and the SFV results show that bugs are covered with less overhead. To illustrate, the number of cycles needed to find the bug is the same as the number of bounds reduced to reach the target state. However, we will not elaborate on these experiments due to space limitation.

4.4 RISC-V Core: Rocket

4.4.1 Introduction to Experiments

Rocket is a 5-stage in-order scalar core generator implemented with Berkeley's Chisel, which supports the RV32G and RV64G ISAs. Its pipeline structure is shown in Figure 4.4. It has one memory management unit (MMU) that supports virtual memory, a non-blocking data cache, and also a branch prediction unit. The source code for the Rocket project can be found on the GitHub repository *freechipsproject/rocket-chip*.

The main reason why we brought this Rocket core into study, as mentioned earlier, is that its RTL code is not human-readable because it is generated by Chisel. Besides, the generated Verilog file is huge with nearly 200,000 lines. Considering these features, we would like to make an assessment on

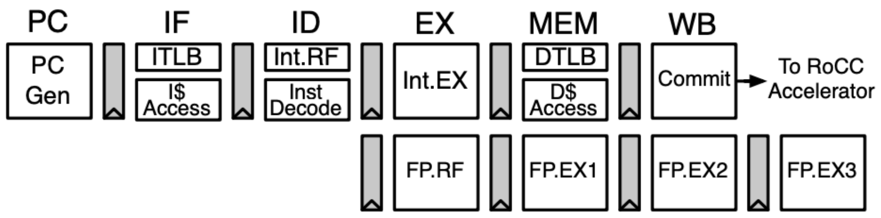


Figure 4.4: The Rocket core pipeline[5]

whether the machine-generated waypoints can be of reasonable value in this case.

The verification flow for Rocket core is introduced as follows (similar steps with the PicoRV32 study case are briefly mentioned).

1. Generate RTL code from the Rocket core design implemented in Chisel.
2. Make the verification plan and run rvfi_insn_checks using the FV method.
3. Find the initial sequence to configure the core correctly (this is different from the previous example, we will elaborate on this in the next section).
4. Run the automatic waypoint generation tool developed in this thesis for the Rocket core and select waypoints for SFV test.
5. Find the traces to the specified waypoints using assertions.
6. Run waypoint-based SFV on the supported ISA set with the new trace to hit the waypoints. Repeat this procedure until the last waypoint – the target state – is reached.
7. Analyze the results.

4.4.2 Initial Configuration Sequence

The first round test is conducted the same way as that used for PicoRV32 case. However, when we simply start the FV checks from the reset state, all properties would find counterexample traces at the very first cycle after the reset cycles. These counterexamples are obvious false negatives after investigating the traces provided by JasperGold.

In order to produce correct and comprehensive processor functions, it is often the case that one needs to initialize the core first by configuring important registers to proper initial states. As a matter of fact, this actual initial state is not the same concept as what we have been referring to as the ones that determined by the reset signal. Instead, it should be considered as a waypoint, representing the state where the processor is ready to work.

In Rocket, the program counter (PC) is always initialized to 0x2000. Therefore, we need to make sure the PC is initialized there, and it should remain at the same address for many cycles before any instructions start to execute. This is because of the memory system feature that simulates a typical latency for the DRAM access. This latency is required for loading the instructions into the cache before executing.

Instruction *JAL* with specifically calculated offset as its operand is used here to initialize PC to 0x2000. However, we also need to add a few *NOP* instructions before executing the *JAL* instruction because we need to flush the pipeline and dump all possible trash data inside the instruction cache.

In this way, we are able to design an initial configuration sequence to bypass the initialization and start the SFV checks for the Rocket core from this pre-configured “waypoint”.

With the initial configuration sequence, we are able to run the SFV checks for all instructions in RV64I. However, even with the ‘big’ server with 32 CPUs and over 300 GB memory, the tool kept running for 72 hours and gave non-deterministic results for some properties in the instruction checks. The situation repeats in other instruction checks but the properties that give non-deterministic results vary. In formal verification, non-deterministic results mean that there are neither proofs nor counterexamples found for the property. A commonly used standard in industrial verification projects is to constrain the check to be within a certain bound (80 in our experiment). It is be good enough to conclude that the design is bug-free within this bound.

4.4.3 Automatic Generated Waypoints

As mentioned above, the Rocket core is specifically suitable for automatically generated waypoints because of its large size (nearly 300 thousand lines) and non-human-readable feature (due to the code being generated by Berkely’s Chisel compiler). Applying the method presented in Chapter 3, we have scored all internal flip-flops in the Rocket RTL design. The scoring result is presented in Table B.4 attached in Appendix B.2.

To delimit critical flip-flops for the Rocket core, the threshold is set to 0.8. We then run the automatic waypoint guide generation tool. One thing needs to be mentioned here is that the nickname (defined in Algorithm 1) searching for Rocket core is slightly modified. To illustrate, the direct connect combinational logic will be extended to also include the RV with only one logic operation between CFF and another signal. In addition, a constant 'Level' is introduced to inform the tool how many hierarchies it should search. This hierarchical level can be modified by user to restrain how many nicknames are found.

The suggested waypoints along with their line information are shown in Appendix B.2. It can be seen from the waypoint report that the machine-generated ones are large in number, which requires a lot more effort to select and order.

4.4.4 Notes for Temporary Results

The experiment results for Rocket core show that some of the properties checked for functional correctness of each instruction are still non-deterministic with all optimizations involved. One obstacle is the unresolved conflicts between the initial sequence and the design. This issue comes from the different macro definitions in the initial configuration stage and the regular work stage. We are not able to dynamically change the macros during the verification process, so the RISC-V verification framework should take the responsibility of altering the behavior pattern. This strategy works when we only apply the initial configuration sequence in the formal checks, but problems occur again when we try to combine the configuration with the traces found to hit the waypoint. This should be the most important issue to resolve in order to make progress in this study case.

In any case, the waypoint-based SFV method, either manual or automatically selected, has been proven to be effective in the previous study cases. The scope of this target might be beyond the reach of this thesis's timeline, but it is still worthy of studying for any related future work. It is also reasonable to make the hypothesis that the efficiency of the RISC-V verification framework for end-to-end verification needs to be improved for implementations of large scale processors. We would also recommend that future research can explore the automation regarding waypoints, not only based on the RTL code, but also based on the status of the formal engine in real time.

Chapter 5

Conclusions and Future Work

In general, this thesis has proposed a waypoint-based SFV approach to extend the capacity of formal tools, which proves to be very effective using performance metrics of both runtime and memory consumption. The detailed summary of this thesis is as follows.

1. A waypoint-based SFV method is proposed in this thesis. The waypoint definition and propagation policy are discussed in detail. The implementation algorithm flow is presented and experiments are conducted for a synchronized FIFO to compare the results from classic FV, single waypoint SFV and multiple waypoint SFV in terms of memory usage and runtime. The results show that the formal engine capacity can be greatly improved by introducing waypoints in the verification flow.

2. As a supplement for the cases where manually selected waypoints are impractical, an automatic waypoint guide generation tool is developed in this thesis. Nine important features from the synthesized netlists are collected to form the training data set, and then a full connection neural network is trained to find the critical flip-flops. These critical flip-flops are then used to generate waypoint guidance by analyzing the Verilog

source codes. The proposed SFV flow is run with the automatically generated waypoints, and the results show that this automation procedure is effective in helping the formal engine to extend its capacity but not as good as the user-defined waypoints in our experiments.

3. Two RISC-V cores, PicoRV32 and Rocket, are selected as study cases to test the effectiveness of our waypoint-based SFV. An expandable and reusable RISC-V verification framework has been applied to perform end-to-end ISA verification, circumventing the troubles to look into design details. Our experiments suggest that the proposed SFV methods is quite promising for extending the tool capacity for such large designs.

Although being powerful, the methods proposed in this work still have many aspects that could be improved. Here we list several suggestions on prospective future work.

1. The instructions verified in this thesis are still a subset of all ISAs supported. Instructions such as multiplication and floating-point operations can be verified using more suitable methodologies.

2. More comprehensive coverage metrics, such as state coverage and pairwise signal coverage, can be applied to evaluate the waypoint-based SFV method.

3. The training data set can be expanded further with more designs because the loss function, though tending to be flat, still shows a trend to

decrease. More advanced neural network structures and other feature vectors are also worth further study.

4. The automatic waypoint generation tool can be more intelligent than just providing guidance. This should involve a comprehensive Verilog parser and more design specific configurations based on different specifications. Realtime information from the formal engine can also be take into consideration.

Appendices

Appendix A

Important Scripts Used in This Thesis

A.1 TCL Command to Run JasperGold

Listing A.1: Jaspergold setup command (example)

```
clear -all;

analyze -sv -f v_file.f;
analyze -sv -f sv_file.f;

elaborate -top {rvfi_testbench} -enable_sva_isunknown
# black-boxing in the elaborate step

clock clock;
reset -expression reset;
# reset -sequence jg_init.seq;
# reset -sequence -vcd init_new.vcd
# the above used for waypoint based SFV, initial sequence can be seq or vcd
    files

# assumption or abstractions can be made here in addition

prove -all;
exit
```

A.2 Cadence Encounter Timing Analysis Script

Listing A.2: get.tcl

```
set_attribute hdl_search_path {./}
set_attribute lib_search_path {./}
set_attribute library [list gscl45nm.lib]

set current_design DUT
set myFiles [list DUT.v]

read_hdl ${myFiles}
elaborate ${current_design}

read_sdc ./constraints.sdc

check_design -unresolved
report timing -lint
```

```
# Synthesize the design to the target library
synthesize -to_mapped

write_hdl -mapped >  ${current_design}_netlist.v

puts "**********************"
puts "all_outputs"
puts "**********************"
puts [all_outputs]
set all_out [all_outputs]

puts "**********************"
puts "all_registers"
puts "**********************"
puts [all_registers]
set all_reg [all_registers]

puts "**********************"
puts "all_inputs"
puts "**********************"
puts [all_inputs]
set all_in [all_inputs]


set fp [ open "test2.rpt" a+ ]


#$all_in

#$all_out

for {set a 0} {$a < [llength $all_reg]} {incr a} {
    set tmp_reg [lindex $all_reg $a]
    regsub -all {\/designs\/DUT\/instances\_seq\/} $tmp_reg "" mytmp
    puts $fp $mytmp

    regsub -all {\[[0-9]*\]} $mytmp "" init_key
    set search_key "*"
    append search_key $init_key
    append search_key "\*Q*"
    append mytmp "\/D"

    set fanin [all_fanin -to $mytmp -startpoints_only]
    set fanin_verbose [all_fanin -to $mytmp]

    regsub -all {\/D} $mytmp {\/CLK} mytmp2
    set fanout [all_fanout -from $mytmp2 -endpoints_only]
    set fanout_verbose [all_fanout -from $mytmp2]

    set in_key "*ports\_in*"
    puts $fp [llength [lsearch -all -inline $fanin $in_key]]

    set out_key "*ports\_out*"
    puts $fp [llength [lsearch -all -inline $fanout $out_key]]
    puts $fp [llength $fanin]
    puts $fp [llength $fanout]
    puts $fp [llength $fanin_verbose]
    puts $fp [llength $fanout_verbose]

    set fanin_loop [all_fanin -to $mytmp]
    if { [llength [lsearch -all -inline  $fanin_loop  $search_key] ]} then {
        puts $fp "1"} else {puts $fp "0"}

    puts $fp [llength $all_in]
    puts $fp [llength $all_out]
    puts $fp [llength $all_reg]
}
puts "registers D_done"

close $fp
```

A.3 Other Codes

Other codes that are too large to fit in the appendix can be found on Github repository: `https://github.com/bearichan/thesis_SFV.git`

The repository includes:

- Neural Network training model and prediction code
- Automatic Waypoint Guide Generation Tool
- Formal Verification Framework for RISC-V (modified from work [54])
- RTL source codes of the design used to gather the training set data
- The script to source information from Candece RC as well as the raw data gathered for scoring prediction

A README file can be found via link above illustrating how the files are origianized and how to use the tools.

Appendix B

Original Experiment Data

B.1 Original Data of PicoRV32 FV and SFV Experiments

Table B.1: Original data collected for PICORV32 core FV and SFV tests

ISA name	FV runtime	FV mem usage	SFV runtime	SFV mem usage	line coverage	signal coverage	runtime improved	mem reduced
add	1.93	12.453	0.362	0.743	86.22%	70.77%	81%	94%
addi	2.173	13.159	0.314	0.749	85.75%	68.37%	86%	94%
and	2.046	14.75	0.355	0.732	87.41%	72.82%	83%	95%
andi	1.937	12.432	0.38	0.783	85.75%	68.37%	80%	94%
auipc	1.072	6.723	0.382	0.743	84.89%	65.54%	64%	89%
beq	1.387	8.464	0.412	0.704	87.51%	72.91%	70%	92%
bge	1.184	7.629	0.412	0.754	87.51%	72.91%	65%	90%
bgeu	1.011	7.536	0.434	0.784	87.51%	72.91%	57%	90%
blt	1.538	11.789	0.449	0.733	87.51%	72.91%	71%	94%
bltu	1.411	8.209	0.407	0.688	87.51%	72.91%	71%	92%
bne	1.41	7.985	0.425	0.607	87.51%	72.91%	70%	92%
jal	1.438	11.716	0.425	0.668	85.26%	64.11%	70%	94%
jalr	1.617	11.011	0.388	0.785	86.19%	67.21%	76%	93%
lb	2.613	14.549	0.445	0.792	86.29%	67.68%	83%	95%
lbu	1.792	13.664	0.476	0.75	86.29%	67.68%	73%	95%
lh	1.808	11.51	0.411	0.761	86.29%	67.71%	77%	93%
lhu	1.785	8.268	0.368	0.722	86.29%	67.71%	79%	91%
lui	1.169	11.37	0.379	0.84	85.23%	63.95%	68%	93%
lw	1.968	12.635	0.344	0.764	86.29%	67.78%	83%	94%
or	1.772	13.571	0.39	0.687	86.83%	70.18%	78%	95%
ori	2.08	13.959	0.35	0.716	86.18%	67.20%	83%	95%
sb	4.029	27.51	0.417	0.902	86.82%	70.26%	90%	97%
sh	3.88	25.507	0.407	0.741	86.82%	70.26%	90%	97%
sll	2.328	13.587	0.398	0.636	86.84%	67.62%	83%	95%
slli	1.606	13.747	0.362	0.618	86.19%	67.12%	77%	96%
slt	2.116	14.682	0.392	0.729	86.83%	70.18%	81%	95%
slti	1.748	11.458	0.357	0.731	86.18%	67.20%	80%	94%
sltiu	1.747	8.148	0.393	0.697	86.18%	67.20%	78%	91%
sltu	2.414	14.162	0.407	0.71	86.83%	70.18%	83%	95%
sra	2.256	14.375	0.373	0.729	86.84%	67.62%	83%	95%
srai	1.949	9.285	0.367	0.575	86.19%	67.12%	81%	94%
srl	1.979	13.513	0.371	0.627	86.84%	67.62%	81%	95%

srli	1.814	11.475	0.374	0.607	86.19%	67.12%	79%	95%
sub	1.832	14.861	0.338	0.788	86.83%	70.18%	82%	95%
sw	1.389	12.878	0.401	0.705	86.82%	70.26%	71%	95%
xor	1.87	13.491	0.424	0.738	86.83%	70.18%	77%	95%
xori	1.341	12.473	0.416	0.748	86.18%	67.20%	69%	94%
c_add	1.722	14.004	0.321	0.668	86.81%	70.12%	81%	95%
c_addi	2.04	14.497	0.303	0.718	86.17%	67.16%	85%	95%
c_addi4spn	1.508	12.423	0.336	0.66	86.17%	67.16%	78%	95%
c_addi16sp	2.061	15.081	0.32	0.69	86.17%	67.16%	84%	95%
c_and	1.942	13.8	0.327	0.708	86.82%	70.13%	83%	95%
c_andi	2.142	15.808	0.344	0.79	86.10%	67.17%	84%	95%
c_beqz	1.612	13.707	0.348	0.684	86.18%	67.16%	78%	95%
c_bnez	1.533	13.5	0.345	0.762	86.18%	67.16%	77%	94%
c_j	1.709	13.498	0.336	0.773	85.26%	64.07%	80%	94%
c_jal	1.439	11.753	0.308	0.735	85.26%	64.07%	79%	94%
c_jalr	1.804	14.566	0.345	0.952	86.17%	67.04%	81%	93%
c_jr	2.387	14.518	0.367	0.864	85.90%	64.85%	85%	94%
c_li	1.517	13.785	0.335	0.771	85.26%	64.10%	78%	94%
c_lui	1.566	10.287	0.333	0.777	85.26%	64.10%	79%	92%
c_lw	2.666	14.79	0.346	0.728	86.29%	67.76%	87%	95%
c_lwsp	2.213	13.609	0.379	0.735	86.27%	67.74%	83%	95%
c_mv	1.389	13.71	0.349	0.7	85.90%	67.04%	75%	95%
c_or	2.294	15.929	0.387	0.683	86.82%	70.13%	83%	96%
c_slli	1.229	7.625	0.305	0.648	86.17%	67.04%	75%	92%
c_srai	1.542	8.365	0.335	0.598	86.18%	67.05%	78%	93%
c_srli	1.237	4.797	0.338	0.637	86.18%	67.05%	73%	87%
c_sub	1.447	10.18	0.293	0.731	86.82%	70.13%	80%	93%
c_sw	0.799	7.413	0.313	0.694	86.82%	70.24%	61%	91%
c_swsp	1.182	10.51	0.331	0.697	86.82%	70.22%	72%	93%
c_xor	2.013	14.642	0.328	0.718	86.82%	70.13%	84%	95%
mul	2.334	14.655	0.35	0.839	85.44%	62.47%	85%	94%
mulh	2.61	14.688	0.357	0.884	85.44%	62.47%	86%	94%
mulhsu	2.365	14.646	0.357	0.883	85.44%	62.47%	85%	94%
mulhu	2.087	13.779	0.351	0.781	85.44%	62.47%	83%	94%
rem	2.657	16.664	0.391	1.614	85.44%	62.47%	85%	90%
remu	2.305	14.505	0.372	1.47	85.44%	62.47%	84%	90%
div	2.614	14.574	0.393	1.686	85.44%	62.47%	85%	88%
divu	2.354	14.596	0.4	1.354	85.44%	62.47%	83%	91%

B.2 Neural Network Training Results

Table B.2: Neural network training result for synchronized FIFO

signal _name	Hier	Rin	Rout	>> R	R <<	>> Rv	Rv <<	Loop	Len	score	ref
full_reg	0	0	0.100	0.058	0.791	0.419	6.465	1	1	0.999	1
dffw1	0	0.08	0.000	0.006	0.814	0.012	6.756	0	1	0.999	0.8
dffw2	0	0	0.000	0.006	0.808	0.017	6.762	0	1	0.999	0.8
wr_reg	0	0	0.000	0.011	0.194	0.088	1.644	1	4	0.956	0.8
dffr1	0	0.08	0.000	0.006	0.116	0.012	4.390	0	1	0.658	0.8
dffr2	0	0	0.000	0.006	0.110	0.017	4.395	0	1	0.585	0.8
rd_reg	0	0	0.000	0.011	0.018	0.083	0.987	1	4	0.411	0.8

empty_reg	0	0	0.100	0.081	0.047	1.267	0.262	1	1	0.328	1
regarray[1]	0	0.01	0.000	0.007	0.002	0.044	0.015	1	8	0.046	0
regarray[9]	0	0.01	0.000	0.007	0.002	0.046	0.015	1	8	0.046	0
regarray[3]	0	0.01	0.000	0.007	0.002	0.046	0.015	1	8	0.046	0
regarray[5]	0	0.01	0.000	0.007	0.002	0.046	0.015	1	8	0.046	0
regarray[0]	0	0.01	0.000	0.007	0.002	0.043	0.019	1	8	0.046	0
regarray[7]	0	0.01	0.000	0.007	0.002	0.047	0.015	1	8	0.046	0
regarray[13]	0	0.01	0.000	0.007	0.002	0.047	0.015	1	8	0.046	0
regarray[11]	0	0.01	0.000	0.007	0.002	0.047	0.015	1	8	0.046	0
regarray[4]	0	0.01	0.000	0.007	0.002	0.044	0.019	1	8	0.046	0
regarray[8]	0	0.01	0.000	0.007	0.002	0.044	0.019	1	8	0.046	0
regarray[2]	0	0.01	0.000	0.007	0.002	0.044	0.019	1	8	0.046	0
regarray[15]	0	0.01	0.000	0.007	0.002	0.049	0.015	1	8	0.046	0
regarray[10]	0	0.01	0.000	0.007	0.002	0.046	0.019	1	8	0.046	0
regarray[12]	0	0.01	0.000	0.007	0.002	0.046	0.019	1	8	0.046	0
regarray[6]	0	0.01	0.000	0.007	0.002	0.046	0.019	1	8	0.046	0
regarray[14]	0	0.01	0.000	0.007	0.002	0.047	0.019	1	8	0.046	0
out	0	0	0.013	0.017	0.002	0.180	0.010	1	8	0.034	0

Table B.3: Neural network training result for PicoRV32

signal_name	Hier	Rin	Rout	$>> R$	$R <<$	$>> Rv$	$Rv <<$	Loop	Len	score
latched_branch	0	0.01	0.098	0.054	0.568	0.86	5.428	1	1	0.995
latched_store	0	0.01	0.098	0.056	0.568	0.867	5.399	1	1	0.995
latched_stalu	0	0.01	0	0.003	0.537	0.019	2.271	1	1	0.985
latched _rd	0	0.002	0	0.001	0.101	0.005	0.79	1	5	0.928
cpu_state	0	0.003	0	0.004	0.079	0.031	0.952	1	7	0.909
mem_state	0	0.01	0.003	0.003	0.048	0.04	0.784	1	2	0.731
mem_do _wdata	0	0.02	0.003	0.008	0.096	0.069	1.558	1	1	0.719
mem_do _rdata	0	0.02	0.003	0.008	0.079	0.063	1.471	1	1	0.682
mem_valid	0	0.02	0.003	0.005	0.078	0.065	1.514	1	1	0.677
mem_do _prefetch	0	0.02	0.101	0.007	0.077	0.051	1.494	1	1	0.666
mem_do_rinst	0	0.02	0.101	0.083	0.095	1.043	1.787	1	1	0.62
decoder_ trigger	0	0.02	0	0.059	0.092	0.94	1.124	0	1	0.619
is_beq_bne _blt_bge_bltu _bgeu	0	0.088	0	0.012	0.029	0.096	0.238	1	1	0.555
decoder_ pseudo_ trigger	0	0.02	0	0.008	0.038	0.069	0.436	0	1	0.552
trap	0	0.01	0.003	0.004	0.037	0.026	0.28	0	1	0.537
is_alu_ reg_imm	0	0.088	0	0.012	0.023	0.096	0.157	1	1	0.486
is_sb_ sh_sw	0	0.088	0	0.012	0.02	0.098	0.138	1	1	0.454
mem_ word-size	0	0.01	0.046	0.004	0.025	0.037	0.304	1	2	0.445
is_lui_auipc _jal_jalr_addi _add_sub	0	0	0	0.005	0.017	0.024	0.168	0	1	0.371

instr_jal	0	0.088	0	0.012	0.074	0.096	4.843	1	1	0.357
latched_is_lb	0	0.02	0	0.007	0.013	0.06	0.097	1	1	0.353
latched_is_lh	0	0.02	0	0.007	0.013	0.06	0.119	1	1	0.35
latched_is_lu	0	0.02	0	0.007	0.013	0.06	0.144	1	1	0.346
is_alu_reg_reg	0	0.088	0	0.012	0.007	0.097	0.055	1	1	0.283
instr_jalr	0	0.118	0	0.015	0.058	0.115	4.664	1	1	0.243
is_lb_lh_ lw_lbu_lhu	0	0.088	0	0.012	0.058	0.097	4.65	1	1	0.235
is_sll_ srl_sra	0	0	0	0.007	0.003	0.048	0.017	1	1	0.232
is_sltiu_ bltu_sltu	0	0	0	0.002	0.003	0.007	0.021	0	1	0.232
is_slti_ blt_slt	0	0	0	0.002	0.003	0.007	0.022	0	1	0.231
mem_instr	0	0.01	0.003	0.005	0.002	0.046	0.004	1	1	0.221
is_lbu_lhu_lw	0	0	0	0.002	0.001	0.007	0.005	0	1	0.212
is_compare	0	0.01	0	0.004	0.001	0.021	0.004	0	1	0.205
reg_sh	0	0.002	0	0.01	0.008	0.079	0.192	1	5	0.201
mem_wstrb	0	0.002	0.001	0.002	0	0.018	0.001	1	4	0.196
instr_sub	0	0.01	0	0.008	0.056	0.041	4.703	1	1	0.193
instr_andi	0	0.01	0	0.004	0.055	0.024	4.728	1	1	0.19
decoded_rs2	0	0.006	0	0.001	0.006	0.011	0.475	1	5	0.188
instr_and	0	0.01	0	0.008	0.055	0.041	4.727	1	1	0.187
instr_ori	0	0.01	0	0.004	0.055	0.025	4.774	1	1	0.185
instr_or	0	0.01	0	0.008	0.055	0.042	4.773	1	1	0.183
instr_xori	0	0.01	0	0.004	0.055	0.026	4.798	1	1	0.182
instr_xor	0	0.01	0	0.008	0.055	0.043	4.798	1	1	0.18
instr_auipc	0	0.088	0	0.012	0.05	0.097	4.652	1	1	0.179
instr_lui	0	0.088	0	0.012	0.05	0.096	4.672	1	1	0.177
instr_rdcycle	0	0	0	0.015	0.056	0.079	4.816	1	1	0.177
instr_rdcycleh	0	0	0	0.015	0.056	0.08	4.838	1	1	0.175
instr_rdinstr	0	0	0	0.015	0.056	0.08	4.844	1	1	0.175
instr_rdinstrh	0	0	0	0.015	0.056	0.081	4.864	1	1	0.173
decoded_rs1	0	0.006	0	0.001	0.004	0.011	0.518	1	5	0.171
decoded_rd	0	0.006	0	0.001	0	0.011	0.002	1	5	0.127
instr_beq	0	0.01	0	0.004	0.041	0.027	4.563	1	1	0.114
instr_bge	0	0.01	0	0.004	0.041	0.025	4.564	1	1	0.114
instr_bgeu	0	0.01	0	0.004	0.041	0.024	4.563	1	1	0.114
instr_bne	0	0.01	0	0.004	0.041	0.026	4.568	1	1	0.114
instr_slti	0	0.01	0	0.004	0.04	0.026	4.544	1	1	0.11
instr_sltiu	0	0.01	0	0.004	0.04	0.025	4.544	1	1	0.11
instr_slt	0	0.01	0	0.008	0.04	0.043	4.544	1	1	0.109
instr_sltu	0	0.01	0	0.008	0.04	0.042	4.544	1	1	0.109
instr_blt	0	0.01	0	0.004	0.04	0.026	4.547	1	1	0.108
instr_bltu	0	0.01	0	0.004	0.04	0.025	4.547	1	1	0.108
instr_lb	0	0	0	0.004	0.04	0.024	4.55	1	1	0.108
instr_lbu	0	0	0	0.004	0.04	0.023	4.553	1	1	0.108
instr_lh	0	0	0	0.004	0.04	0.02	4.55	1	1	0.108
instr_lhu	0	0	0	0.004	0.04	0.022	4.553	1	1	0.108
instr_addi	0	0.01	0	0.004	0.04	0.027	4.559	1	1	0.107
instr_lw	0	0	0	0.004	0.04	0.023	4.545	1	1	0.106
instr_add	0	0.01	0	0.008	0.04	0.044	4.559	1	1	0.105
instr_sb	0	0	0	0.004	0.04	0.024	4.551	1	1	0.105
instr_sh	0	0	0	0.004	0.04	0.02	4.555	1	1	0.105

instr_sw	0	0	0	0.004	0.039	0.023	4.553	1	1	0.103
instr_sll	0	0.01	0	0.008	0.039	0.043	4.591	1	1	0.101
is_lui_auipc_jal	0	0	0	0.002	0.038	0.007	4.593	0	1	0.101
instr_slli	0	0	0	0.007	0.039	0.037	4.589	1	1	0.1
instr_srl	0	0.01	0	0.008	0.039	0.042	4.624	1	1	0.099
instr_sra	0	0.01	0	0.008	0.039	0.043	4.632	1	1	0.098
instr_srai	0	0	0	0.007	0.039	0.037	4.63	1	1	0.097
instr_srli	0	0	0	0.007	0.039	0.037	4.623	1	1	0.097
is_slli_srli_srai	0	0	0	0.007	0.038	0.048	4.536	1	1	0.097
is_jalr_addi_slti_sltiu_xori_ori_andi	0	0	0	0.004	0.038	0.042	4.577	1	1	0.096
decoded_imm_uj	0	0.003	0	0.001	0.001	0.005	0.006	1	10	0.033
alu_out_q	0	0	0	0.001	0.001	0.012	0.004	0	32	0.001
reg_out	0	0.002	0	0.001	0.001	0.012	0.004	0	32	0.001
cpuregs[10]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[11]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[12]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[13]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[14]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[15]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[16]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[17]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[18]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[19]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[1]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[20]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[21]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[22]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[23]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[24]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[25]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[26]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[27]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[28]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[29]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[2]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[30]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[31]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[3]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[4]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[5]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[6]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[7]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[8]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
cpuregs[9]	0	0	0	0.001	0	0.003	0.001	1	32	0.001
decoded_imm	0	0	0	0	0	0.001	0.002	1	32	0.001
mem_addr	0	0	0	0	0	0.002	0	1	30	0.001
mem_rdata_q	0	0.001	0	0	0	0	0.002	1	32	0.001
mem_wdata	0	0	0	0	0	0.001	0	1	32	0.001

reg_next_pc	0	0	0	0.001	0	0.013	0.003	1	31	0.001
reg_op1	0	0.001	0	0.002	0	0.02	0.004	1	32	0.001
reg_op2	0	0	0	0.001	0	0.012	0.004	1	32	0.001
reg_pc	0	0	0	0	0.008	0.001	0.028	1	31	0.001
count_cycle	0	0	0	0	0	0.001	0.002	1	64	0
count_instr	0	0	0	0	0	0.002	0.001	1	64	0

Table B.4: Neural network training result for Rocket Core

signal _name	Hier	Rin	Rout	>> *R*	*R* <<	>> *Rv*	*Rv* <<	Loop	Len	score
core/wb_reg_inst	1	0	0	0.001	0.295	0.004	2.891	1	17	0.997
core/wb_ ctrl__mem	1	0	0	0.001	0.408	0.004	6.396	1	1	0.977
core/wb_ reg_replay	1	0	0	0.003	0.408	0.023	6.357	1	1	0.977
core/wb_ reg_xcpt	1	0	0	0.005	0.408	0.034	6.394	1	1	0.977
core/wb_ reg_valid	1	0	0	0.005	0.408	0.037	6.991	1	1	0.977
core/wb_ ctrl__wxd	1	0	0	0.001	0.406	0.004	6.860	1	1	0.976
core/wb_ ctrl__csr	1	0	0	0.001	0.382	0.004	1.450	1	3	0.968
dtim_adapter /state	1	0	0	0.004	0.034	0.041	0.226	1	3	0.700
core/div /state	2	0	0	0.001	0.039	0.017	0.658	1	3	0.628
dcache/s2_ req_cmd	1	0	0	0.014	0.043	0.146	0.302	1	5	0.626
dcache/s2_ valid_pre_ xcpt	1	0	0	0.004	0.043	0.029	0.243	1	1	0.598
core/mem_ reg_valid	1	0	0	0.005	0.103	0.036	3.935	1	1	0.596
dcache/pstore1 _held	1	0	0	0.003	0.041	0.032	0.223	1	1	0.592
dcache/_T _602	1	0	0	0.014	0.043	0.140	0.238	1	1	0.585
frontend/fq/ _T_60_0	2	0	0	0.007	0.062	0.056	0.562	1	1	0.585
dcache/pstore2 _valid	1	0	0	0.003	0.041	0.033	0.326	1	1	0.585
dcache/s2 _req_phys	1	0	0	0.014	0.043	0.143	0.247	1	1	0.584
frontend/fq/ _T_60_4	2	0	0	0.007	0.062	0.054	0.513	1	1	0.584
dcache/_T_2984	1	0	0	0.014	0.043	0.147	0.247	1	1	0.584
dcache/s2_hit _state_state	1	0	0	0.017	0.043	0.160	0.238	1	1	0.582
dcache/_T _2986_ma_st	1	0	0	0.018	0.043	0.186	0.245	1	1	0.579
dcache/_T _2986_ma_ld	1	0	0	0.019	0.043	0.187	0.245	1	1	0.579
dcache/_T _2986_ae_st	1	0	0	0.018	0.043	0.187	0.244	1	1	0.579

dcache/_T _2986_ae_ld	1	0	0	0.019	0.043	0.187	0.244	1	1	0.579
core/mem_reg _flush_pipe	1	0	0	0.001	0.037	0.006	0.355	1	1	0.576
dcache/s1_valid	1	0	0	0.014	0.039	0.154	0.308	1	1	0.566
dcache/ un-cachedInFlight _0	1	0	0	0.005	0.028	0.034	0.152	1	1	0.549
buffer_1/ Queue/value	2	0	0	0.001	0.043	0.005	0.223	1	1	0.540
buffer_1/ Queue/_T_39	2	0	0	0.003	0.043	0.019	0.222	1	1	0.539
buffer_1/ Queue/value_1	2	0	0	0.003	0.043	0.017	0.309	1	1	0.535
buffer/Queue/ value	2	0	0	0.001	0.042	0.005	0.215	1	1	0.535
buffer/Queue/ _T_39	2	0	0	0.001	0.042	0.008	0.215	1	1	0.535
dtim_adapter/ acq_param	1	0	0	0.004	0.023	0.039	0.128	1	3	0.535
dtim_adapter/ acq_opcode	1	0	0	0.004	0.023	0.039	0.129	1	3	0.535
buffer/Queue/ value_1	2	0	0	0.001	0.042	0.006	0.298	1	1	0.532
frontend/fq/ _T_60_3	2	0	0	0.007	0.037	0.058	0.281	1	1	0.506
frontend/fq/ _T_60_1	2	0	0	0.007	0.037	0.057	0.342	1	1	0.502
frontend/fq/ _T_60_2	2	0	0	0.007	0.037	0.058	0.341	1	1	0.501
dcache/ block-Uncached-Grant	1	0	0	0.003	0.023	0.035	0.219	1	1	0.484
buffer_1/Queue _1/value	2	0	0	0.001	0.027	0.006	0.141	1	1	0.471
buffer_1/Queue _1/_T_39	2	0	0	0.001	0.027	0.008	0.140	1	1	0.471
buffer/Queue _1/ value	2	0	0	0.001	0.026	0.006	0.138	1	1	0.467
buffer_1/Queue_1/ value_1	2	0	0	0.001	0.027	0.006	0.194	1	1	0.467
buffer/Queue _1/ _T_39	2	0	0	0.001	0.026	0.012	0.137	1	1	0.466
core/ex_reg _replay	1	0	0	0.004	0.038	0.043	0.358	0	1	0.463
buffer/Queue _1/value_1	2	0	0	0.001	0.026	0.010	0.191	1	1	0.462
core/ex_reg _xcpt_interrupt	1	0	0	0.005	0.038	0.046	0.356	0	1	0.461
dcache/s1_ req_typ	1	0	0	0.000	0.017	0.003	0.112	1	3	0.443
core/ex_reg _valid	1	0	0	0.035	0.079	0.563	3.755	1	1	0.438

frontend/icache/ s2_valid	2	0	0	0.001	0.017	0.005	0.207	1	1	0.401
frontend/_T_222	1	0	0	0.002	0.013	0.012	0.106	1	1	0.367
frontend/s2_valid	1	0	0	0.002	0.013	0.012	0.108	1	1	0.367
frontend/s2 _speculative	1	0	0	0.002	0.013	0.009	0.112	1	1	0.367
core/ex_reg _rs_lsb_1	1	0	0	0.045	0.024	0.706	0.419	1	2	0.366
frontend/s2_tlb _resp_ae_inst	1	0	0	0.006	0.013	0.032	0.112	1	1	0.359
core/mem_reg _xcpt_interrupt	1	0	0	0.003	0.024	0.025	0.183	0	1	0.358
core/csr/ _T_1242	2	0	0	0.000	0.012	0.006	0.100	1	1	0.356
core/mem_ reg_rvc	1	0	0	0.001	0.012	0.006	0.079	1	1	0.356
frontend/ icache/ invalidated	2	0	0	0.001	0.012	0.006	0.107	1	1	0.355
core/csr/ reg_wfi	2	0	0	0.001	0.012	0.013	0.100	1	1	0.354
core/mem_ reg_replay	1	0	0	0.005	0.023	0.035	0.180	0	1	0.354
core/mem_ ctrl_csr	1	0	0	0.001	0.042	0.006	3.382	1	3	0.348
core/ex_ reg_rs_ bypass_1	1	0	0	0.033	0.024	0.571	0.375	1	1	0.343
core/mem_ ctrl_jalr	1	0	0	0.001	0.055	0.006	3.550	1	1	0.341
core/mem_ ctrl_jal	1	0	0	0.001	0.054	0.006	3.476	1	1	0.341
core/mem_ ctrl_branch	1	0	0	0.001	0.054	0.006	3.480	1	1	0.340
core/div/ resHi	2	0	0	0.002	0.012	0.028	0.226	1	1	0.328
dcache/s1_ req_cmd	1	0	0	0.000	0.026	0.002	0.182	1	5	0.328
fragmenter_1/ Repeater/full	2	0	0	0.002	0.009	0.017	0.080	1	1	0.313
core/ex_ctrl __sel_imm	1	0	0	0.033	0.012	0.567	0.165	1	3	0.302
core/wb_ctrl __div	1	0	0	0.001	0.047	0.004	3.400	1	1	0.286
dcache/_T_1213	1	0	0	0.003	0.016	0.021	0.082	0	1	0.283
core/ibuf/buf __replay	2	0	0	0.010	0.009	0.230	0.058	1	1	0.279
frontend/icache /refill_valid	2	0	0	0.002	0.005	0.016	0.029	1	1	0.277
core/ibuf/ nBufValid	2	0	0	0.010	0.009	0.230	0.145	1	1	0.264
core/ex_ctrl__csr	1	0	0	0.032	0.041	0.567	3.381	1	3	0.255
dtim_adapter /acq_size	1	0	0	0.004	0.000	0.039	0.002	1	2	0.254
core/ex_ctrl __mem	1	0	0	0.032	0.054	0.564	3.597	1	1	0.252

frontend/_T_241	1	0	0	0.000	0.013	0.001	0.108	0	1	0.249
core/wb_reg_flush_pipe	1	0	0	0.005	0.043	0.038	3.389	1	1	0.248
core/mem_reg_xcpt	1	0	0	0.005	0.013	0.038	0.121	0	1	0.244
core/ex_ctrl__mem_type	1	0	0	0.033	0.008	0.565	0.131	1	3	0.244
core/mem_ctrl_wxd	1	0	0	0.001	0.042	0.007	3.395	1	1	0.244
core/mem_ctrl_div	1	0	0	0.001	0.042	0.007	3.382	1	1	0.241
core/mem_ctrl_mem	1	0	0	0.001	0.042	0.006	3.385	1	1	0.240
core/blocked	1	0	0	0.001	0.041	0.004	3.380	1	1	0.240
core/mem_reg_slow_bypass	1	0	0	0.001	0.041	0.007	3.382	1	1	0.239
core/ex_ctrl__sel_alu1	1	0	0	0.038	0.012	0.794	0.155	1	2	0.238
core/ex_ctrl__sel_alu2	1	0	0	0.037	0.012	0.794	0.179	1	2	0.236
tlMasterXbar/_T_1111_1	1	0	0	0.002	0.002	0.011	0.054	1	1	0.233
core/csr/reg_mstatus_mie	2	0	0	0.003	0.002	0.054	0.017	1	1	0.230
core/csr/reg_mstatus_mpie	2	0	0	0.002	0.002	0.054	0.014	1	1	0.229
tlMasterXbar/_T_1111_0	1	0	0	0.002	0.002	0.011	0.082	1	1	0.229
core/mem_ctrl_fence_i	1	0	0	0.001	0.001	0.006	0.002	1	1	0.228
buffer/Queue_1/_T_35_opcode[1]	2	0	0	0.001	0.001	0.006	0.006	1	1	0.226
core/div/ isHi	2	0	0	0.001	0.001	0.006	0.005	1	1	0.226
buffer/Queue_1/_T_35_opcode[0]	2	0	0	0.001	0.001	0.006	0.006	1	1	0.226
core/wb_ctrl__fence_i	1	0	0	0.001	0.000	0.004	0.002	1	1	0.226
frontend/s1_speculative	1	0	0	0.002	0.001	0.016	0.003	1	1	0.226
buffer_1/Queue_1/_T_35_opcode[1]	2	0	0	0.001	0.000	0.006	0.002	1	1	0.223
buffer_1/Queue_1/_T_35_opcode[0]	2	0	0	0.001	0.000	0.006	0.002	1	1	0.223
core/ex_ctrl__rxs2	1	0	0	0.032	0.012	0.564	0.208	1	1	0.221
core/div/req_dw	2	0	0	0.001	0.000	0.004	0.033	1	1	0.219
tlMasterXbar/_T_1026	1	0	0	0.002	0.001	0.012	0.076	1	1	0.218
core/ex_ctrl__alu_fn	1	0	0	0.037	0.012	0.798	0.083	1	4	0.217
frontend/fq/_T_82 _4_replay	2	0	0	0.007	0.001	0.055	0.002	1	1	0.214
frontend/fq/_T_82_4_xcpt_ae_inst	2	0	0	0.007	0.001	0.055	0.002	1	1	0.214

frontend/fq/ _T_82_1 _xcpt_ae_inst	2	0	0	0.007	0.001	0.061	0.003	1	1	0.213
frontend/fq/ _T_82_3 _replay	2	0	0	0.007	0.001	0.061	0.003	1	1	0.213
core/mem_br _taken	1	0	0	0.077	0.054	0.941	3.480	1	1	0.213
frontend/fq/ _T_82 _1_replay	2	0	0	0.007	0.001	0.061	0.003	1	1	0.213
frontend/fq/ _T_82_3 _xcpt_ae_inst	2	0	0	0.007	0.001	0.061	0.003	1	1	0.213
frontend/fq/ _T_82_2_replay	2	0	0	0.007	0.001	0.061	0.003	1	1	0.213
frontend/fq/ _T_82 _2_xcpt _ae_inst	2	0	0	0.007	0.001	0.062	0.003	1	1	0.213
fragmenter_1/ _T_222	1	0	0	0.001	0.002	0.017	0.028	1	3	0.213
frontend/fq/ _T_82_0 _xcpt_ae _inst	2	0	0	0.007	0.000	0.060	0.002	1	1	0.212
frontend/fq/ _T_82_0 _replay	2	0	0	0.007	0.000	0.061	0.002	1	1	0.211
fragmenter_1/ _T_323	1	0	0	0.003	0.001	0.029	0.014	1	3	0.199
dcache/ uncachedReqs _0_addr	1	0	0	0.003	0.001	0.022	0.002	1	3	0.197
dcache/ uncachedReqs _0_typ	1	0	0	0.003	0.001	0.022	0.003	1	3	0.197
fragmenter_1 /_T_224	1	0	0	0.001	0.000	0.012	0.002	1	3	0.196
frontend/icache /s2_tl_error	2	0	0	0.017	0.000	0.140	0.001	1	1	0.195
core/div/neg_out	2	0	0	0.026	0.001	0.144	0.006	1	1	0.195
core/ex_ reg_rs_ lsb_0	1	0	0	0.049	0.012	0.929	0.381	1	2	0.192
dcache/s2 _req_typ	1	0	0	0.015	0.001	0.147	0.064	1	3	0.190
core/ibuf/buf__ xcpt_ae_inst	2	0	0	0.010	0.000	0.230	0.002	1	1	0.184
fragmenter_1/ Repeater/ saved_size	2	0	0	0.002	0.009	0.017	0.045	1	3	0.180
fragmenter_1/ Repeater/ saved_param	2	0	0	0.002	0.009	0.017	0.046	1	3	0.180
fragmenter_1/ Repeater/ saved_opcode	2	0	0	0.002	0.009	0.017	0.047	1	3	0.180
core/ex_ctrl __div	1	0	0	0.032	0.042	0.565	3.389	1	1	0.179

core/csr/ reg_misa	2	0	0	0.003	0.002	0.049	0.018	1	2	0.175
core/ex_ctrl__jalr	1	0	0	0.033	0.041	0.565	3.381	1	1	0.174
core/ex_ctrl __wxd	1	0	0	0.033	0.041	0.564	3.390	1	1	0.174
core/id_reg_fence	1	0	0	0.032	0.041	0.565	3.380	1	1	0.173
core/ex_ reg_rs_ bypass_0	1	0	0	0.039	0.012	0.792	0.389	1	1	0.173
core/id_reg_pause	1	0	0	0.033	0.041	0.598	3.379	1	1	0.170
buffer/Queue_1/ _T_35_source	2	0	0	0.001	0.001	0.006	0.006	1	2	0.163
buffer/Queue/ _T_35_source	2	0	0	0.001	0.000	0.006	0.002	1	2	0.160
buffer/Queue_1/ _T_35_corrupt	2	0	0	0.001	0.000	0.006	0.002	1	2	0.160
core/ex_ctrl __alu_dw	1	0	0	0.038	0.006	0.793	0.037	1	1	0.153
frontend/ icache/s2 _hit	2	0	0	0.017	0.017	0.138	0.206	0	1	0.142
core/ex_ctrl __mem_cmd	1	0	0	0.033	0.001	0.564	0.007	1	1	0.140
core/ex_reg _load_use	1	0	0	0.032	0.001	0.566	0.007	1	1	0.139
core/ex_ctrl __fence_i	1	0	0	0.032	0.001	0.564	0.002	1	1	0.137
core/ex_ctrl__jal	1	0	0	0.032	0.001	0.564	0.002	1	1	0.137
core/ex_ctrl __branch	1	0	0	0.033	0.001	0.566	0.003	1	1	0.137
core/ex_reg _flush_pipe	1	0	0	0.033	0.001	0.582	0.002	1	1	0.135
frontend/ icache/ _T_154	2	0	0	0.001	0.012	0.004	0.179	0	1	0.123
frontend/ s1_valid	1	0	0	0.001	0.000	0.005	0.001	0	1	0.120
dcacheArb/ _T_212	1	0	0	0.000	0.000	0.001	0.002	0	1	0.119
dcache/ doUn-cachedResp	1	0	0	0.000	0.000	0.003	0.001	0	1	0.119
core/_T _1189	1	0	0	0.001	0.000	0.004	0.001	0	1	0.119
dcacheArb /_T_210	1	0	0	0.000	0.000	0.001	0.051	0	1	0.119
core/csr/reg_mie	2	0	0	0.002	0.002	0.040	0.019	1	3	0.117
core/ex_reg_rvc	1	0	0	0.040	0.001	0.791	0.009	1	1	0.117
frontend/icache/ s1_valid	2	0	0	0.001	0.011	0.006	0.080	0	1	0.116
buffer_1/Queue/ _T_35_opcode[1]	2	0	0	0.001	0.001	0.006	0.006	1	3	0.111
buffer_1/Queue/ _T_35_opcode[0]	2	0	0	0.001	0.001	0.006	0.006	1	3	0.111
buffer_1/Queue/ _T_35_param[0]	2	0	0	0.001	0.001	0.006	0.006	1	3	0.111
buffer_1/Queue/ _T_35_param[1]	2	0	0	0.001	0.001	0.006	0.006	1	3	0.111

buffer/Queue/ _T_35_size[1]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer_1/Queue_1/ _T_35_size[1]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer/Queue/ _T_35_opcode[1]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer/Queue/ _T_35_param[1]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer_1/Queue/ _T_35_size[0]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer_1/Queue/ _T_35_size[1]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer_1/Queue _1/_T_35_size[0]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer/Queue/ _T_35_size[0]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer/Queue/ _T_35_opcode[0]	2	0	0	0.001	0.000	0.006	0.002	1	3	0.109
buffer/Queue/ _T_35_param[0]	2	0	0	0.001	0.000	0.007	0.002	1	3	0.109
dcache/mask	1	0	0	0.003	0.025	0.035	0.144	1	8	0.084
dcache/ pstore1_mask	1	0	0	0.014	0.025	0.144	0.146	1	8	0.084
core/mem_ reg_cause	1	0	0	0.001	0.001	0.006	0.003	1	5	0.077
core/wb_ reg_cause	1	0	0	0.002	0.000	0.013	0.026	1	5	0.076
core/ex_reg_xcpt	1	0	0	0.040	0.000	0.788	0.002	0	1	0.074
core/ex_ reg_cause	1	0	0	0.035	0.001	0.568	0.003	1	5	0.072
buffer/Queue_1/ _T_35_size[1]	2	0	0	0.001	0.000	0.006	0.002	1	4	0.072
buffer/Queue_1/ _T_35_size[0]	2	0	0	0.001	0.000	0.006	0.002	1	4	0.072
dtim_adapter /acq_mask	1	0	0	0.004	0.023	0.039	0.121	1	8	0.071
intsink/ Synchronizer-ShiftReg _w1_d3/ sync_2	2	0	0	0.000	0.000	0.000	0.001	0	1	0.068
intsink/ Synchronizer-ShiftReg _w1_d3/ sync_1	2	0	0	0.000	0.000	0.001	0.001	0	1	0.068
intsink/ Synchronizer-ShiftReg _w1_d3/ sync_0	2	0	0	0.000	0.000	0.001	0.000	0	1	0.067
core/csr/_T_241	2	0	0	0.002	0.013	0.041	0.103	1	6	0.061
core/csr/_T_251	2	0	0	0.003	0.013	0.042	0.103	1	6	0.061
core/csr/ reg_mcause	2	0	0	0.002	0.002	0.046	0.012	1	5	0.049
dcache/s1 _req_tag	1	0	0	0.000	0.001	0.003	0.002	1	6	0.046

dcache/ uncachedReqs _0_tag	1	0	0	0.003	0.001	0.022	0.003	1	6	0.046
dcache/s2 _req_tag	1	0	0	0.015	0.001	0.148	0.003	1	6	0.045
buffer_1/Queue/ _T_35_source[0]	2	0	0	0.001	0.000	0.006	0.002	1	5	0.045
buffer_1/ Queue_1/ _T_35_source[0]	2	0	0	0.001	0.000	0.006	0.002	1	5	0.045
buffer_1/Queue/ _T_35_source[1]	2	0	0	0.001	0.000	0.006	0.002	1	5	0.045
buffer_1/ Queue_1/ _T_35_source[1]	2	0	0	0.001	0.000	0.006	0.002	1	5	0.045
core/div/req_tag	2	0	0	0.001	0.000	0.004	0.001	1	5	0.045
fragmenter_1 /Repeater/ saved_source	2	0	0	0.002	0.000	0.017	0.002	1	5	0.044
dcache/ pstore2_addr	1	0	0	0.003	0.025	0.035	0.143	1	11	0.043
dcache/ pstore1_addr	1	0	0	0.014	0.025	0.144	0.143	1	11	0.037
dtim_ adapter/ acq_source	1	0	0	0.004	0.010	0.039	0.053	1	9	0.022
tlMasterXbar/ _T_1015	1	0	0	0.002	0.002	0.018	0.097	1	8	0.018
core/div/count	2	0	0	0.001	0.002	0.009	0.019	1	7	0.017
dcache/_T_1283	1	0	0	0.003	0.000	0.035	0.003	1	8	0.016
dcache/_T_1289	1	0	0	0.003	0.000	0.035	0.003	1	8	0.016
dcache/_T_1295	1	0	0	0.003	0.000	0.035	0.003	1	8	0.016
dcache/_T_1301	1	0	0	0.003	0.000	0.034	0.003	1	8	0.016
dcache/_T_1307	1	0	0	0.003	0.000	0.034	0.003	1	8	0.016
dcache/_T_1313	1	0	0	0.003	0.000	0.034	0.003	1	8	0.016
dcache/_T_1319	1	0	0	0.003	0.000	0.034	0.003	1	8	0.016
dcache/_T_1325	1	0	0	0.003	0.000	0.034	0.003	1	8	0.016
frontend/ icache/ _T_171	2	0	0	0.002	0.014	0.019	0.150	1	9	0.014
dcache/_T_2726	1	0	0	0.002	0.002	0.027	0.036	1	9	0.013
buffer/Queue/ _T_35_mask[0]	2	0	0	0.001	0.000	0.006	0.002	1	8	0.009
buffer_1/Queue/ _T_35_mask[0]	2	0	0	0.001	0.000	0.006	0.002	1	8	0.009
buffer_1/Queue/ _T_35_mask[1]	2	0	0	0.001	0.000	0.006	0.002	1	8	0.009
buffer/Queue/ _T_35_mask[1]	2	0	0	0.001	0.000	0.006	0.002	1	8	0.009
frontend/fq/ _T_82_0_data	2	0	0	0.007	0.062	0.060	0.512	1	32	0.003
core/ibuf/ buf__data	2	0	0	0.010	0.007	0.230	0.042	1	16	0.001
core/mem _reg_inst	1	0	0	0.001	0.019	0.006	0.746	1	25	0.001
core/ex_reg_inst	1	0	0	0.033	0.009	0.572	0.690	1	25	0.000

core/mem_ reg_pc	1	0	0	0.001	0.009	0.006	0.054	1	33	0.000
dcache/ s1_req_addr	1	0	0	0.007	0.011	0.035	0.068	1	34	0.000
frontend/icache/ refill_addr	2	0	0	0.001	0.003	0.007	0.026	1	26	0.000
frontend/s1_pc	1	0	0	0.005	0.003	0.028	0.020	1	33	0.000
dtim_adapter /acq_address	1	0	0	0.004	0.000	0.039	0.002	1	32	0.000
core/_T _1320	1	0	0	0.004	0.041	0.034	3.391	1	31	0.000
frontend/ s2_pc	1	0	0	0.002	0.001	0.012	0.003	1	33	0.000
core/wb_reg_pc	1	0	0	0.001	0.000	0.004	0.003	1	33	0.000
dcache/s2 _req_addr	1	0	0	0.014	0.001	0.145	0.010	1	32	0.000
core/ex_ reg_pc	1	0	0	0.035	0.001	0.567	0.008	1	33	0.000
core/csr/ reg_mtvec	2	0	0	0.002	0.001	0.040	0.012	1	31	0.000
buffer/Queue/ _T_35_address[1]	2	0	0	0.001	0.000	0.006	0.002	1	32	0.000
frontend/icache/ s2_dout_0	2	0	0	0.001	0.000	0.006	0.002	1	32	0.000
buffer_1/Queue/ _T_35_address[0]	2	0	0	0.001	0.000	0.006	0.002	1	32	0.000
buffer_1/Queue/ _T_35_address[1]	2	0	0	0.001	0.000	0.006	0.002	1	32	0.000
buffer/Queue/ _T_35_address[0]	2	0	0	0.001	0.000	0.006	0.002	1	32	0.000
fragmenter_1/ Repeater/ saved_address	2	0	0	0.002	0.000	0.017	0.002	1	32	0.000
frontend/fq/ _T_82_4_data	2	0	0	0.007	0.001	0.055	0.002	1	32	0.000
frontend/fq/ _T_82_1_data	2	0	0	0.007	0.001	0.061	0.003	1	32	0.000
frontend /fq/_T_82 _3_data	2	0	0	0.007	0.001	0.061	0.003	1	32	0.000
frontend/fq/ _T_82_2_data	2	0	0	0.007	0.001	0.062	0.003	1	32	0.000
core/csr /reg_mepc	2	0	0	0.002	0.002	0.040	0.016	1	33	0.000
frontend/fq/ _T_82_0_pc	2	0	0	0.007	0.002	0.061	0.018	1	33	0.000
frontend/fq/ _T_82_4_pc	2	0	0	0.007	0.001	0.055	0.002	1	33	0.000
frontend/fq/ _T_82_1_pc	2	0	0	0.007	0.001	0.061	0.003	1	33	0.000
frontend/fq/ _T_82_3_pc	2	0	0	0.007	0.001	0.061	0.003	1	33	0.000
frontend/fq/ _T_82_2_pc	2	0	0	0.007	0.001	0.061	0.003	1	33	0.000
core/ibuf /buf__pc	2	0	0	0.010	0.000	0.230	0.002	1	32	0.000

core/csr/ reg_mtval	2	0	0	0.001	0.002	0.041	0.015	1	34	0.000
core/csr/_T_244	2	0	0	0.008	0.006	0.067	0.045	1	58	0.000
core/csr/_T_254	2	0	0	0.008	0.006	0.068	0.046	1	58	0.000
core/wb _reg_wdata	1	0	0	0.011	0.007	0.070	0.046	1	64	0.000
dcache/s2_data	1	0	0	0.001	0.004	0.007	0.028	1	64	0.000
core/_T_525[18]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[22]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[28]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[2]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[30]	1	0	0	0.009	0.001	0.077	0.008	1	64	0.000
core/_T_525[4]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/mem_ reg_rs2	1	0	0	0.004	0.000	0.033	0.002	1	64	0.000
dtim_adapter /_T_238	1	0	0	0.001	0.000	0.004	0.001	1	64	0.000
core/_T_525[0]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[10]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[11]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[12]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[13]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[16]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[17]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[19]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[1]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[20]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[21]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[23]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[24]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[25]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[27]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[29]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[3]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[5]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[6]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[7]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[9]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
dtim_adapter /acq_data	1	0	0	0.004	0.000	0.039	0.001	1	64	0.000
dcache/ pstore1_data	1	0	0	0.014	0.001	0.144	0.004	1	64	0.000
core/_T_525[14]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[15]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/_T_525[8]	1	0	0	0.009	0.001	0.080	0.008	1	64	0.000
core/_T_525[26]	1	0	0	0.009	0.001	0.079	0.008	1	64	0.000
core/ex_reg _rs_msb_1	1	0	0	0.043	0.001	0.699	0.010	1	62	0.000
core/ex_reg _rs_msb_0	1	0	0	0.048	0.001	0.923	0.017	1	62	0.000
core/div /divisor	2	0	0	0.013	0.007	0.072	0.072	1	65	0.000

buffer/Queue_1/ _T_35_data[0]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
buffer/Queue_1/ _T_35_data[1]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
buffer_1/Queue/ _T_35_data[0]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
buffer_1/Queue/ _T_35_data[1]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
buffer_1/ Queue_1 /_T_35_data[0]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
buffer_1/ Queue_1/ _T_35_data[1]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
frontend/icache/ vb_array	2	0	0	0.004	0.001	0.028	0.009	1	64	0.000
buffer/Queue/ _T_35_data[0]	2	0	0	0.001	0.000	0.007	0.002	1	64	0.000
buffer/Queue/ _T_35_data[1]	2	0	0	0.001	0.000	0.006	0.002	1	64	0.000
core/csr/ reg_mscratch	2	0	0	0.001	0.001	0.030	0.009	1	64	0.000
core/ mem_reg_wdata	1	0	0	0.080	0.001	0.917	0.028	1	64	0.000
core/div/ remainder	2	0	0	0.024	0.013	0.429	0.159	1	130	0.000

B.3 Automatic Waypoint Guide Report

Listing B.1: PicoRV32

```
Flip_Flop: latched_branch
Nickname: latched_branch,
CL1: L1281(latched_branch||irq_state||!resetn)
SL1: L1502(latched_branch)
SL2:

Flip_Flop: latched_store
Nickname: latched_store,
CL1:
SL1:
SL2:

Flip_Flop: latched_stalu
Nickname: latched_stalu,
CL1:
SL1:
```

```
SL2:

Flip_Flop: latched_rd
Nickname: latched_rd ,
CL1:
SL1: L1320(resetn&&cpuregs_write&&latched_rd)
SL2:

Flip_Flop: cpu_state
Nickname: cpu_state ,
CL1: L1170(cpu_state==cpu_state_trap)
    L1171(cpu_state==cpu_state_fetch)
    L1172(cpu_state==cpu_state_ld_rs1)
    L1173(cpu_state==cpu_state_ld_rs2)
    L1174(cpu_state==cpu_state_exec)
    L1175(cpu_state==cpu_state_shift)
    L1176(cpu_state==cpu_state_stmem)
    L1177(cpu_state==cpu_state_ldmem)
    L1295(cpu_state==cpu_state_fetch)
    L2059(cpu_state==cpu_state_trap)
    L2060(cpu_state==cpu_state_fetch)
    L2061(cpu_state==cpu_state_ld_rs1)
    L2062(cpu_state==cpu_state_ld_rs2)
    L2063(cpu_state==cpu_state_exec)
    L2064(cpu_state==cpu_state_shift)
    L2065(cpu_state==cpu_state_stmem)
     L2066(cpu_state==cpu_state_ldmem)
SL1: L1468(cpu_state)
SL2:
```

Listing B.2: Rocket Core

```
Flip_Flop: wb_reg_inst
Nickname: wb_reg_inst ,wb_reg_inst_0 ,
CL1:
SL1:
SL2:

Flip_Flop: wb_ctrl__mem
Nickname: wb_ctrl__mem ,_T_1204 ,_T_1205 ,_T_1207 ,_T_1216
```

```
_T_1217 , _T_1209 , _T_1218 , _T_1211 , _T_1219
_T_1213 , _T_1220 , wb_xcpt , _T_1242 , _T_1243
take_pc_wb , take_pc_mem_wb , wb_dcache_miss , replay_ex_load_use ,
    _T_1184
tval_valid , unpause , io__imem_req_valid , wb_ctrl_mem ,
    wb_dcache_miss_0
wb_xcpt_0 , ibuf_io__kill , csr_io__exception ,
CL1: L168124(take_pc_mem_wb==1'h0) L168141(wb_xcpt==1'h0)
    L168296(take_pc_wb==1'h0) L168396(take_pc_wb==1'h0)
SL1: L169118(unpause)
SL2:

Flip_Flop: wb_reg_replay
Nickname: wb_reg_replay , replay_wb_common , replay_wb , _T_1242 ,
    _T_1243
take_pc_wb , take_pc_mem_wb , _T_1184 , unpause , io__imem_req_valid
replay_wb_0 , ibuf_io__kill ,
CL1: L168124(take_pc_mem_wb==1'h0) L168139(replay_wb==1'h0)
    L168296(take_pc_wb==1'h0) L168396(take_pc_wb==1'h0)
SL1: L169118(unpause)
SL2:

Flip_Flop: wb_reg_xcpt
Nickname: wb_reg_xcpt , _T_1216 , _T_1217 , _T_1218 , _T_1219
_T_1220 , wb_xcpt , _T_1242 , _T_1243 , take_pc_wb
take_pc_mem_wb , _T_1184 , tval_valid , unpause , io__imem_req_valid
wb_xcpt_0 , ibuf_io__kill , csr_io__exception ,
CL1: L168124(take_pc_mem_wb==1'h0) L168141(wb_xcpt==1'h0)
    L168296(take_pc_wb==1'h0) L168396(take_pc_wb==1'h0)
SL1: L169118(unpause)
SL2:

Flip_Flop: wb_reg_valid
Nickname: wb_reg_valid , _T_1484 , _T_1486 , _T_1255 , _T_1257
_T_1331 , _T_983 , _T_1010 , _T_1011 , _T_1204
_T_1205 , _T_1207 , _T_1451 , pstore_drain , dataArb_io_in_0_valid
dataArb_io_in_0_bits_write , _T_985 , _T_1332 , _T_1333 , _T_1334
replay_wb_rocc , replay_wb , _T_1216 , _T_1217 , _T_1209
_T_1218 , _T_1211 , _T_1219 , _T_1213 , _T_1220
```

```
wb_xcpt , _T_1242 , _T_1243 , take_pc_wb , take_pc_mem_wb
id_wb_hazard , _T_1349 , id_sboard_hazard , wb_wxd, wb_valid
wb_wen, rf_wen , _T_1184 , tval_valid , _T_1350
_T_1354 , unpause , _T_1569 , io__imem_req_valid ,
   io__imem_sfence_valid
io__imem_flush_icache , wb_xcpt_0 , replay_wb_0 , wb_reg_valid_0 ,
   wb_wen_0
ibuf_io__kill , csr_io__exception , csr_io__retire ,
CL1: L666(_T_1011==9'h0) L3763(io_in_d_bits_param==_T_983)
   L3766(io_in_d_bits_size==_T_985) L6983(_T_1486==1'h0)
   L8333(_T_1011==9'h0) L11576(_T_1011==9'h0)
      L16340(_T_1257==1'h0) L33903(io_in_d_bits_param==_T_983
         ) L33906(io_in_d_bits_size==_T_985) L38146(
         io_in_d_bits_param==_T_983) L38149(io_in_d_bits_size
         ==_T_985)
      L41393(io_in_d_bits_param==_T_983) L41396(
         io_in_d_bits_size==_T_985) L45356(io_in_d_bits_param
         ==_T_983) L45359(io_in_d_bits_size==_T_985) L49265(
         io_in_d_bits_param==_T_983)
      L49268(io_in_d_bits_size==_T_985) L56760(
         io_in_d_bits_param==_T_983) L56763(io_in_d_bits_size
         ==_T_985) L123059(_T_983==_T_983) L123077(_T_983!=2'
         h0)
      L153920(pstore_drain==1'h0) L153921(pstore2_valid==
         pstore_drain) L158044(_T_1011==9'h0) L168079(_T_1331
         ==1'h0) L168110(wb_wxd==1'h0)
      L168124(take_pc_mem_wb==1'h0) L168139(replay_wb==1'h0)
         L168141(wb_xcpt==1'h0) L168296(take_pc_wb==1'h0)
         L168386(_T_1350==1'h0)
      L168396(take_pc_wb==1'h0)
SL1: L32836(_T_965&_T_985) L32847(_T_965&_T_985) L62881(
   _T_965&_T_985) L62892(_T_965&_T_985) L66830(_T_965&_T_985
   ) L66841(_T_965&_T_985)
      L70057(_T_965&_T_985) L70068(_T_965&_T_985) L74065(
         _T_965&_T_985) L74076(_T_965&_T_985) L78028(_T_965&
         _T_985)
      L78039(_T_965&_T_985) L111248(_T_965&_T_985) L111259(
         _T_965&_T_985) L148647(_T_1011&_T_1019) L148658(
         _T_1011&_T_1019)
```

```
     L151388(_T_1011&_T_1019) L151399(_T_1011&_T_1019)
        L169118(unpause) L169739(_T_1354)
SL2:

Flip_Flop: wb_ctrl__wxd
Nickname: wb_ctrl__wxd,data_hazard_wb,wb_wxd,wb_wen,rf_wen
_T_1350,_T_1354,_T_1569,wb_ctrl_wxd,wb_wen_0

CL1: L168110(wb_wxd==1'h0) L168386(_T_1350==1'h0)
SL1: L169739(_T_1354)
SL2:

Flip_Flop: wb_ctrl__csr
Nickname: wb_ctrl__csr,csr_io__rw_cmd,
CL1: L168150(wb_ctrl__csr!=3'h0)
SL1:
SL2:
```

Appendix C

Verilog source code

C.1 Synchronized FIFO

Listing C.1: Synchronized FIFO source code

```
module sync_fifo # (
        parameter abits = 2,     //fifo depth
    parameter dbits = 8      //data width
)(
    input clock,
    input reset,
    input wr,
    input rd,
    input [dbits-1:0] din,
    output empty,
    output full,
    output [dbits-1:0] dout
);

wire db_wr, db_rd;
reg dffw1, dffw2, dffr1, dffr2;
reg [dbits-1:0] out;
reg [dbits-1:0] regarray[2**abits-1:0];
reg [abits-1:0] wr_reg, wr_next, wr_succ;
reg [abits-1:0] rd_reg, rd_next, rd_succ;
reg full_reg, empty_reg, full_next, empty_next;

always @ (posedge clock) dffw1 <= wr;
always @ (posedge clock) dffw2 <= dffw1;
assign db_wr = ~dffw1 & dffw2;
assign wr_en = db_wr & ~full;

always @ (posedge clock) dffr1 <= rd;
always @ (posedge clock) dffr2 <= dffr1;
assign db_rd = ~dffr1 & dffr2;

always @ (posedge clock) begin
  if(wr_en) regarray[wr_reg] <= din;
end

always @ (posedge clock) begin
  if(db_rd) out <= regarray[rd_reg];
end
```

```
always @ (posedge clock or posedge reset) begin
  if (reset) begin
   wr_reg <= 0;
   rd_reg <= 0;
   full_reg <= 1'b0;
   empty_reg <= 1'b1;
   end
  else begin
   wr_reg <= wr_next;
   rd_reg <= rd_next;
   full_reg <= full_next;
   empty_reg <= empty_next;
   end
 end

always @(*) begin
  wr_succ = wr_reg + 1;
  rd_succ = rd_reg + 1;
  wr_next = wr_reg;
  rd_next = rd_reg;
  full_next = full_reg;
  empty_next = empty_reg;

   case({db_wr,db_rd})
    2'b01: //read
     begin
      if(~empty) begin
        rd_next = rd_succ;
        full_next = 1'b0;
       if(rd_succ == wr_reg) empty_next = 1'b1;
       end
     end
    2'b10: //write
     begin
      if(~full)  begin
        wr_next = wr_succ;
        empty_next = 1'b0;
        if(wr_succ == rd_reg) full_next = 1'b1;
       end
     end
    2'b11:
     begin
      wr_next = wr_succ;
      rd_next = rd_succ;
     end
    default:
    endcase
 end

assign full = full_reg;
assign empty = empty_reg;
assign dout = out;
endmodule
```

Bibliography

[1] S. Agbaria, D. Carmi, O. Cohen, D. Korchemny, M. Lifshits, and A. Nadel. Sat-based semiformal verification of hardware. In *Formal Methods in Computer Aided Design*, pages 25–32, Oct 2010.

[2] Sabih Agbaria, Dan Carmi, Orly Cohen, Dmitry Korchemny, Michael Lifshits, and Alexander Nadel. An experience of complex design validation: How to make semiformal verification work. *ACM*, 2010.

[3] G Allan, G Chidolue, T Ellis, H Foster, M Horn, P James, and M Peryer. Coverage cookbook, mentor graphics. *available on-line https://verificationacademy.com*, n.d.

[4] Nina Amla and Ken L McMillan. A hybrid of counterexample-based and proof-based abstraction. In *International Conference on Formal Methods in Computer-Aided Design*, pages 260–274. Springer, 2004.

[5] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.

[6] Nathaniel Ayewah, Nikhil Kikkeri, Peter-Michael Seidel, and Sven Beyer. Challenges in the formal verification of complete state-of-the-art processors. In

Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings. 2005 IEEE International Conference on, pages 603–606. IEEE, 2005.

[7] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 1212–1221. IEEE, 2012.

[8] Jason Baumgartner, Andreas Kuehlmann, and Jacob Abraham. Property checking via structural analysis. In *International Conference on Computer Aided Verification*, pages 151–165. Springer, 2002.

[9] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 317–320. IEEE, 1999.

[10] Per Bjesse and James Kukula. Using counter example guided abstraction refinement to find complex bugs. In *Proceedings of the conference on Design, automation and test in Europe-Volume 1*, page 10156. IEEE Computer Society, 2004.

[11] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr. A universal technique for fast and flexible instruction-set architecture simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(12):1625–1639, Dec 2004.

[12] François Chollet et al. Keras, 2015.

[13] JR Burch EM Clarke and D Long. Representing circuits more efficiently in symbolic model checking. In *28th ACM/IEEE Design Automation Conference*, pages 403–407, 1991.

[14] Ben Cohen, Srinivasan Venkataramanan, Ajeetha Kumari, and Lisa Piper. *SystemVerilog Assertions Handbook:... for Dynamic and Formal Verification.* CreateSpace Independent Publishing Platform, 2015.

[15] Flavio M de Paula and Alan J Hu. Everlost: A flexible platform for industrial-strength abstraction-guided simulation. In *International Conference on Computer Aided Verification*, pages 282–285. Springer, 2006.

[16] Flavio M De Paula and Alan J Hu. An effective guidance strategy for abstraction-guided simulation. In *Design Automation Conference, 2007. DAC'07. 44th ACM/IEEE*, pages 63–68. IEEE, 2007.

[17] David L Dill. What's between simulation and formal verification? In *Proceedings 1998 Design and Automation Conference. 35th DAC.(Cat. No. 98CH36175)*, pages 328–329. IEEE, 1998.

[18] Niklas Een, Alan Mishchenko, and Nina Amla. A single-instance incremental sat formulation of proof-and counterexample-based abstraction. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 181–188. FMCAD Inc, 2010.

[19] Harry Foster, Lawrence Loh, Bahman Rabii, and Vigyan Singhal. Guidelines for creating a formal verification testplan. *Proc. DVCon*, 2006.

[20] Malay Ganai, Praveen Yalagandula, Adnan Aziz, Andreas Kuehlmann, and Vigyan Singhal. Siva: A system for coverage-directed state space search. *Journal of Electronic Testing*, 17(1):11–27, 2001.

[21] Malay K Ganai and Adnan Aziz. Rarity based guided state space search. In *Proceedings of the 11th Great Lakes symposium on VLSI*, pages 97–102. ACM, 2001.

[22] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.", 2017.

[23] Saurav Gorai, Saptarshi Biswas, Lovleen Bhatia, Praveen Tiwari, and Raj S Mitra. Directed-simulation assisted formal verification of serial protocol and bridge. In *Design Automation Conference, 2006 43rd ACM/IEEE*, pages 731–736. IEEE, 2006.

[24] Aarti Gupta. Formal hardware verification methods: A survey. In *Computer-Aided Verification*, pages 5–92. Springer, 1992.

[25] Kshitiz Gupta et al. Automatic generation of coverage directives targeting signal relationships by statically analyzing rtl. Master's thesis, The University of Texas at Austin, 2017.

[26] Ziyad Hanna, Craig Franklin Deaton, Kathryn Drews Kranen, Björn Håkan Hjort, and Lars Lundgren. Guided exploration of circuit design states, June 21 2016. US Patent 9,372,949.

[27] Klaus Havelund and Natarajan Shankar. Experiments in theorem proving and model checking for protocol verification. In *International Symposium of Formal Methods Europe*, pages 662–681. Springer, 1996.

[28] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[29] John A Hertz. *Introduction to the theory of neural computation*. CRC Press, 2018.

[30] C Richard Ho, Michael Theobald, Brannon Batson, J Grossman, Stanley C Wang, Joseph Gagliardo, Martin M Deneroff, Ron O Dror, and David E Shaw. Post-silicon debug using formal verification waypoints. In *Design and Verification Conf*, 2009.

[31] Pei Hsin Ho, Thomas Shiple, Kevin Harer, James Kukula, Robert Damiano, Valeria Bertacco, Jerry Taylor, and Jiang Long. Smart simulation using collaborative formal and simulation engines. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 120–126. IEEE Press, 2000.

[32] Bob Hu. Hummingbird e200 opensource processor core. `https://github.com/SI-RISCV/e200_opensource`, 2018.

[33] Brian Keng and Andreas Veneris. Automated debugging of missing input constraints in a formal verification environment. In *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pages 101–105. IEEE, 2012.

[34] Olof Kindgren. Serv. `https://github.com/olofk/serv`, 2018.

[35] Vladik Ya Kreinovich. Arbitrary nonlinearity is sufficient to represent all functions by neural networks: a theorem. *Neural networks*, 4(3):381–383, 1991.

[36] Howard E Krohn. Design verification of large scientific computers. In *Proceedings of the 14th Design Automation Conference*, pages 354–361. IEEE Press, 1977.

[37] Daniel Kröning. *Formal verification of pipelined microprocessors.* PhD thesis, Universitat des Saarlandes, 2001.

[38] Andreas Kuehlmann and Cornelis AJ van Eijk. Combinational and sequential equivalence checking. In *Logic synthesis and Verification*, pages 343–372. Springer, 2002.

[39] Ulrich Kühne, Sven Beyer, Jorg Bormann, and John Barstow. Automated formal verification of processors based on architectural models. In *Formal Methods in Computer-Aided Design (FMCAD), 2010*, pages 129–136. IEEE, 2010.

[40] Pradeep Kumar Nalla, Raj Kumar Gajavelly, Jason Baumgartner, Hari Mony, Robert Kanzelman, and Alexander Ivrii. The art of semi-formal bug hunting.

In *Computer-Aided Design (ICCAD), 2016 IEEE/ACM International Conference on*, pages 1–8. IEEE, 2016.

[41] Kuntal Nanshi and Fabio Somenzi. Guiding simulation with increasingly refined abstract traces. In *Proceedings of the 43rd annual Design Automation Conference*, pages 737–742. ACM, 2006.

[42] Minh D Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. Unbounded protocol compliance verification using interval property checking with invariants. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(11):2068–2082, 2008.

[43] Ankur Parikh, Weixin Wu, and Michael S Hsiao. Mining-guided state justification with partitioned navigation tracks. In *Test Conference, 2007. ITC 2007. IEEE International*, pages 1–10. IEEE, 2007.

[44] JasperGold Formal Verification Platform. Cadence inc.

[45] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with isa-formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[46] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[47] Sharukh Shahajahan Shaikh. Implementation of verification methodologies. Master's thesis, The University of Texas at Austin, 2018.

[48] Avinash Sharma. Understanding activation functions in neural networks. *medium. com*, 2017.

[49] Donald F Specht. A general regression neural network. *IEEE transactions on neural networks*, 2(6):568–576, 1991.

[50] Rob Sumners, Jayanta Bhadra, and Jacob Abraham. Automatic validation test generation using extracted control models. In *VLSI Design, 2000. Thirteenth International Conference on*, pages 312–317. IEEE, 2000.

[51] Shinya Takamaeda-Yamazaki. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *Applied Reconfigurable Computing*, volume 9040 of *Lecture Notes in Computer Science*, pages 451–460. Springer International Publishing, Apr 2015.

[52] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanović. The risc-v instruction set manual volume ii: Privileged architecture version 1.10. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2017*, 2017.

[53] Andrew Waterman, Yunsup Lee, David Patterson, and Krste Asanovic. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2.2, 2017.

[54] C Wolf. End-to-end formal isa verification of risc-v processors with riscv-formal, 2017.

[55] C Wolf. Picorv32-a size-optimized risc-v cpu. *github. com/cliffordwolf/picorv32*, 2018.

[56] Praveen Yalagandula, Vigyan Singhal, and Adnan Aziz. Automatic lighthouse generation for directed state space search. In *Proceedings of the conference on Design, automation and test in Europe*, pages 237–242. ACM, 2000.

[57] C Han Yang and David L Dill. Validation with guided search of the state space. In *Proceedings of the 35th annual Design Automation Conference*, pages 599–604. ACM, 1998.

[58] Jun Yuan, Jian Shen, Jacob Abraham, and Adnan Aziz. On combining formal and informal verification. In *International Conference on Computer Aided Verification*, pages 376–387. Springer, 1997.