Enhancing Compiler-Driven HDL Design with Automatic Waveform Analysis

Frans Skarman†
Linköping University
Linköping, Sweden
frans.skarman@liu.se

Lucas Klemmer†
Johannes Kepler University
Linz, Austria
lucas.klemmer@jku.at

Oscar Gustafsson
Linköping University
Linköping, Sweden
oscar.gustafsson@liu.se

Daniel Große
Johannes Kepler University
Linz, Austria
daniel.grosse@jku.at

***Abstract*—The time-to-market of a new product is one of its most crucial factors for success, therefore, reducing this time is of utter importance. However, this reduction must not come at the expense of a less thorough development process.**

This paper presents a compiler-driven approach for automatically analyzing metrics such as transaction delays or bus throughput on simulation waveforms of projects developed in the Spade *Hardware Description Language* (HDL). By utilizing the Spade compiler's knowledge about design internals, an automatic analysis of the waveforms created during simulation is possible using the *Waveform Analysis Language* (WAL). Analysis programs can be bundled with Spade projects or libraries, such that they are automatically detected by Spade and can be reused by other projects using simple annotations. We call these bundled WAL programs *analysis passes*, since they fit into the Spade workflow and provide thorough analysis at no additional cost to the users of these libraries.

In a detailed description, we present how new analysis passes can be defined using the example of a data streaming interface. Additionally, we highlight the possibilities of analysis passes in two case studies, including *Finite State Machine* (FSM) and Wishbone protocol analysis.

***Index Terms*—Performance Analysis, Hardware Description Languages, Debugging**

I. Introduction

One of the most critical aspects for the success of a new product is the time it takes from the conception of the idea to the time the product is available on the market. When it comes to shortening this *time-to-market* no other industry has been as successful as the software industry. This agility we see in the software domain today is to a large extent enabled by continuous improvements to tools and workflows.

Compared to software tooling, the hardware domain saw relatively little advancements in this area until the emergence of the open-source *Electronic Design Automation* (EDA) community, with tools such as Yosys [1], [2] or the Chisel *Hardware Description Language* (HDL) [3]. This is, in part, due to the naturally much lower abstraction level of hardware design. *Register Transfer Level* (RTL) languages, such as Verilog and VHDL, typically express very little high-level design intent to simulators, synthesis flows, and other tools. The description consists of instantiations of individual components that manipulate individual signals, but properties of those signals and instances, such as which signals make up a bus, or which registers correspond to a pipeline versus which are used for state machines, is not expressed in the language. This makes it difficult to build re-usable tools for analysis and debugging, which in turn means that debugging often has to be done through manual waveform inspection, a tedious process which has to be re-done for every design change.

Waveform Analysis Language (WAL) [4] is a language which allows writing programs to perform automatic waveform analysis. For example, it can be used to analyze bus traffic, analyze the transitions of state machines, or measure the performance of processors [5] [6]. However, because RTL languages allow great flexibility in expressing designs, there is relatively little, if any, shared structure between different designs. Therefore, these analysis programs have often to be written on a per-design basis.

Spade [7] is a statically typed HDL with explicit constructs for registers, pipelines, and memories. In addition to boosting productivity by adding more abstraction, Spade exposes more high-level design intent to the compiler. Registers comprising a state machine become explicit, pipelines being built into the language makes the compiler aware of which registers are used for pipelining, and which signals logically belong together. Finally, via Spade's strict type system for defining buses and other related signal groups, such as ready-valid interfaces, information about which analysis is applicable to which signals is readily available to WAL.

By bundling WAL analysis programs with Spade projects, they can query the Spade compiler for additional information about the design and they become automatically discoverable by the Spade toolchain. We call WAL programs bundled with Spade projects *analysis passes* as they can be automatically run as an additional step in the build and simulation flow.

The tight integration of Spade and WAL allows users to directly take advantage of these analysis passes simply by annotating the signals or structures they want the analysis to be performed on. Thanks to the knowledge the compiler has, these annotations are all that is required to connect the design with the analysis. Now, compiling, simulating, and analyzing is handled automatically by the Spade build system. In addition, this not only works for a single Spade project, but analysis passes can seamlessly be re-used across the Spade ecosystem. This allows library (IP) authors to provide not only

† Authors contributed equally to this work.

979-8-3503-0737-5/23/$31.00

```
[libraries.wishbone]
git = "https://gitlab.com/spade-lang/lib/fishbone"
branch = "main"
[plugins.wal_analysis]
git = "https://gitlab.com/spade-lang/wal_analysis"
branch = "main"
```

Listing 1: The Wishbone implementation and WAL analysis plugin can be included as a dependency in the swim project configuration file.

an implementation but also analysis passes, thus giving their users a head start on ensuring design quality.

The main contributions of this work are twofold. The first and primary contribution is the integration of WAL and Spade specifically. This includes augmenting the Spade compiler and language with additional annotations to mark types and values for analysis as well as adding several functions and macros to WAL which make it easier to write Spade specific analysis passes. In addition, we developed a plugin for the Spade build system to bundle WAL analysis passes with Spade libraries and to detect and run those automatically. This plugin is available to Spade users on GitLab.[1] The code for the example analysis passes is also available on GitLab.[2] The changes to the Spade compiler have been included in its repository.

The second major contribution of this work is the methodology used to perform the integration. Its core is to bind analysis programs to designs using compiler-generated signals leveraging a strong type system. This methodology can be used to integrate WAL or similar tools with other modern HDLs.

The rest of the paper is structured as follows. First, an example showcasing the power of the proposed automated waveform analysis is given in Section II. Then, in Section III, Spade and WAL are introduced. In Section IV, a more thorough example of how analysis passes are used and created is given. Section V presents the built-in state machine analysis pass. Finally, related work is discussed in Section VII, and the paper is concluded in Section VIII.

II. Motivating Example

As a motivating example, we will study a sample design consisting of two Wishbone masters, which communicate with a shared slave via an arbiter. Wishbone [8] is a widely used on-chip communication protocol with broad adoption especially in the open-source hardware community. It utilizes handshaking-based communication and allows the implementation of various network topologies such as buses or point-to-point communication.

A Spade implementation of the Wishbone bus is available in a git repository,[3] which can be added as a dependency to a Spade project via a configuration file for the build system, Swim. This is shown in Listing 7.

The top level of our motivating example is shown in Listing 2. A more thorough description of the Spade language is given in Section III-A, but for this example, the most interesting statements are on Line 5 and Line 7, which define

```
1  entity wb_harness(clk: clock, rst: bool)
2    -> (int<16>, int<16>)
3  {
4      #[wal_trace(target=wb1, clk=clk, rst=rst))]
5      let (wb1, wb1inv) = port;
6      #[wal_trace(target=wb2, clk=clk, rst=rst))]
7      let (wb2, wb2inv) = port;
8
9      let _ = inst wishbone_arbiter(
10           clk, rst, wb1inv, wb2inv, ...
11         );
12     let o1 = inst wishbone_master(
13           clk, rst, wb1, ...
14         );
15     let o2 = inst wishbone_master(
16           clk, rst, wb2, ...
17         );
18
19     (o1, o2)
20 }
```

Listing 2: Spade description of two wishbone masters communicating via a shared arbiter. The novel `#[wal_trace(...)]` annotations enable the analysis for an interface

two wishbone buses, `wb1` and `wb2`. On Lines 9–17, the ports are passed to the arbiter and the two masters. The arbiter is configured to reply to port 1 with a latency of 3 cycles, and port 2 with a latency of 6 cycles (not shown in Listing 2). The masters perform single random reads or bursts of writes, with write bursts being more common and reads happening randomly between the write bursts.

To meet the systems requirements, passing functional tests is not enough. Let us look at the bus example: Given a workload, i.e. simulation trace, various metrics such as the delay and number of read/write transactions, average read/write latency, number of idle cycles etc. have to be determined. Usually, the engineer starts using a waveform viewer using features like markers to determine the initial results based on fully manual inspection. However, this quickly becomes unfeasible for tasks such as finding the average delay over the complete simulation trace. Therefore, later in the design process, custom monitoring logic is added to the design or the testbench is extended with analysis logic. Both approaches have their drawbacks as they require lots of additional work by users of the Wishbone library, either by invasive changes to the design logic for debugging purposes (which should be avoided), or by writing custom testbenches which are hardly re-usable across projects.

By tightly coupling Spade and WAL, the users of the Wishbone library who wish to determine the metrics mentioned above only have to annotate the Spade code with a single additional line per Wishbone instance as shown on Line 4 and Line 6 of Listing 2. Now, by simply running `swim plugin analysis`, the design is compiled, simulated and the WAL analysis programs are executed automatically for every simulation trace. Listing 3 shows the output of the Wishbone analysis pass which collects performance metrics. The output includes information about the number of reading and writing transactions, average read and write delays, the number of error responses by the slave, and the number of cycles in which the interface is idle. These metrics are analyzed and

[1] https://gitlab.com/spade-lang/wal_analysis
[2] https://gitlab.com/TheZoq2/remora-code
[3] https://gitlab.com/spade-lang/lib/fishbone

```
[ANALYSIS] # wb_harness.wb1
[ANALYSIS]    Nr. transactions : 697
[ANALYSIS]    Nr. reads        : 16
[ANALYSIS]    Avg read latency : 6 clock cycles
[ANALYSIS]    Nr. writes       : 681
[ANALYSIS]    Avg write latency: 9 clock cycles
[ANALYSIS]    Inactive cycles  : 6%
[ANALYSIS] # wb_harness.wb2
[ANALYSIS]    Nr. transactions : 421
[ANALYSIS]    Nr. reads        : 0
[ANALYSIS]    Avg read latency : - clock cycles
[ANALYSIS]    Nr. writes       : 421
[ANALYSIS]    Avg write latency: 12 clock cycles
[ANALYSIS]    Inactive cycles  : 22%
```

Listing 3: Analysis results of the Wishbone analysis pass.

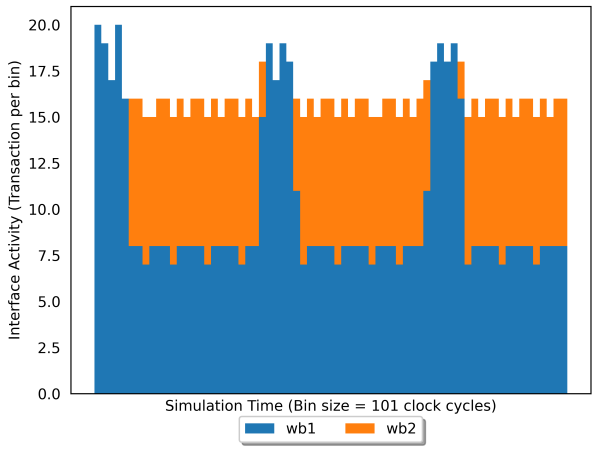

Fig. 1: Histogram of transactions on Wishbone interfaces. The y-axis value is the number of transactions in a given time slice of the simulation (bin).

presented for every annotated Wishbone interface, in the case of this example the two interfaces `wb_harness.wb1` (`wb1`) and `wb_harness.wb2` (`wb2`).

Additionally, the Wishbone analysis pass also generates a histogram containing the transactions of every traced Wishbone interface. The generated histogram displays the activity of all interfaces over the complete simulation time using a stacked bar-chart. Each colored bar of the histogram represents the number of transactions on a given Wishbone bus in a slice of the simulation time. Fig. 1 shows the activity of the two Wishbone interfaces. From the chart we can see that the two traced interfaces show different behaviors. `wb1` (blue) is sending a constant stream of data while `wb2` (orange) is sending data in larger chunks. Interestingly, the number of transactions by `wb1` spikes three times. This always happens when `wb2` is not accessing the bus, thus leaving more bandwidth to `wb1`. Another notable observation is that the total number of completed transactions is higher when only `wb1` is used, compared to when both `wb1` and `wb2` are active. This is because the requests from `wb2` take longer to finish, 6 cycles compared to 3.

From the perspective of library maintainers providing analysis programs is as simple as placing them in an analysis directory in their libraries. Then, these programs are automatically detected by Swim and by this become analysis passes.

The key takeaway from this example is that by bundling analysis passes with libraries, and having convenient annotations for opting into those analysis passes, one can, almost without effort, get a large amount of information about the dynamic behavior of the system. Information that would be very difficult to find simply by looking at wave forms in a traditional waveform viewer, or very cumbersome to generate in traditional testbenches.

III. Languages

In this section, we introduce the two languages that are central to this paper: Spade and WAL.

A. *Spade*

Spade [7] is a statically typed open source HDL which adds abstraction without sacrificing low level control over the generated hardware. These abstractions include language level support for pipelines, explicit constructs for registers and memories, as well as support for ports with linear type checking to ensure correctness.

In addition, Spade integrates several ideas found in modern software languages. Rather than being imperative, it is expression based, meaning that the result of a conditional, like an if expression, is assigned to a variable, instead of the variable being set in each branch. The language has type inference, meaning that types usually do not have to be explicitly spelled out, the compiler will infer them and report errors if it detects inconsistencies.

For a full description of the Spade language, see [9]. However, for the purpose of this paper, it is enough to understand the code shown in Listing 4, which will also be used as a running example throughout the rest of the paper. At a high level, this defines a compute-unit which takes a stream of commands and processes them one at a time. There are two types of commands: first the mode selectors, `Mult` and `Add`, and secondly `Data` which contains two values to be multiplied or added depending on the previous mode command. The enum type containing these commands is defined in Lines 1–5. The input and outputs to the module are streamed with a data valid signal. Line 7 defines this stream type as a struct that is generic over type T and contains the valid bit as well as data of type T. The last type for this example is defined on Line 9. It specifies the internal states that the unit can have: `Add` and `Mult`.

The implementation of the unit starts in Line 11, beginning with its name and specifying that it is a pipeline of depth 3, i.e. a pipeline that has 3 registers between input and output. The next four lines specify the inputs and outputs, apart from a clock and a reset, the unit takes a stream of commands, and produces a stream of computed values. Lines 16–22 define a register containing the current state of the unit. The register is called `state`, is clocked by `clk` and is reset back to the `Mult` state by the `rst` signal. The next state is given by the match expression in Lines 17–22, which retains the current state if

```
1  enum Cmd {
2    Data{l: int<16>, r: int<16>}
3    Mult,
4    Add,
5  }
6
7  struct Stream<T> { valid: bool, data: T }
8
9  enum State { Add, Mult }
10
11 pipeline(3) main(
12   clk: clock,
13   rst: bool,
14   cmd: Stream<Cmd>
15 ) -> Stream<int<16>> {
16     reg(clk) state reset(rst: State::Mult()) =
17       match cmd {
18         Stream(true, Cmd::Mult) => State::Mult(),
19         Stream(true, Cmd::Add) => State::Add(),
20         Stream(true, Cmd::Data(_, _)) => state,
21         Stream(false, _) => state
22       };
23
24     let (out_valid, l, r) = match cmd {
25       Stream(true, Cmd::Data(l ,r)) =>
26         (true, l, r),
27       _ =>
28         (false, 0, 0)
29     };
30
31     let sum = trunc(l+r);
32     let prod = trunc(l*r);
33   reg * 3;
34     let result = match state {
35       State::Mult => Stream(out_valid, prod),
36       State::Add => Stream(out_valid, sum)
37     }
38     result
39 }
```

Listing 4: A Spade unit processing a stream of commands into a stream of integers.

the command is not valid, or if it is data to be processed. Otherwise, the state is set to perform addition or multiplication depending on the command.

Lines 24–32 extract the left- and right-hand operands of data commands if present before performing both addition and multiplication. The `trunc` function truncates a variable to the bit-width of the target variable. In the case of the truncations on Lines 31–32 the correct bit-width is inferred by the Spade compiler from the type parameter `T` of the `Stream` result of the pipeline. Line 33 specifies that before doing anything else, the compiler should insert three pipeline registers for all the signals in the design. Using three registers allows the synthesis tool to efficiently map the multiplication into a DSP-block (the dedicated hardware for, among other things, multiplication) in an FPGA. Finally, in Lines 34–37, the output is selected depending on the state, validity of the result and the sum and product. The pipeline construct ensures that the `state` variable here refers to the state of the unit at the time when the computation was started.

To highlight how the pipelines work, Fig. 2 contains an example trace of the unit performing one product, then switching the mode to addition to compute a single sum before returning to multiplication mode. In the interest of space, the pipelined

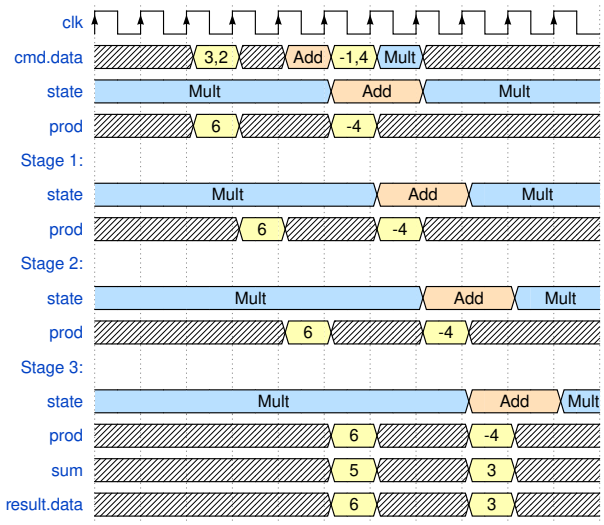

Fig. 2: Waveform of the pipeline defined in Listing 4 processing a short data stream.

copies of the `sum` variable have been omitted.

Spade also comes with a build system called Swim[4]. Among other things, this build system runs synthesis and simulation tools, manages dependencies and supports extensions through plugins. The plugin and dependency system is of particular interest in this paper, as it allows developers of Spade libraries to bundle WAL analysis passes with their libraries. Additionally, it allows the build system to automatically detect and run these passes when requested.

B. Waveform Analysis Language (WAL)

WAL [4] is a programming language created specifically for debugging and analyzing waveforms. The main idea behind WAL is that getting the value of a signal from a waveform should be as simple as getting the value of a variable in any other programming language and, that other concepts central to hardware design such as time and design hierarchy are integral parts of the language. Thus, accessing signals in WAL is similar to accessing variables, with the difference that the returned value depends on the loaded waveform and the time at which the signal is accessed.

WAL also allows writing generic programs that can exploit recurring structures present in nearly every design (e.g. standard buses or state machines).

WAL's syntax is based on Symbolic expressions [10] (S-expressions for short) which are common in languages related to Lisp, such as Common Lisp or Scheme. S-expressions can be either atoms or lists. Atoms are literals like numerical or string values, e.g. `1`, `0xff`, `"text"`, or symbols. Lists are multiple S-expressions separated by white space and enclosed in parentheses `(expr1 expr2 ...)`. All operators and function calls are written in prefix notation, e.g., `(+ 3 b)` to compute the sum of `3` and `b`.

WAL extends standard S-expressions to accommodate typical hardware development tasks into the language. First and

[4]https://gitlab.com/spade-lang/swim

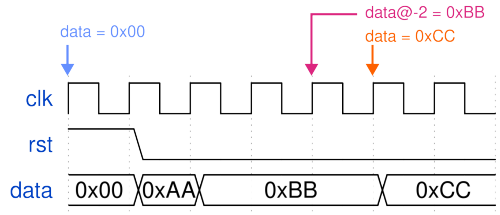

Fig. 3: An annotated waveform showing the WAL programming principle.

foremost, free symbols (i.e. symbols to which no values have been assigned) are interpreted as signals in the loaded waveform. This means, that if a free symbol is evaluated (i.e. its value is computed) the value is looked-up in the loaded waveform. Since looking up the value in a waveform depends on time, WAL keeps track of a pointer into the waveform which is called `INDEX`. The index can be moved forwards or backwards using the `(step offset)` function or, for a specific expression only, using the `expression@offset` syntax.

To show the programming principle of WAL consider the waveform in Fig. 3. After starting WAL and loading this waveform the `INDEX` points to the start of the waveform (this is indicated by the blue arrow). If we now evaluate the expression `data` we get the value `0x00`. Evaluating `(step 10)` moves the index forwards by ten timestamps (indicated by the orange arrow). Note, that the index is not incremented for each rising edge of the `clk` signal but whenever **any** signal is changed. Now, evaluating the same `data` expression results in the value `0xCC`. Finally, we can move the index locally for just one expression using the `expression@offset` syntax. Using this syntax the `expression` is evaluated at `INDEX + offset` and after the evaluation the index is restored to its previous value. Thus, evaluating `data@-2` at `INDEX = 10` results in the value `0xBB` (indicated by the magenta arrow).

Additionally, WAL allows writing generic analysis programs by decoupling signals from their location inside a design. This works by specifying only the local name of a signal and filling in the full path only later during program runtime. For example, the expression `(in-scope 'tb.dut (&& ~clk (! ~rst)))` evaluates the expression `(&& ~clk (! ~rst))` in the scope `tb.dut` and thus the signal `~clk` expands to `tb.dut.clk`.

WAL's language features for timing and writing generic programs allow expressing hardware analysis problems in a natural and efficient way. However, to make expressing these problems even simpler, WAL also contains a range of higher-level functions that are often required while analyzing waveforms. For example, the `whenever` function can be understood as a `while` loop on wave forms, since it evaluates the expressions in its body at every timestamp at which the condition evaluates to true. Further, there are functions to `find` time stamps at which an expression evaluates to true, to change when signal values are sampled, to get information about signals or scopes, and more.

IV. Using and Defining Analysis Passes

In this section we will show how the automatic waveform analysis is integrated into a Spade program. We will use the running example from Listing 4 and show how it is augmented to both define and use analysis passes.

```
7  #[wal_traceable(uses_clk, uses_rst)]
8  struct port Stream<T> { valid: bool, data: T } {

12 pipeline(3) main(
13   clk: clock,
14   rst: bool,
15   #[wal_trace(clk=clk, rst=rst)]
16   cmd: Stream<Cmd>
17 ) -> Stream<int<16>> {

34 #[wal_trace(clk=clk, rst=rst)]
35 let result = match state {
36   State::Mult => Stream(out_valid, prod),
37   State::Add => Stream(out_valid, sum)
38 }
```

Listing 5: Tracing added to the streams from Listing 4

The primary candidates for analysis in the running example are the two streams of data going into and out of the unit. Those can be analyzed by applying the `wal_trace` attribute to the input stream in Line 14, and to the output stream in Line 34 as shown in Listing 5.

However, because the `Stream` type is defined in the project, not fetched as a library which already includes analysis passes, we need to also define how streams should be analyzed. For most passes, this is done in three steps: annotating the struct to inform the compiler that it is an analyzable struct, writing the WAL code for performing the analysis, and, optionally, writing a Python wrapper for additional Python integration.

A. Struct Annotation

The first step in enabling automatic analysis is annotating the structs which can be analyzed with the `#[wal_traceable]` attribute as shown in Line 7 of Listing 5. This annotation communicates that an analysis pass is available and allows specifying if clock and reset signals need to be provided in addition to the struct signals. With the `wal_trace` and `wal_suffix` signals in place, the Spade compiler will generate `X__valid__proj::main::Stream` and `X__data__proj::main::Stream` signals as well as corresponding signals for the clock and reset signals for each instantiation of the `Stream` struct. For each instance of the stream interface `X` is replaced by the hierarchical name of the variable defining the stream. The path `proj::main::Stream` is the fully qualified path of the struct in a Spade project, and depends on the name of the project and which file the struct is defined in. It is unique and unchanged when a library is added as a dependency to another project.

B. WAL Code

The WAL analysis program for the stream type is defined in Listing 6. This program uses WAL's grouping feature together with the Spade integration to automatically run the `stream-utilization` function for every detected stream instance. Lines 1–3 use the `spade-struct` macro create a list of all groups of signals which are generated by the `wal_trace` attribute. This list is then passed to the `in-spade-struct` macro in Line 5,

```
1  (define stream-instances
2    (spade-struct proj::main::Stream
3                  [clk rst valid data]))
4
5  (in-spade-structs
6      proj::main::Stream stream-instances
7    (let [(send 0) (idle 0)]
8      (whenever (&& (rising #clk) (= 0 #rst))
9        (if #valid
10         (inc send)
11         (inc idle)))
12     (define time-sending
13       (round (* (/ send (+ send idle)) 100)))
14     (log/analysis CG ": " time-sending "%")))
```

Listing 6: The WAL implementation of the stream analysis pass in file "utilization.wal".

```
1  class StreamUtilizationPass(WalAnalysisPass):
2    def __init__(self, pass_dir, wavefile):
3      super().__init__(pass_dir, wavefile)
4
5    def run(self):
6      self.wal.eval_str('(require utilization)')
```

Listing 7: The Python implementation of the stream analysis pass in file "utilization.py".

which performs analysis on each of the stream interfaces. The `in-spade-struct` macro replaces `#field` with the signal corresponding to that field, for example, `#valid` is expanded to `X__valid__proj::main::Stream` where X is the name of the struct instance being analyzed. One metric that is of interest to users of the `stream` interface is the utilization of an instance of this interface, i.e. the percentage of time at which valid data is transmitted. This is measured in Lines 8–14. First, the `whenever` function is used to visit every time step at which the clock rises and the reset is low (Line 8). These are the time steps at which potential data transactions can occur. Next, depending on the value of the `#valid` signal, the `send` or `idle` variables are incremented in Line 9–11. Finally, the fraction of time the interface is sending is calculated in Line 13. This result is then printed together with the name of the interface (stored inside the special variable `CG` which stores the Current Group) (Line 14).

C. Making the Pass Discoverable

To make an analysis pass discoverable by the swim build tool, it has to be placed inside the Spade project at a predefined location. All analysis passes must be placed inside a *wal* directory in the root of the Spade project.

There are two ways to make an analysis pass discoverable by the swim build tool: 1) the pass can be implemented using plain WAL files, providing a `run-pass` function or macro definition and 2) the pass can be implemented using WAL files and Python wrappers. If one of the two options is implemented, the swim analysis plugin automatically discovers the pass and runs it on every produced wave file.

The analysis pass presented in this example uses the second option of using a Python wrapper class. Therefore, the Spade project contains the two files "utilization.py", which contains the Python wrapper , and "utilization.wal", which contains

```
[INFO] Running Stream Utilization Pass on tb.vcd
[ANALYSIS] proj::main::main.cmd_n1: 80%
[ANALYSIS] proj::main::main.result: 26%
```

Listing 8: Example output of the analysis.

the WAL program presented that Section IV-B.[5] A Python wrapper must be a class that inherits from `WalAnalysisPass`, which in turn initializes the WAL interpreter, sets up the Spade compiler integration, and provides new WAL functions, for example for translating Spade values. The constructor for the `WalAnalysisPass` class takes two arguments, the directory of the pass (i.e., the path to the current Spade project) and the waveform that will be analyzed by this pass. WAL analysis passes without a Python wrapper are wrapped automatically so that they have access to the same Spade integration.

Lines 2–3 contain the constructor of the class which in this case only initializes base class. The `run` function is defined in Line 5. As most of the logic for this pass is implemented in the WAL program, all the `run` function does is requiring the *utilization.wal* file in Line 6 which evaluates all the expressions it contains. This is done by evaluating the `(require utilization )` expression using the `eval_str` function of the WAL object, which was already created by the base class.

To run all analysis passes, the swim build tool is invoked as `swim plugin analysis`, which in this case produces the output shown in Listing 8 when run on a stream of 1000 random commands with a four in five chance of generating a valid command. As one might expect, the percentage of valid commands is around 80%, while the output, having to switch between modes on roughly one in three valid commands, contains valid data at around 26% clock cycles. This gives a designer a valuable insight: re-ordering computations to avoid mode switches would improve the throughput of the system.

Finally, Listing 9 defines an analysis pass without a Python wrapper. The pass prints the operands and results of all valid multiplications, along with the time at which they were performed. The first two lines define the pass and specify the scope of the following signals, to avoid having to type out the name of the top module several times. Lines 3–8 filter out all time stamps under which the `prod` variable has valid data. Line 7 calls the `spade/translate` function to query the compiler for the Spade representation of the current value of the `state` variable. Listing 10 shows the result of this pass on a short sequence of inputs.

V. Finite State Machine Analysis Pass

The analysis plugin contains a set of pre-defined analysis passes, one of them providing *Finite State Machine* (FSM) tracing. This pass analyzes the state distribution and state transitions of state machines. All users have to do to analyze a state machine is to add an `#[fsm]` annotation to the signal containing the state as shown in Listing 11. The FSM analysis pass produces two outputs for each annotated state machine.

[5]The file names of analysis passes are irrelevant for auto-detection by the swim analysis plugin.

```
(defun run-pass []
  (in-scope 'proj::main::main
    (whenever
        (&& (rising ~clk)
            (= ~rst 0)
            (= "Mult"
               (spade/translate "state" ~state))
            ~out_valid)
      (log INDEX ":" ~l "*" ~r "=" ~prod))))
```

Listing 9: A simple WAL analysis pass which prints valid values of the `product` variable in Listing 4.

```
[INFO] 2 : 10241 * 38434 = 58914
[INFO] 6 : 16111 * 1527 = 25497
[INFO] 20 : 63773 * 46328 = 47128
[INFO] 24 : 56822 * 10195 = 27586
```

Listing 10: Sample output of the pass defined in Listing 9.

First, it produces a *Control Flow Graph* (CFG) like the graph shown in Fig. 4. This CFG shows every visited state together with further analysis results about this state. These analysis results include the percentage of time the FSM was in a given state, the average time the FSM spends in a state, and the number of times each state transition was taken. For example, Fig. 4 shows the CFG of *wb1* from the previously presented Wishbone masters that were analyzed on the same simulation results as the analysis passes in Section II. Of the three states, the *Write* state was active the longest time, and most transitions happened between this and the *Wait* state. Additionally, the 16 transitions between the *Wait* and *Read* states reflect the 16 reads we observed in Section II.

Secondly, all state transitions are rendered into sequentially numbered images. A selection of these images is shown in Fig. 5 (read from left to right and from top to bottom). These images contain the full control flow graph, in which the current state and the transition that led to this state are colored in red. Additionally, the current state register value is translated using the Spade compiler integration into a string representation of the state name, including its name and current values. This state transition animation allows stepping through each transition, closely following the execution of the FSM.

The FSM analysis is an example of an analysis pass that uses external software packages. Since WAL is fully interoperable with Python, analysis pass authors can use the extensive Python ecosystem to report analysis results in a fitting format, or to integrate the analysis into other existing workflows.

VI. Implementation

The implementation of the WAL integration into Spade is primarily based on the generation of groups of signals with a known suffix, which WAL analysis passes look for. The use of these suffixes allows the WAL passes to be written without much involvement of the compiler. In particular, the suffixes communicate clearly which signals should be traced and which should not, and exposes individual fields of structs without requiring knowledge of the struct packing. The compiler simply emits Verilog signals with these known suffixes for signals, or parts of signals that should be traced. The compiler exposes a Python API for cases where WAL needs information

```
#[fsm]
reg(clk) state reset(rst: State::Mult()) =
    match cmd { ... }
```

Listing 11: Annotating the FSM from the running example to opt into FSM analysis.

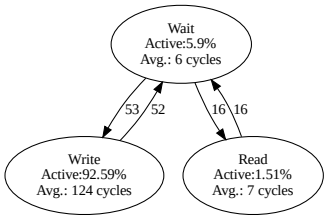

Fig. 4: State distribution and state transitions.

from the compiler, such as to translate a value from its bit representation to a human-readable Spade value.

The Spade compiler is a multi-stage compiler with an architecture as shown in Fig. 6. The compilation process starts with lexing and parsing to generate an *Abstract Syntax Tree* (AST). This AST is lowered into a *High-level Intermediate Representation* (HIR), a process which retains the tree structure of the AST but resolves names and scoping rules, and performs initial semantic analysis. On the HIR, type checking and some transformation passes are performed, and the HIR along with the type information is used to generate a *Mid-level Intermediate Representation* (MIR). In this step, more semantic analysis is performed, and the tree structure is flattened to a list of simple statements.

Attributes are part of the AST, and are baked into the HIR nodes during AST lowering. The WAL related attributes are type checked, and then lowered into dedicated MIR statements during HIR lowering. Finally, those statements are lowered to standard MIR statements which alias the required signals or sub-signals. This has to be done so late in the process since the final names and types of variables and instances is not decided until the MIR has been generated.

It is also worth discussing the benefits of integrating WAL with Spade compared to directly using WAL. WAL's primary method of discovering all signals for analysis is, as discussed, via signals with dedicated suffixes. Without integration, these signals must be defined manually. For example, the tracing of the `result` struct in our running example would look like Listing 12, as compared to the last block of Listing 5. Of course, the same signals would also need to be defined for the `cmd` bus. In addition to being much more tedious to write, this also requires the user to update all the extra signals whenever the fields of a struct are updated. This is extra problematic as the struct being traced in this case is defined in an external library. In the proposed implementation, the compiler also emits an error when the user attempts to trace a struct which is not marked as `wal_traceable`. This ensures that only structs with associated analysis passes are traced, another guarantee which is lost without integration. Finally, without integration with the build tool, it would not be as easy to bundle WAL passes with libraries in a structured way and have those analysis passes run automatically.

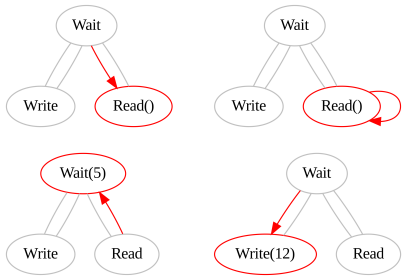

Fig. 5: Frames from the generated FSM state transition animation.

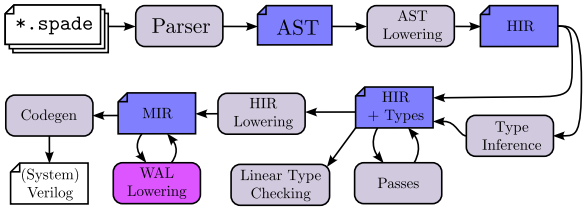

Fig. 6: Architecture of the Spade compiler.

VII. Related Work

There are several modern HDLs available. Perhaps the most well known is Chisel [3], a hardware construction language where hardware description takes the form of Scala programs which instantiate hardware components. Several other hardware construction languages exist, for example, SpinalHDL [11], also embedded in Scala and Amaranth [12] which is embedded in Python. Other notable projects include Clash [13], a compiler from Haskell to hardware, Pipeline-C [14], a HDL heavily inspired by C, and Silice [15], a standalone HDL primarily focused on describing algorithms. To the best of the authors' knowledge however, none of these projects integrate the automated waveform analysis described in this paper.

In [16], the authors explore FIRRTL [17], the Flexible Internal Representation for RTL, as a basis for hardware libraries. They also argue that traditional design languages have been slow to adopting abstraction and modularity and that these features can significantly improve the design speed and reusability. FIRRTL also provides design analysis options such as design coverage instrumentation or logic optimization. However, simulation analysis is not the main focus of FIRRTL and the coverage analysis relies on code transformations that inject additional logic into design. Compared to [16], our approach also injects additional signals into the design, however, these only provide the bridge to a fully-fledged programming analysis that provides much more analysis capabilities. In addition, our approach emphasizes that new analysis passes are provided by library authors, and that this is possible with as little work as possible, often with just one automatically detected file inside the library. Overall, since FIRRTL also encodes the high-level design intent similar to Spade, integrating WAL analysis passes into FIRRTL should be no problem.

Yosys [1] can be used to extract FSMs from a flattened netlist after synthesis. However, this approach only provides a static view on an FSM, so state distribution and state transitions as well as frames for FSM transitions wrt. simulation scenarios cannot be determined.

```
33  let result = match state {
34    ...
38  };
39  let result__valid__stream = result.valid;
40  let result__data__stream = result.data;
41  let result__clk__stream = result.clk;
42  let result__rst__stream = result.rst;
```

Listing 12: Tracing added to the streams from Listing 4

VIII. Conclusion

By integrating the WAL waveform analysis language with the Spade hardware description language, developers can easily get more information out of their simulation wave forms. A few examples of this include finding the performance characteristics of a wishbone bus, and the run time behavior of finite state machines. The integration of WAL in the Spade ecosystem allows library authors to define custom analysis programs for the types and designs they provide, and for users to take advantage of these programs by simply annotating the signals they would like to analyze.

Acknowledgments

This work has partially been supported by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.
[2] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: An open source framework from Verilog to bitstream for commercial FPGAs," 2019, pp. 1–4.
[3] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *DAC*, 2012, pp. 1212–1221.
[4] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASPDAC*, 2022, pp. 358–364.
[5] ——, "Waveform-based performance analysis of RISC-V processors: late breaking results," in *DAC*, 2022, pp. 1404–1405.
[6] L. Klemmer, E. Jentzsch, and D. Große, "Programmable analysis of RISC-V processor simulations using WAL," in *DVCON Europe*, 2022.
[7] F. Skarman, G. Sörnäs, and O. Gustafsson, "Spade," Apr. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.7729341
[8] OpenCores, "Wishbone B4," Tech. Rep., 2010. [Online]. Available: https://cdn.opencores.org/downloads/wbspec_b4.pdf
[9] F. Skarman and O. Gustafsson, "Spade: An expression-based HDL with pipelines," in *3rd Workshop on Open-Source Design Automation (OSDA)*, 2023.
[10] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Commun. ACM*, vol. 3, no. 4, pp. 184–195, Apr. 1960.
[11] SpinalHDL contributors, "SpinalHDL," https://github.com/SpinalHDL/SpinalHDL, 2022.
[12] Amaranth contributors, "Amaranth HDL," https://github.com/amaranth-lang/amaranth, 2022.
[13] C. Baaij, "Digital circuits in cλaSH," PhD. Thesis, University of Twente, Jan. 2015.
[14] J. Kemmerer, "PipelineC," Nov. 2022. [Online]. Available: https://github.com/JulianKemmerer/PipelineC/tree/ab87bb0b
[15] S. Lefebvre, "Silice," https://github.com/sylefeb/Silice/tree/5003ec72, Nov. 2022.
[16] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD*, 2017, pp. 209–216.
[17] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html