The Essence of Verilog (Artifact Document)

1 Introduction

Our accompanying paper describes a core language, λ_V, that defines formal operational semantics for Verilog.

This artifact provides a Java implementation of λ_V, which includes an λ_V interpreter based on its formal semantics and a frontend that converts Verilog to λ_V.

Furthermore, this artifact offers an evaluation environment that allows for the reproduction of the results stated in our companion paper. Specifically:

- In our contribution, we describe that λ_V can detect semantic bugs in real-world Verilog simulators and expose ambiguities in Verilog's standard specification. While Sections 5.1 and 5.2 of our paper discuss some bugs and ambiguities, space constraints limit the inclusion of all findings. Sections 3.1 and 3.2 in this document elaborate on the unaddressed bugs and ambiguities, providing instructions and test cases for reproducing them.

- In Section 4 of our paper, we describe that λ_V is tested in terms of totality for desugaring real-world Verilog programs and its conformance to the Verilog specification. Our testing involves two distinct test suites:

  1. **Language Features Suite**. This suite comprises 824 test cases designed to comprehensively cover various Verilog language features. Section 3.3 provides detailed instructions to reproduce results mentioned in Section 4 of our paper related to this test suite. It includes the results shown in Table 3 in our paper, and the validation regarding whether λ_V, *Icarus Verilog*, and *Verilator* can pass this test suite. Note that *Icarus Verilog* and *Verilator* are two reputable open-source Verilog simulators, and their results on this test suite serve as as the closest approximation to the absent formal Verilog specification, against which we gauge λ_V's conformance.
  2. **Real-World Programs Suite**. This second test suite draws from real-world programs that tightly combine language features. Section 3.4 provides instructions to verify the results mentioned in Section 4 of our paper regarding λ_V's ability to pass this test suite.

Before proceeding with the evaluation of these claims, let's first set up our artifact and perform a sanity check.

2 Getting Started

2.1 Basic Requirements

- **Hardware**. In our paper, all experiments were conducted on a mid-range desktop or laptop with 4-6 cores and 8GB-16GB of memory. Therefore, we believe that the results can be reproduced on most devices commonly used for daily work. However, for better performance, we recommend a configuration with 6 cores and 16GB of memory.
- **Environment**. To simplify the setup of our artifact, we have provided a Docker image that includes the fully configured evaluation environment. This Docker image includes the installation of JDK 17, *Icarus Verilog*, and *Verilator*. Hence, the only requirement is to have Docker installed on your machine. Our Docker image has been tested on Linux. If you encounter any issues on Windows or macOS, we recommend using a Linux device for optimal compatibility.

2.2 Artifact Setup

To get started, please install Docker on your system. If you have already installed Docker, you can skip this step. Follow the instructions below based on your operating system:

- For Mac or Windows users, please visit `https://docs.docker.com/get-docker/` and follow the instructions to install Docker Desktop.
- For Linux users, we recommend installing the Docker engine by following the instructions at `https://docs.docker.com/engine/install/`.

Once Docker is installed, you can easily set up our artifact by following these steps:

1. Load the Docker image into your system. Note that for Mac or Windows users, you need to first start Docker Desktop to enable the docker command. Then, type the following command in your terminal:

   ```
   $ docker load --input lv-artifact.tar.gz
   ```

   Please note that this step may take several minutes due to the large size of the Docker image. (Note: If you are using Finch instead of Docker, please add the option `--all-platforms` to the command.)

2. Launch a container from the loaded image. Specify the image name as `lv:oopsla23` and choose a container name, such as `lv`:

   ```
   $ docker run --name lv -it lv:oopsla23
   ```

Once the container is launched, you will enter an interactive shell of the container. To exit the interactive shell, use the command `exit` or `Ctrl-d`. If you want to re-enter the container after exiting, use the following command to restart and re-enter the same container:

```
$ docker start -i lv
```

2.3 Artifact Content

Once you have launched the container and entered the interactive shell, please use the `cd` command to navigate to the home directory (`/root`). You can view the content of our artifact by executing the `ls` command:

```
$ cd && ls
```

In the home directory, you will find the following five directories:

- `qihe/`: This directory contains our implementation of λ_V (*including its full source code*), which is a Java project using Gradle as the build tool. Additionally, this directory houses our two essential test suites. The test cases for the Language Features Suite are located in the following path:

  ```
  qihe/lambdav/src/test/resources/testcases/verilog/iverilog
  ```

  Meanwhile, the test cases for the Real-World Programs Suite can be found here:

  ```
  qihe/lambdav/src/test/resources/testcases/verilog/realwolrd
  ```

  For ease of reference when inspecting test cases in Section 3, we've exported these paths as environment variables, namely `LANG_SUITE` and `RW_SUITE`. You can verify their definition using the following commands:

  ```
  $ echo $LANG_SUITE
  /root/qihe/lambdav/src/test/resources/testcases/verilog/iverilog
  $ echo $RW_SUITE
  /root/qihe/lambdav/src/test/resources/testcases/verilog/realworld
  ```

- `iverilog/` and `verilator/`: These directories contain the source code of two other popular open-source Verilog simulators that we will compare with in our evaluation. They have been compiled and installed using their latest stable versions at the time of our paper submission.

- `scripts/`: This directory provides easy-to-use scripts that facilitate the reproduction of the results presented in our paper.

- `demo/`: This directory contains some cases for demonstration purposes.

2.4 Sanity Check

The evaluation of our artifact involves running the λ_V interpreter and two simulators: *Icarus Verilog* and *Verilator*. We have compiled and installed them all in our Docker image. Let's perform a sanity check to ensure they can run.

To check λ_V, we provide a script called `lv` to execute the main class of our Java project, which is placed in the `/root/qihe/lambdav/build/install/lv/bin` directory. Because we have exported this directory to the `PATH` environment variable, you can run the `lv` script directly:

```
$ lv --help
```

You will see the usage information displayed:

```
Usage: lv [-chiVx] [-o=<out>] [--seed=<seed>] [-t=<till>] [SOURCE...]
lv compiler and interpreter
      [SOURCE...]     the LambdaV program to interpret or the Verilog program
                        to compile
  -c, --compile       compile a Verilog program into a LambdaV program
  -h, --help          Show this help message and exit.
  -i, --interpret     interpret a LambdaV program
  -o, --out=<out>     output a LambdaV program into a file
      --seed=<seed>   set the seed of random number
  -t, --till=<till>   interpret the program for some specific time steps
  -V, --version       Print version information and exit.
  -x, --explore       explore the state space of a LambdaV program
Example: lv -c in.v -o out.lv  (compile Verilog to LambdaV program)
  or:  lv -i a.lv -t 4         (interpret a LambdaV program for 4 time steps)
  or:  lv -x a.lv -t 4         (explore the state space of a LambdaV program)
  or:  lv -ci in.v -t 4        (compile and interpret a Verilog program)
  or:  lv a.lv -o b.lv         (format a LambdaV file)
```

To further verify the functionality, we have included a classic "Hello World" Verilog program. You can execute the following command to compile the Verilog program into an equivalent λ_V program and interpret it:

```
$ lv -ci /root/demo/hello.v -o /tmp/hello.lv
```

In this command, the `-c` option instructs the `lv` to compile the Verilog program `hello.v` into an equivalent λ_V program. The `-o /tmp/hello.lv` option specifies the output path for the compiled λ_V program, which in this case is `/tmp/hello.lv`. You can omit this option if you do not want to inspect the compiled λ_V program. After compilation, the `lv` will interpret the compiled program due to the `-i` option. The expected output of this interpretation should be:

```
Hello, world!
```

The generated λ_V program will look like this:

```
$ cat /tmp/hello.lv
var .dumb : b0

proc main.$p0 .dumb = $display("Hello, world!")
```

To check the executables of *Icarus Verilog* and *Verilator*, you can use the following commands and observe the corresponding results:

```
$ iverilog -h
Usage: iverilog [-EiRSuvV] [-B base] [-c cmdfile|-f cmdfile]
                [-g1995|-g2001|-g2005|-g2005-sv|-g2009|-g2012] [-g<feature>]
                [-D macro[=defn]] [-I includedir] [-L moduledir]
                [-M [mode=]depfile] [-m module]
                [-N file] [-o filename] [-p flag=value]
                [-s topmodule] [-t target] [-T min|typ|max]
                [-W class] [-y dir] [-Y suf] [-l file] source_file(s)
```

```
See the man page for details.

$ verilator --version
Verilator 5.008 2023-03-04 rev v5.008
```

2.5 Run Single File

In Section 3, you may need to run different tools on individual Verilog file to reproduce the results mentioned in our paper. To simplify this process, we provide an easy-to-use script, `/root/scripts/run.sh`, which allow you to run a single Verilog file. Since we have exported `/root/scripts` to the `PATH` environment variable, you can simply run the script and see its usage as follows:

```
$ run.sh
Usage: run.sh lv <file> [args]...
   or: run.sh iverilog <file>
   or: run.sh verilator <file> [timeout]
```

The first argument indicates which tool to run (λ_V, *Icarus Verilog*, or *Verilator*), and the second argument `<file>` specifies the path of the Verilog program to be executed. The remaining optional arguments (e.g., `[args], [timeout]`) depend on the selected tool. For `lv`, these arguments will be forwarded to the `lv` command. Therefore, executing `run.sh lv <file> [args]...` is roughly equivalent to executing:

```
$ lv -ci <file> [args]...
```

When using verilator, the third optional argument allows you to specify a timeout value for killing the execution, as verilator may not automatically finish in certain cases. However, you can omit this argument as we have already set a sensible default timeout of one second.

As an example, you can easily run the "Hello World" program using λ_V by executing the following command:

```
$ run.sh lv /root/demo/hello.v -o /tmp/hello.lv
```

As explained above, this command is roughly equivalent to executing:

```
$ lv -ci /root/demo/hello.v -o /tmp/hello.lv
```

Therefore, the output should be "`Hello, world!`", and the compiled λ_V program will be stored in `/tmp/hello.lv`.

Additionally, you can run the example using *Icarus Verilog* and *Verilator* and observe the output "`Hello, world!`" by executing the following commands:

```
$ run.sh iverilog /root/demo/hello.v
$ run.sh verilator /root/demo/hello.v
```

Please note that executing `run.sh verilator` may take longer than other commands because *Verilator* converts Verilog programs into `C++` files and then compiles them, which can be a time-consuming process.

3 Step-by-Step Instructions

In this section, we aim to reproduce the results described in our paper.

First, as stated in our contribution, we utilized λ_V to detect semantic bugs in real-world simulators and find ambiguities in Verilog specification. Therefore, we present and discuss the test cases that can reproduce those bugs and ambiguities in Section 3.1 and Section 3.2, respectively.

Second, as explained in Section 4 of our paper, we tested the totality and conformance properties of λ_V. This involved: (1) running the λ_V interpreter and comparing its results with those of *Icarus Verilog* and *Verilator* on the Language Features Suite, a comprehensive collection of 824 test cases; (2) running the λ_V interpreter on the Real-World Programs Suite. Therefore, in Section 3.3 and 3.4, we reproduce the results obtained during this testing process, respectively. Specifically, we include the following materials that are consistent with the descriptions in our paper:

- Table 3 (in our paper), which presents the distribution of representative Verilog features across the Language Features Suite.
- The confirmation that *Icarus Verilog* fails to pass three test cases in the Language Feature Suite and *Verilator* fails to pass over 100 test cases (as described in Section 4.2 of our paper).
- The identification of specific test cases within the Language Features Suite where λ_V encounters failures. It is important to note that these failures do not compromise the totality and conformance properties of λ_V. Some failures are attributed to bugs in *Icarus Verilog* or ambiguities in the Verilog specification, as described in Section 4.2 of our paper and further discussed in Section 3.1 and Section 3.2 of this document. Additionally, some failures are the result of either erroneous test cases or the stricter type checker employed in our evaluation, as explained in Section 4.2 of our paper. Figure 1 provides a summary of these cases, encompassing bugs, ambiguities, and failures caused by erroneous test cases or our stricter type checker.
- The confirmation that λ_V can pass test cases in the Real-World Programs Suite.

3.1 Bugs

As mentioned in Section 5.1 of our paper, λ_V identified real-world semantic bugs that were even erroneously interpreted by the two most popular open-source Verilog simulators, *Icarus Verilog* and *Verilator*. These bugs fall into three categories including context-determined expressions, hidden data races, and loss of newest results.

3.1.1 Context-Determined Expressions

In Section 5.1.1 of our paper, we discussed the bug related to context-determined expressions in *Icarus Verilog*. In this section, we explain how to confirm this bug.

Since this bug is not a new discovery during our testing process, you can refer to `https://github.com/steveicarus/iverilog/issues/20` to verify that an older version of *Icarus Verilog* incorrectly interprets the context-determined expressions as discussed in our paper. Figure 2 shows the content of the URL, which includes the test case that triggers the bug. The expected result of that case should be `0`, but the older version of *Icarus Verilog* outputs `1` instead. This bug has been confirmed by the developer.

Root Cause	Short Description	Cases	Number	Running Result of LV	Related Sections
iv bug	hidden data race	see scripts/data-race-cases.list	99	most passed, several depend on scheduling	Section 5.1.2 of the paper (Line 915-971) and Section 3.1.2 of this document
	loss of newest results	pr1662508.v	1	depend on scheduling	Section 3.1.3 of this document
ambiguity	undefined initial output of delayed driver	ldelay1.v	1	failed	Section 5.2 of the paper (Line 983-997) and Section 3.2.1 of this document
	procedural continuous assignments	force_lval_part.v	2	depend on scheduling	Section 3.2.2 of this document
		pr2943394.v			
	unmatched port connections	gate_connect1.v	1	failed	Section 3.2.3 of this document
	named event as event exprs	event_list.v	3	passed	Section 3.2.4 of this document
		br_gh508b.v			
		pr2788686.v			
erroneous case	using == rather than ===	casex3.9D.v	5	failed	Section 4.2 of the paper (Line 849-857) and Section 3.3 of this document
		casex3.9E.v			
		casez3.10E.v			
		casez3.10D.v			
		casez3.10C.v			
	shared loop variable	br1000.v	1	depend on scheduling	Section 3.3 of this document
	arithmetic overflow	sqrt32.v	1	depend on generated random numbers	
stricter type check	implicit convertion of port direction	bufif.v	1	failed	Section 4.2 of the paper (Line 858-863) and Section 3.3 of this document
	inout with different netKinds	br1001.v	1	failed	

Figure 1: Test cases within the Language Features Suite that λ_V fails to pass due to: (1) bugs in *Icarus Verilog* and *Verilator*; (2) ambiguities in Verilog; (3) erroneous cases; (4) our stricter type checker.

Fortunately, λ_V can handle these expressions correctly. We have saved the case from the GitHub issue URL mentioned above in `/root/demo/ce.v`. You can run it using `run.sh` as instructed in Section 2.5:

```
$ run.sh lv /root/demo/ce.v
```

Executing this command will output `0`, indicating that λ_V produces the correct result for this test case.

3.1.2 Hidden Data Race

The hidden data race bug, as discussed in Section 5.1.2 of our paper, can be reproduced in both *Icarus Verilog* and *Verilator*. In this section, we provide instructions on how to reproduce it.

Data races are often hidden by *Icarus Verilog* and *Verilator*, leading programmers to overlook them when writing test cases. As a result, our Language Features Suite contains 99 test cases that exhibit data races, which can be used to confirm the hidden data race bug in *Icarus Verilog* and *Verilator*. These test cases are listed in the file `/root/scripts/data-race-cases.list`. You can run them using the script `run.sh` introduced in Section 2.5.

For example, let's consider the first test case, `always3.1.5A.v`, in `data-race-cases.list`. You can inspect the content of this test case by running the command:

```
$ vim $LANG_SUITE/always3.1.5A.v
```

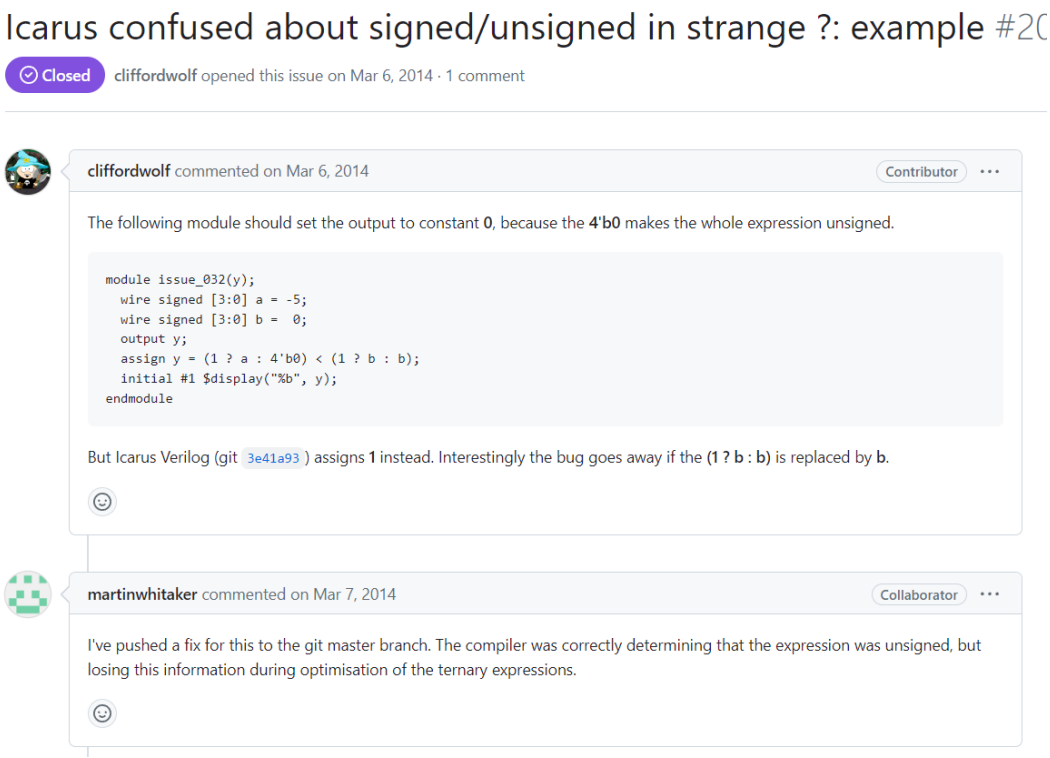

Figure 2: The bug report of context-determined expression in the older version of *Icarus Verilog*

, and you will find that there is a data race on the variable `value1` (line 30 reads the variable, and line 40 writes the variable) (currently, you may well ignore the synchronous statement `#0` in line 39, whose usage is explained later). As a result, the read of `value1` can produce both the old value (`4'd4`) or the new value (`4'd5`). Therefore, running this test case may output either `PASSED` or `FAILED` according to the Verilog specification.

You can run the following commands to see the results of different tools on this case:

```
$ run.sh iverilog $LANG_SUITE/always3.1.5A.v
PASSED
/root/test/always3.1.5A.v:34: $finish called at 5 (1s)
$ run.sh verilator $LANG_SUITE/always3.1.5A.v
PASSED
- temp.v:34: Verilog $finish
$ run.sh lv $LANG_SUITE/always3.1.5A.v # If PASSED
PASSED
$ run.sh lv $LANG_SUITE/always3.1.5A.v # If FAILED
FAILED - always 3.1.5A always if ( constant) statement
```

You will observe that *Icarus Verilog* and *Verilator* only output `PASSED` because they hide the data race, regardless of how many times the command is executed, while λ_V outputs both `PASSED` and `FAILED` in different runs as expected. Please note that the final result of the λ_V interpreter is indeterminate due to its random scheduler. *You may need to run the command several times to see both results.*

Because it would be verbose to explain all the other data race cases individually, we provide a general description of their expected results. On those cases, both `PASSED` and `FAILED` are possible outputs due to data races. However, *Icarus Verilog* and *Verilator* will hide these data races and output only `PASSED`. Fortunately, λ_V will expose the data races and may output either `PASSED` or `FAILED` depending on its random scheduling.

Note that the indeterminate output of λ_V on the data race cases can cause a much larger number of failed cases when running λ_V on the entire test suite, which can make the output of our script that shows failed cases and their running log too long to inspect manually. To address this issue, we have made efforts to eliminate data races in as many test cases as possible by adding synchronous statements `#0` at strategic locations, so that λ_V can stably pass them. For example, in line 39 of `always3.1.5A.v`, we added `#0` to eliminate data races. However, data races in cases such as `sched2.v`, `memidx.v`, and `pr2986528.v` cannot be eliminated. Therefore, in Figure 1, the cases with data races are marked as "most passed, several depend on scheduling".

Recall that you can successfully observe the data races when running the example test case above. It is because we have removed the synchronous statements (`#0`) during the execution of the script `run.sh`.

Next, we reproduce the results of the simple case presented in Section 5.1.2 of our paper, which is saved as `/root/demo/same-diff.v`. You can execute the following commands to observe the behavior of *Icarus Verilog* and *Verilator*:

```
$ run.sh iverilog /root/demo/same-diff.v
same
$ run.sh verilator /root/demo/same-diff.v
same
```

Running these commands will demonstrate that *Icarus Verilog* and *Verilator* can only output `same`, regardless of how many times those commands are executed. In contrast, running the following commands will show the results yielded by λ_V:

```
$ run.sh lv /root/demo/same-diff.v # One possible result
same
$ run.sh lv /root/demo/same-diff.v # The other possible result
different
```

By using λ_V, you will observe that it can print both `same` and `different`, as depicted in our paper. Note again that you may need to run the command several times to see both results.

3.1.3 Loss of Newest Results

The loss of newest results is a type of simulator bug that occurs when the simulator fails to carry out the effect of a newly generated event. In our Language Features Suite, the test case `pr1662508.v` demonstrates this bug in both *Icarus Verilog* and *Verilator*. You can inspect the content of this test case by running the command:

```
$ vim $LANG_SUITE/pr1662508.v
```

In this test case, both `PASSED` and `FAILED` can be displayed according to the Verilog specification, while *Icarus Verilog* and *Verilator* can only print `PASSED`. Let's explore how `FAILED` can be displayed.

The `always` block in line 47 is sensitive to the negedge of the `clk`. When running the test case, the `clk` is initialized with the default value `x`. Then the `initial` block in line 25 is executed concurrently with other blocks, and its first iteration of the for-loop (line 26-29) sets `clk` to 0 (line 27). This generates an event that indicates the `clk` has a negedge transition from `x` to `0`, according to Table 9-1 of the Verilog specification [1]. This event could trigger the `always` block in line 47. Once triggered, the `always` block asserts that `read_bus === addr[3:0]*addr[3:0]`. However, at this point, both `addr` and `read_bus` have just been initialized with their default values, `0` and `x`, respectively. Therefore, the assertion fails, resulting in the printout of `FAILED`.

Due to the scheduler of *Icarus Verilog* and *Verilator* always missing the effect of the first event on the negedge of the clock, they fail to produce the result where `FAILED` can indeed be displayed in this test case. In contrast, λ_V can faithfully reproduce both possible results as expected by the Verilog specification. To observe the results of different tools on this test case, you can use the following commands:

```
$ run.sh iverilog $LANG_SUITE/pr1662508.v  # PASSED
$ run.sh verilator $LANG_SUITE/pr1662508.v # PASSED
$ run.sh lv $LANG_SUITE/pr1662508.v        # Both PASSED and FAILED
```

3.2 Ambiguities

As mentioned in Section 5.2 of our paper, λ_V has helped identify seven cases with certainty where inconsistent results are produced by different simulators due to the ambiguities in Verilog's standard specification. These cases fall into four types of scenarios including timing-controlled assignments, procedural continuous assignments, unmatched port connections, and even the syntax regarding named events. Because *Verilator* cannot pass many of those cases, we will only use *Icarus Verilog* for illustrating the ambiguities in this section.

3.2.1 Timing-controlled Assignment

The ambiguity related to timing-controlled assignments was discussed in Section 5.2 of our paper. In our Language Features Suite, the `ldelay1.v` test case can reproduce the inconsistency between λ_V and other simulators due to this ambiguity.

You can inspect the content of the test case by using the following command:

```
$ vim $LANG_SUITE/ldelay1.v
```

In line 28 of `ldelay1.v`, there is a driver (a logic AND gate `and #6 (q, a, b)`) that assigns the result of `a & b` to `q` with a delay of `6` time steps. In the initial block at line 42, `a` and `b` are set to `0` and `1` at time step 0, respectively, and `q` will be set to `0` at time step 6 due to the delay in the driver. At time step 5, the value of `q` is still its default initialization value, and line 48 calls the `ok` task to assert that the value of `q` is `1'bx`.

However, as explained in our paper, the Verilog specification [1] states that "Nets with drivers shall assume the output value of their drivers", but it does not clarify what should happen if the driver's output value is not available during initialization. In this case, at time step 5, the AND gate will not drive `q` until time step 6. Therefore, the value of `q` is undefined, and λ_V assumes its value to be `z`, while *Icarus Verilog* assumes its value to be `x`. As a result, λ_V outputs `FAILED`, while *Icarus Verilog* outputs `PASSED`. To observe the results of different tools on this test case, you can use the following commands:

```
$ run.sh iverilog $LANG_SUITE/ldelay1.v
PASSED
$ run.sh lv $LANG_SUITE/ldelay1.v
00000000000000000005: FAILED: q=z, expect x
```

Next, we reproduce the results of the simple case presented in Section 5.2 of our paper, which is saved as `/root/demo/assume-dr.v`. To observe the results as depicted in our paper, you can run the following commands:

```
$ run.sh iverilog /root/demo/assume-dr.v
x
$ run.sh lv /root/demo/assume-dr.v
1
```

3.2.2 Procedural Continuous Assignments

In our paper, we only mentioned the identification of this type of ambiguities without any further explanation. We introduce it here and provide tests to reproduce them for the completeness of our artifact.

Procedural continuous assignments, such as "`force lhs = rhs`", are statements used to force the left-hand side's value to be the right-hand side's value. The ambiguity regarding procedural continuous assignments in Verilog stems from the lack of specification on when the right-hand side's value should be assigned to the left-hand side after executing a procedural continuous assignment statement

Intuitively, one might expect the right-hand side's value to be immediately assigned to the left-hand side upon executing a procedural continuous assignment. However, in λ_V, the right-hand side's value is assigned to the left-hand side in a new concurrent thread, which may not result in an immediate assignment.

We made this design choice for the following reason. During our modeling of the semantics of procedural continuous assignments, we discovered that they share the same underlying

mechanism as plain continuous assignments. As the name suggests, procedural continuous assignments essentially create a driver that continuously assigns values to the left-hand side, just like a plain continuous assignment does. The difference is that the driver is created at runtime by a procedural continuous assignment while it is created at the preprocessing phase by a continuous assignment. Therefore, to simplify λ_V, we decided to use the same mechanism for both types of assignments.

When discussing the semantics of continuous assignments, there are two steps involved: (1) creating a concurrent thread for each continuous assignment to serve as a driver, continuously assigning the right-hand side's value to the left-hand side (note that such thread is represented in other equivalent way in our paper); and (2) starting these concurrent threads. Consequently, the right-hand side's value of a continuous assignment is not immediately assigned to the left-hand side.

This mechanism works well for continuous assignments because they are intended to run concurrently, and there is no need to immediately evaluate their right-hand sides as the variables in the right-hand sides may not have been initialized yet. For example:

```
1 assign a = b;
2 assign b = 1;
```

In this case, when the thread is created for the first continuous assignment, the value of `b` has not yet been assigned `1`. Therefore, there is no need to evaluate `b` at that point.

Following the same mechanism, when a procedural continuous assignment is executed in λ_V, it first creates a concurrent thread to act as a driver and then allows the thread to run. As a result, in λ_V, the right-hand side's value is not immediately assigned to the left-hand side. In contrast, *Icarus Verilog* chooses to immediately assign the right-hand side's value upon executing a procedural continuous assignment. Given the ambiguity in Verilog's specification regarding this aspect, we believe that both approaches to treating procedural continuous assignments are reasonable.

In our Language Features Suite, we have two cases that can demonstrate the inconsistent behavior between λ_V and *Icarus Verilog* regarding this ambiguity: `force_lval_part.v` and `pr2943394.v`, as listed in Figure 1.

`force_lval_part.v` In the case of `force_lval_part.v`, the line `force value[0 +: 4] = in` in line 16 continuously assigns the value of `in` (i.e., `4'bzx10`) to `value`. The subsequent `if` statement in line 17 checks whether `value` has been assigned `4'bzx10`. Since *Icarus Verilog* immediately assigns the right-hand side's value, it can pass the test:

```
$ run.sh iverilog $LANG_SUITE/force_lval_part.v
PASSED
```

However, in λ_V, there are three possible results:

```
$ run.sh lv  $LANG_SUITE/force_lval_part.v # Result 1
Failed: force value, expected 4'bzx10, got 1001
$ run.sh lv  $LANG_SUITE/force_lval_part.v # Result 2
Failed: force value, expected 4'bzx10, got zx10
$ run.sh lv  $LANG_SUITE/force_lval_part.v # Result 3
PASSED
```

The first two commands generate schedules that fail the test, but the command using seed 0 prints the old value (`1001`) of `value`, while the command using seed 1 prints the new value (`zx10`) of `value`. This difference arises from whether the right-hand side's value of the force

statement is assigned to the left-hand side before or after the `$display` statement in line 17 is executed. The third command produces the same result as *Icarus Verilog* by generating a schedule that immediately updates `value`.

`pr2943394.v` In the case of `pr2943394.v`, line 20 uses the force statement to continuously assign `4'b1001` to `val`, while line 21 uses the release statement to remove this assignment. Due to the fact that λ_V does not immediately assign the right-hand side's value to the left-hand side, `val` could still have the old value (`4'b0110`) and subsequently fail the assertion in lines 13-17, printing "`Failed release of forced sig, expected 4'b1001, got 0110`". Since there are two additional `force`/`release` pairs in lines 47-48 and 74-75, each of which may or may not print a failure message, there are a total of $2^3 = 8$ possible outcomes. We just demonstrate λ_V's results where all failure messages are printed and where the case passes:

```
$ run.sh lv $LANG_SUITE/pr2943394.v # Result with all failure messages
Failed release of forced sig, expected 4'b1001, got 0110
Failed pv release of forced sig, expected 4'b1011, got 1001
Failed release of forced sig, expected 2.0, got 1.0
$ run.sh lv $LANG_SUITE/pr2943394.v # Result where the case passes
PASSED
```

On the other hand, *Icarus Verilog* only prints `PASSED` on this test case:

```
$ run.sh iverilog $LANG_SUITE/pr2943394.v
PASSED
```

3.2.3 Unmatched Port Connections

In our paper, we mentioned the identification of this type of ambiguity without providing further details. Here, we explain it in more detail and provide tests to reproduce it for the completeness of our artifact.

The ambiguity regarding unmatched port connections in Verilog arises from the lack of specification on how to handle connections between ports of different widths in gate instantiations. In our Language Features Suite, the test case `gate_connect1.v` demonstrates the inconsistency between λ_V and *Icarus Verilog*.

You can inspect the content of `gate_connect1.v` by using the following command:

```
$ vim $LANG_SUITE/gate_connect1.v
```

In lines 11-13 of the file, the outputs of three `buf` gates have a width of 1 bit, while the inputs connected to them have widths of either 32 bits or 2 bits. Due to the ambiguity about this situation, λ_V will throw a `TypeCheckException` to warn about this usage when running this case. On the other hand, *Icarus Verilog* will truncate the inputs to match the 1-bit width requirement of the output ports. Consequently, λ_V fails to pass this case, while *Icarus Verilog* succeeds. To observe the results, you can use the following commands:

```
$ run.sh lv $LANG_SUITE/gate_connect1.v       # Exception thrown
$ run.sh iverilog $LANG_SUITE/gate_connect1.v # PASSED
```

3.2.4 Syntax of Named Events

In our paper, we mentioned the identification of this type of ambiguity without providing further details. Here, we explain it in more detail for the completeness of our artifact.

In Verilog, we can declare a named event using `event ev` and then use a named event control `@ev` to guard a statement, waiting for it to be triggered by another statement `-> ev`. However, the syntax related to named events has an ambiguity according to the specification.

The syntax of named event control is defined as follows:

```
   event_control ::= @ hierarchical_event_identifier
                   | @ ( event_expression )
event_expression ::= expression
                   | posedge expression
                   | negedge expression
                   | event_expression or event_expression
                   | event_expression , event_expression
```

where the `hierarchical_event_identifier` refers to a named event in the program.

According to the syntax, a named event should not be surrounded by brackets when used in an `event_control` because the production of `event_expression` does not allow it. However, there is a case in Section 18.1.6 (page 329) of the specification [1] that contradicts this rule. The related code snippet is as follows:

```
1 event do_dump;
2 initial @(do_dump)
3     forever #10000 $dumpall;
```

In this case, it can be seen that the named event `do_dump` in the event control is surrounded by brackets, which goes against the defined syntax.

Initially, when implementing the front-end of λ_V, we followed the syntax specified in the appendix of the Verilog specification. As a result, λ_V failed three cases in the Language Features Suite where named events were surrounded by brackets, as listed in Figure 1, while *Icarus Verilog* can pass them. Upon inspecting those cases, we discovered the ambiguity regarding named events, as explained above. To make λ_V more robust, we modified our front-end and λ_V to allow named events to be involved in event expressions. As a result, λ_V can now pass those test cases successfully. To observe the results, you can use the following commands:

```
$ run.sh lv $LANG_SUITE/event_list.v
PASSED
$ run.sh lv $LANG_SUITE/br_gh508b.v
00000000000000000001: A
00000000000000000002: B
00000000000000000002: A
00000000000000000003: B
PASSED
$ run.sh lv $LANG_SUITE/pr2788686.v
PASSED
```

3.3 Results on Language Features Suite

This section provides instructions to reproduce the results related to the Language Features Suite presented in Section 4 of our paper. The beginning of Section 3 of this document has outlined those results.

Please note that running scripts in this section may take dozens of seconds to several minutes. The provided reference time is based on running the scripts on an Intel Core i5-9500 processor with 6 cores. Also note that all the scripts are located in the `/root/scripts` directory, and to facilitate their execution, `/root/scripts` has been added to the `PATH` environment variable. This means you can run the scripts without specifying the full path.

Reproducing Table 3 The feature distribution presented in Table 3 of our paper was obtained by traversing the abstract syntax tree (AST) of each test case file and checking whether it has tree nodes corresponding to those representative Verilog features. To reproduce this result, you can run the following script:

```
$ count-feature.sh
```

The script will generate the result in about **10 seconds**, which will be displayed as follows:

```
Feature V: 824
Feature N: 342
Feature CE: 730
Feature TC: 542
Feature BA: 653
Feature NBA: 83
Feature PCA: 44
Feature AFC: 24
Feature CA: 293
Feature SCH: 539
Feature CONN: 303
Feature LT: 105
Feature STF: 824
Feature GATE: 28
```

Each line corresponds to the count of different representative features, which can be compared to the one in Table 3 of our paper.

run *Icarus Verilog* In our paper, we described that *Icarus Verilog* (the compared popular simulator) actually encountered three failures out of the 824 cases in the Language Features Suite due to a known bug resulting from an incorrect implementation of procedural continuous assignment. To observe the results, you can run the following script, which takes approximately **40 seconds** to complete. Here are the last 5 lines of the script's output:

```
$ test-iverilog-lang.sh 2>/tmp/iverilog.err | tee /tmp/iverilog.out
...
wiresub1.v: PASSED
wirexor1.v: PASSED
xnor_test.v: PASSED
zero_repl.v: PASSED
Passed 821 / 824
```

The three failed cases are:

```
$ grep FAILED /tmp/iverilog.out
br605a.v: FAILED
br605b.v: FAILED
br971.v: FAILED
```

Those failures occurred due to a bug explained in `br971.v`:

```
$ head -n 2 $LANG_SUITE/br971.v
// Icarus doesn't properly support variable expressions on the right hand
// side of a procedural CA - see bug 605.
```

The mentioned bug report, named "bug 605", can be found at the following link: https://sourceforge.net/p/iverilog/bugs/605/. The content of the bug report is shown in Figure 3. The bug report highlights that *Icarus Verilog* implements procedural continuous assignment by evaluating the right-hand side expression only once at the time the assignment is executed. This implementation may provide intuitive behavior and allow *Icarus Verilog* to pass certain test cases discussed in Section 3.2.2. However, it comes at the cost of failing other test cases due to the limitations of this approach.

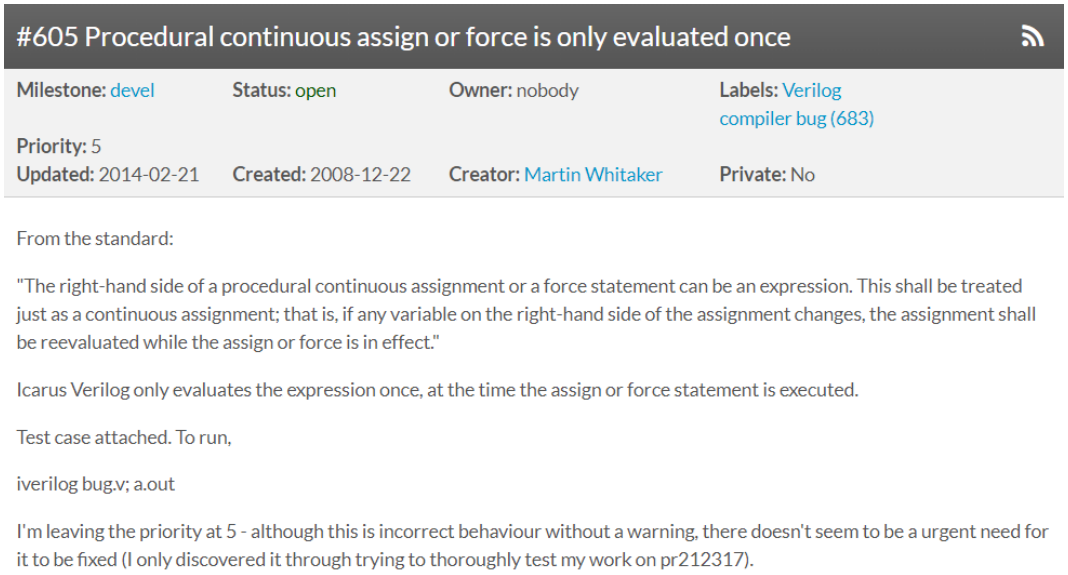

Figure 3: The bug report on procedural continuous assignments.

run *Verilator* *Verilator* fails to pass over 100 test cases in the Language Features Suite. To reproduce this result, run the following script, which takes about **five minutes**:

```
$ test-verilator-lang.sh
```

Note that the script runs much more slowly than *Icarus Verilog* because *Verilator* cannot automatically finish on some test cases. We need to wait for a large enough time (i.e., one second) for its total output and then kill it.

After waiting for a while, You can observe the following result (only show the last 7 lines):

```
...
wirexor1.v: PASSED
xnor_test.v: PASSED
zero_repl.v: PASSED
```

```
Passed: 609
Failed: 130
Compilation Error: 85
Total: 824
```

As shown, *Verilator* encounters compilation errors on 85 test cases and fails to pass 130 test cases. This is due to the lack of support for many Verilog behavioral features in *Verilator*.

run λ_V λ_V encounters failures in certain cases within the Language Features Suite, as summarized in Figure 1. Some failures are attributed to bugs in *Icarus Verilog* or ambiguities in the Verilog specification, as discussed in Section 3.1 and Section 3.2. Additionally, some failures are the result of either erroneous test cases or the stricter type checker employed in our implementation (described in our paper and also mentioned above), which we will discuss later.

To reproduce these failures, you can run the following script, which takes approximately **one minute** to complete by using **six threads**:

```
$ test-lv-lang.sh 2>/tmp/lv.err | tee /tmp/lv.out
```

Please note that this script will display some Gradle-specific information alongside the test results because it utilizes gradlew to execute the JUnit tests in our Java project, which in turn runs the test cases using the λ_V interpreter. In order to extract useful information from the output, you can use the following script:

```
$ process-lv-out.sh /tmp/lv.out
```

This script generates a summary of the results obtained by running λ_V on the Language Features Suite. For each failed test case, including those that threw exceptions, it provides relevant information. If a failure is due to known causes summarized in Figure 1, a brief description is printed. Otherwise, detailed error messages will be provided. It's important to note that because the results of some cases depend on scheduling, as we have explained in previous sections, the output may vary across different runs of the `test-lv-lang.sh` script:

```
Report 824 results collected from totally 824 test cases:
#Passed: 809
#Failed: 12
#Timeout: 0
#Unsupported: 0
#Exception caught: 3
#Assertion failed: 0

Following cases fail due to known reasons:
bufif.v: stricter-type-checker
ldelay1.v: ambiguity undefined-driver-output
casez3.10E.v: err-case
casez3.10D.v: err-case
br1000.v: err-case
sched2.v: bug hidden-data-race
br1001.v: stricter-type-checker
casez3.10C.v: err-case
gate_connect1.v: ambiguity unmatched-ports
```

```
    force_lval_part.v: ambiguity force
    pr2986528.v: bug hidden-data-race
    casex3.9E.v: err-case
    casex3.9D.v: err-case
    pr2943394.v: ambiguity force
    pr1662508.v: bug loss-of-newest-results

Gradle Test Executor 1 finished executing tests.
```

Because all failures have been meticulously accounted for, you can observe that all failures are explicitly linked to known reasons. You can also cross-reference the results with those listed in Figure 1 manually.

Next, we will further discuss the failures that are attributed to erroneous cases and our stricter type checker.

The first type of erroneous cases involves the misuse of the `!=` operator instead of the `!==` operator. These cases include the files `casex3.9D.v`, `casex3.9E.v`, `casez3.10E.v`, `casez3.10D.v`, and `casez3.10C.v`. The impact of this misuse is explained in Section 4.2 of our paper (lines 849-857). When *Icarus Verilog* executes these cases, the `result` variable retains its initial value `x` due to its schedule, allowing it to pass all the if statements because `x != anything` evaluates to `x` and the statement `if (x)` does not branch to the `then` block.

The second type of erroneous cases involves the sharing of a loop variable between concurrent always blocks. This is exemplified in the file `br1000.v`, where the always blocks in lines 10 and 17 share the loop variable `i`. This is incorrect because when one always block updates `i`, it can trigger the other always block to loop and modify `i`, resulting in a data race. *Icarus Verilog* can pass this case due to its non-preemptive scheduling.

During the writing of this document, we discovered a new type of erroneous cases caused by arithmetic overflow. This is exemplified in the case `sqrt32.v`, where 10001 32-bit unsigned integers are randomly generated to test whether a Verilog module can correctly calculate the square root of these integers. In this case, the generated unsigned integer is stored in the 32-bit reg `A` declared on line 108, and the computed square root is stored in a 16-bit reg `Z`. The case then assigns `A` and `Z` to `a` and `z` of `integer` type, and asserts the following condition on line 220 and 228:

```
z * z <= a < (z+1) * (z+1)
```

However, since `a` and `z` are of signed type, the expression `(z+1) * (z+1)` could result in an overflow and produce a negative number (e.g., `a=2147438958` and `z=46340`), causing the assertion to fail. Therefore, if the generated random number is sufficiently large, a correct simulator should fail to pass this case. *Icarus Verilog* can pass this case because it fixes the seed of its random number generator and generates 10001 numbers that do not lead to overflow, while λ_V generates numbers fully randomly. Hence, whether λ_V can pass this case depends on its random number generator. At the time of submitting our paper, we did not observe λ_V's failure on this case and therefore did not mention it. We will include this information in a later version of our paper.

Additionally, our stricter type checker leads to two failures in the cases `bufif.v` and `br1001.v`, as discussed in Section 4.2 of our paper (lines 858-863). In the `bufif.v` case, the wire `ad`, declared as an `output` in line 27, is used as an `input` in line 35. According to the Verilog standard (section 12.3.8), a port declared as `input` (or `output`) but used as an `output` (or `input`) may be coerced to `inout`, and a warning should be issued if not coerced. However, in our implementation, we reject this feature to avoid unexpected behavior.

In the `br1001.v` case, the wires `x` and `y`, of different types, are connected by the `submod` module in lines 10-11. According to the Verilog standard (section 12.3.9), if different net types (i.e., net kinds in λ_V) are connected through a module port, specific rules outlined in Table 12-1 should be applied to resolve the dissimilar port connection. However, in our implementation, as explained in our paper, we reject such programs during type checking to avoid unexpected behavior.

3.4 Results on Real-World Programs Suite

In this section, we provide instructions for verifying that λ_V successfully passes the Real-World Programs Suite. As detailed in our paper, this test suite encompasses a range of designs, including notable examples such as an rv32i CPU. These tests are sourced from various origins, with some originating from the open-source OpenPiton CPU project [2], while others are primarily sourced from Altera Corporation.

Due to the substantial computational demands posed by tests derived from OpenPiton, we will initially guide you through running the remaining tests.

The tests (excluding those from OpenPiton) are conveniently located within the directory specified by the environment variable `$RW_SUITE`:

```
$ ls $RW_SUITE/*.v
/root/qihe/lambdav/src/test/resources/testcases/verilog/realworld/crypto_des.v
/root/qihe/lambdav/src/test/resources/testcases/verilog/realworld/fp_sqrt.v
/root/qihe/lambdav/src/test/resources/testcases/verilog/realworld/lfsr.v
/root/qihe/lambdav/src/test/resources/testcases/verilog/realworld/priority_mux.v
/root/qihe/lambdav/src/test/resources/testcases/verilog/realworld/reg_cam.v
/root/qihe/lambdav/src/test/resources/testcases/verilog/realworld/rv32i.v
```

Each of these tests is encapsulated within a separate file, consisting of a design-under-test and a test bench responsible for providing inputs to the design and verifying the outputs.

To execute λ_V on these tests, run the following script, which typically takes about **5 minutes** (by using **6 threads**) to complete:

```
$ test-lv-rw-misc.sh 2>/tmp/lv.err | tee /tmp/lv.out
```

The output format is similar to running `test-lv-lang.sh`. Therefore, to obtain a clear output, use the following script, as we have used before:

```
$ process-lv-out.sh /tmp/lv.out
    Report 6 results collected from totally 6 test cases:
    #Passed: 6
    #Failed: 0
    #Timeout: 0
    #Unsupported: 0
    #Exception caught: 0
    #Assertion failed: 0

Gradle Test Executor 1 finished executing tests.
```

As shown above, all tests have passed.

For tests originating from OpenPiton, you'll find them housed within the `$RW_SUITE/openpiton` directory:

```
$ ls $RW_SUITE/openpiton/
counter_cases      ifu_esl_lfsr.v      lfsr_cases
ifu_esl_counter.v  ifu_esl_shiftreg.v  shiftreg_cases
```

In this directory, each test consists of a `.v` file containing both the design-under-test and the associated test bench, along with a corresponding `*_cases` directory that holds individual test cases. These cases include inputs and expected outputs used by the test bench. For example, the `ifu_esl_counter.v` file contains the design named "counter" to be tested as well as a test bench that reads test cases from the directory `counter_cases` to test "counter". You can use the following script to list all available designs-under-test and their test cases:

```
$ test-lv-rw-op.sh -l
Available (design, test-case) pairs:
  counter, clear_set
  counter, clear
  counter, pause
  counter, set
  counter, step_clear_set
  counter, step_clear
  counter, step_set
  counter, step
  lfsr, exhaust
  lfsr, ldstep
  lfsr, pause
  lfsr, seed
  lfsr, state0
  shiftreg, pause
  shiftreg, set_shift
  shiftreg, set
  shiftreg, shift
```

You can utilize the same script to run a single test case for a specific design-under-test using the following format:

```
$ test-lv-rw-op.sh <design> <test-case>
```

For example, if you intend to run the test with the largest test case from OpenPiton, as mentioned in our paper, which generates several megabytes of output over approximately 2,000,000 clock cycles, you can employ the following command, which completes in approximately **1.5 hours**:

```
$ test-lv-rw-op.sh counter step
Testing 'counter' by case 'step'

 Entering Test Suite: ifu_esl_counter
        [PASSED] Test (Timeout check) succeeded
Simulation -> PASS (HIT GOOD TRAP)
```

If the test passes, you will see "`HIT GOOD TRAP`" in the output. Note that tests from OpenPiton may occasionally encounter data races, leading to λ_V failing the test, which is the

same problem discussed in Section 3.1.2. In such cases, simply rerun the test as instructed above; typically, you will observe "`HIT GOOD TRAP`" after several attempts.

If you want to run all tests, you can execute the following script, which runs test cases for each design-under-test simultaneously:

```
$ test-lv-rw-op.sh --all
Testing 'shiftreg' by case 'pause'

 Entering Test Suite: ifu_esl_shiftreg
        [PASSED] Test (Timeout check) succeeded
Simulation -> PASS (HIT GOOD TRAP)
Testing 'shiftreg' by case 'set_shift'

 Entering Test Suite: ifu_esl_shiftreg
        [PASSED] Test (Timeout check) succeeded
Simulation -> PASS (HIT GOOD TRAP)

...

 Entering Test Suite: ifu_esl_counter
        [PASSED] Test (Timeout check) succeeded
Simulation -> PASS (HIT GOOD TRAP)
Testing 'counter' by case 'step'

 Entering Test Suite: ifu_esl_counter
        [PASSED] Test (Timeout check) succeeded
Simulation -> PASS (HIT GOOD TRAP)
```

Given the substantial input data provided by each test cases, ranging from several kilobytes to hundreds of kilobytes, completing all tests may require approximately **13.5 hours** to finish, despite our optimization efforts to run test cases for each design-under-test in parallel..

Note again that running some tests may not print "`HIT GOOD TRAP`" due to hidden data races. In such cases, the output contains information about each test's design and test case name, and you can rerun those tests using the command as instructed above to see that λ_V can indeed pass them:

```
$ test-lv-rw-op.sh <design> <test-case>
```

References

[1] Ieee standard for verilog hardware description language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pages 1–590, 2006.

[2] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. Openpiton: An open source manycore research framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 217–232, New York, NY, USA, 2016. ACM.