WAVING Goodbye to Manual Waveform Analysis in HDL Design with WAL

Lucas Klemmer, *Student Member, IEEE,* Daniel Große, *Senior Member, IEEE*

***Abstract*—Starting points for design understanding and debugging of a *Hardware Description Language* (HDL) design are generated waveforms. However, waveform viewing is still a highly manual and tedious process, and unfortunately, there has been no progress for automating the analysis of waveforms. Therefore, we introduce the *Waveform Analysis Language* (WAL) in this paper. WAL allows to create and execute analysis programs on waveforms. We have realized WAL as a *Domain Specific Language* (DSL). This design choice has many advantages ranging from a natural expressiveness of a waveform analysis problem to providing an *Intermediate Representation* (IR) well-suited as a compilation target from other languages.**

We demonstrate the capabilities of WAL in four case studies, covering the analysis of hardware performance of different RISC-V processors, combined hardware/software profiling, the usage of WAL to analyze bus transactions, and the implementation of a new embedded DSL using WALs macro system.

***Index Terms*—Design understanding, debugging, waveform, waveform analysis, waveform analysis language (WAL), domain specific language, automation, HDL**

I. Introduction

The development of next-generation electronic systems poses significant challenges to all phases of the design process. In particular, the verification phase is the most time-consuming part. Verification aims to find design errors as early as possible, and verification is multidimensional [1]: Both, functional requirements and non-functional requirements (e.g. timing, performance, latency) have to be verified, and in practice there is often an intersection of the respective tasks and models used [2]. If one again breaks down the verification task, then it is dominated by debugging [3]. Even worse, debugging is rated as least predictable since it requires a deep design understanding [4]. The cornerstone for design understanding is the waveform. The waveform is generated during simulation of a *Hardware Description Language* (HDL) design and describes the circuit signals together with hierarchy information over time [5].

In both, the design phase and the verification phase of a digital system, waveforms are heavily used. Initially, directed test stimuli are created to see that the currently designed *Hardware* (HW) blocks are "alive" and produce some meaningful output. When the design matures, the verification plan is followed and advanced verification techniques, e.g. assertion-based methods together with coverage-based solutions, are employed [6], [7]. Along this highly iterative process, waveforms demonstrating expected behavior or unexpected behavior (e.g. in case of a failed assertion or a violated timing constraint) have to be analyzed and understood. For this task, (commercial) waveform viewers are utilized. Waveform viewers are software tools which allow viewing signal values over time. Besides selecting the radix of each signal and grouping signals together, the user can zoom in and out, can jump to the next time point where the value of a signal changes, can determine the time difference between two cursors, etc. However, while all these features help in understanding and debugging, **waveform viewing is a highly manual and tedious process**.

So far, most research has either concentrated on specific design understanding approaches such that the manual analysis of the waveform is reduced to a minimum, or it has been confined on the generation of "better" waveforms, e.g. by employing formal methods, reducing their length, or minimizing the signals involved in a failing trace (more details are discussed in related work). While these approaches have introduced automation in general, there has been almost **no progress for automating the analysis of waveforms**. The potential for automation becomes clear when looking at typical non-trivial analysis questions raised for waveforms dumped for instance when running *Software* (SW) on a processor:

- How many instructions per cycle are executed by the processor?
- What is the latency of the bus interfaces?
- What throughput is the bus achieving?
- When and why is the processor pipeline flushed or stalled during SW execution?
- Which parts of a program are executed on the processor?

In this work, we **bring automation to the analysis of waveforms** and introduce the *Waveform Analysis Language* (WAL)[1]. We have realized WAL as a *Domain Specific Language* (DSL) [8]. WAL is based on the idea that **signals from waveforms become variables**, and that **simulation time and design hierarchy are fundamental parts** of the language.

WAL Basis: For WAL, we first identified the essential operations for processing waveform data; this includes loading (multiple) waveforms, access to signals, time manipulation, and logical grouping of signals. Second, since verification (and regression) environments differ widely in terms of requirements and work-flows, we strive to maximize the versatility of WAL. Therefore, we decided to define the syntax of our proposed WAL DSL following the established Lisp principle of *symbolic expressions* (or S-expressions[2]) [9]. S-expressions have three main advantages: an extremely regular syntax based on lists, code and data are represented by the same data

[1]WAL is available open-source at https://github.com/ics-jku/wal

[2]An S-expression is an atom (also called symbol) or it is a list of S-expressions.

structure, and code generation is simple as only lists have to be created. As a consequence, we can:

1) feature a minimal and clean syntax for WAL,
2) easily integrate the above mentioned essential waveform operations as well as advanced operations via functions,
3) provide an *Intermediate Representation* (IR) well-suited as a compilation target from other languages, and
4) introduce a macro system to WAL which makes extending and modifying the language simple, opening the door to additional DSLs build in WAL itself.

To demonstrate the capabilities of WAL, we present four different case studies. The case studies cover the analysis of HW performance, combined HW/SW profiling, bus transaction analysis, and the implementation of an embedded DSL using WALs macro system.

Paper Structure: This journal paper includes and extends published material from the three previous conference papers [10], [11], [12]. We start by outlining the new contributions of this paper in the next paragraph. After a discussion of related work in Section II, Section III describes the waveform analysis problem. In Section IV, we introduce WAL: We present requirements, review S-expressions in the context of WAL, present the essential and advanced operations of WAL including the macro system. WAL core, our reference implementation of WAL is presented in Section V. The case studies are given in Section VI. Finally, the paper is concluded in Section VIII.

New Contributions: In comparison to the three conference papers [10], [11], [12], new contributions in this paper include: (1) a comprehensive overview of the major WAL functions, (2) the introduction of a macro system for WAL, (3) a new WAL standard library which is in part implemented using the new macro system, (4) new analysis concepts such as timeframes, configurable sampling, and (5) the enhancements and the addition of (new) case studies to showcase WAL.

II. Related Work

Design understanding and debugging is an active field of research. Several methods targeting specific problems and different abstraction levels have been proposed [13]. For example, this includes natural language techniques to derive assertions from specifications [14], feature localization in ESL models [15] or in RTL descriptions [16], assertion mining at RTL [17], identification of instruction pipelines using static analysis on the netlist [18], template-based understanding of circuit components [19], and reverse engineering at the gate-level [20], [21]. However, all these solutions focus on dedicated design understanding sub-problems and do not provide a generic user-programmable analysis for waveforms.

In general, advanced testbench constructs, for example provided by SystemVerilog, are relevant in this context. SystemVerilog offers several features and methodologies that can help in monitoring, capturing, and analyzing signal behaviors during simulation. This includes assertions, as already mentioned in the introduction. They capture complex temporal behaviors and are used to flag unexpected signal values or sequences during simulation.

However, the goal of assertion-based verification is to determine "only" whether the underlying temporal logic formula evaluates to true or false on a trace (or waveform). In contrast, WAL allows a much wider user-programmable analysis and application: With WAL programs, complex signal relations can be caught too, but then the user can perform arbitrary actions for all kinds of computations including the use of high-level data-structures, such as lists, hashmaps, etc. In principle, a user could perform certain analysis in the testbench of a design by utilizing, for instance, classes and the object-oriented features of SystemVerilog. However, such an approach is extremely complex, would require significant effort and also re-simulation, quickly reaching the limits of practicability.

The *Universal Verification Methodology* (UVM) [22] does not directly provide features for programmable waveform analysis, as it is primarily a methodology and a library for creating structured, reusable verification environments in SystemVerilog. UVM focuses on the creation, management, and use of verification components and data, such as stimulus generation, checking mechanisms, coverage collection, and reporting. Its primary goal is to improve the efficiency and reusability of the verification process.

To the best of our knowledge no waveform analysis language comparable to the user-programmable expressiveness of WAL is available (also not in the commercial solutions from Cadence, Synopsys, Siemens EDA, etc.).

III. Programmable Waveform Analysis Problem

As already stated in the introduction, after the generation of a waveform from a HW simulation (typically in a format such as *Value Change Dump* (VCD) [23]), the contained signals and their relations are traditionally analyzed by visual inspection using waveform viewers. The major reason is that waveforms are an elegant method of visually expressing concurrency and design hierarchy. Waveforms typically contain not only the value of signals over the simulation time, but also information about the signals (bitwidth, type) and about hierarchical scope information, i.e., what modules make up the design and how they are composed. Hence, debugging and understanding of sophisticated module behavior and inter-module interactions can be performed. However, as the analysis is manual it can quickly become tedious or totally impractical in case of complex or repeating problems. Therefore, we introduce user-programmable waveform analysis, transforming a manual repeating analysis problem into a one-time-only effort that scales with increasing complexity and waveform sizes and which is often reusable across projects.

In the following, we provide a simple but illustrative example for such an analysis problem. We will use this example throughout the paper when introducing the proposed WAL. Fig. 1 shows the waveform of a bus communication using the typical request-acknowledge protocol scheme.

Two components (`comp1` and `comp2`) are connected to the bus and the respective `req`/`ack` signals have been traced as can be seen in the waveform. Assume that the design team has to determine the average latency for each component attached to the bus for system optimization. To solve such a problem

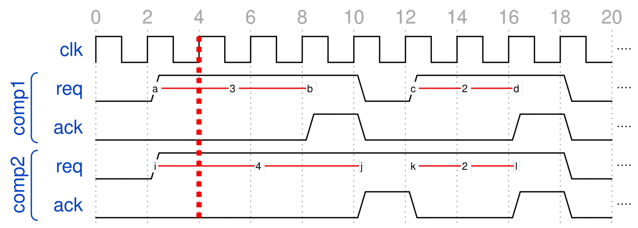


Fig. 1: Request-Acknowledge bus communication.

with a waveform viewer is practically not possible. More precisely, the task is to walk through the trace and count cycles where `req` is high and `ack` is low. Finally, this result has to be divided by the number of acknowledgments. In Fig. 1, for `comp1` we get $3 + 2 = 5$ (see marked lines a to b and c to d respectively) divided by 2, which gives the average latency of 2.5. Clearly, manual navigation and calculation on the waveform is a poor solution only. Even worse, for waveforms with tens of thousands of cycles this approach fails. In contrast, this calculation can be easily performed with WAL as we will show later.

Moreover, the design in the example has two components and therefore we are interested in the average latency over all, or a subset, of the components. This requires to extend the calculation from `comp1` to `comp2`, which would lead to doubling the code. For such problems, we added advanced features to WAL which allow for writing code in a generic and flexible way.

In the next section, we introduce WAL and show that the average latency can be easily determined using WAL.

IV. Waveform Analysis Language (WAL)

First, we consider the requirements on design and implementation of the WAL DSL (Section IV-A). Then, we briefly review symbolic expressions as proposed by Lisp, which we extend to accommodate the specifics of WAL and the HW domain (Section IV-B). We continue with the essential WAL operations (Section IV-C) and WALs macro system (Section IV-D). Finally, we close this section with a presentation of the advanced WAL operations in Section IV-E).

A. Design and Implementation Requirements

After conceptualizing WALs functional scope and before starting a first implementation we had to decide upon a suitable architecture. At first glance it seems advantageous to implement all functionality in a library of an established programming language (i.e. Python or C) as this provides a proven basis and is easy to pick up for most developers. Unfortunately, this approach has significant drawbacks to the versatility and expressiveness of WAL as all feasible languages are geared towards general purpose computing. This means, that expressing waveform analysis problems would require large amounts of boilerplate code, for example "getter" functions for signal access, as many waveform specific actions are not native to the language. However, we envisioned a system where all aspects of the waveform and HW design domain are **first-class citizens of the language**. Designing WAL as a DSL with waveform analysis in mind enables users to directly express their problems naturally instead of forcing their problem onto the paradigm of a different language. Finally, even though we developed a reference implementation of WAL from the ground up it is possible to implement WAL on top of other programming languages (similar to languages implemented on top of Racket [24]).

B. S-expressions and WAL Specific Syntax

Symbolic expressions (abbrev. as S-expressions), are common in languages related to Lisp, such as Common Lisp [25] or Scheme [26]. Fundamentally, S-expressions can be of two kinds: *atoms* or *lists*. Atoms are literals like numerical values (e.g., `1`, `0xff`), string values (e.g., `"text"`), Booleans (e.g., `#t`, `#f`), 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`.

A simplified definition of WALs syntax is shown in *Backus-Naur Form* (BNF) form in Fig. 2. Due to space limitations, the definitions for numbers, symbols, and strings are omitted (however, they mostly follow SystemVerilog formats). In particular, Fig. 2 focuses on the WAL specific additions to the S-expression syntax, namely the *resolved*, *sliced*, and *timed* constructs. These three special language features are valid WAL syntax, however they are transformed into regular S-expressions as we will shown in Table II.

Now, let us look at S-expressions in WAL, i.e. we consider them in the context of waveforms. As a consequence, the *symbols* of S-expressions are either signal names contained in a waveform, e.g. `top.module1.out`, or variable names defined in a WAL program. With respect to evaluation of an S-expression, we define the *current time index* (or just index) for a waveform at hand. In Fig. 1 this is nothing else than the red dashed line shown at position 4. So a signal name (symbol) is evaluated to the value of the signal at the current time index, for instance `comp1.req = 1` and `comp1.ack = 0` at the current time index 4.

Besides the access to signal values, all operations targeting the analysis of waveforms are integrated into WAL S-expressions using dedicated functions. In the following sections, we introduce these functions and demonstrate how they allow formulating compact and easy-to-use programs for waveform analysis.

C. Essential WAL Operations

We provide an intuitive introduction to the essential operations of WAL and therefore we incrementally develop a WAL program to solve the latency analysis problem as presented in Section III. For the essential WAL operations three main categories can be distinguished: waveform handling, signal access, and timing. As a foundation for all WAL expressions, WAL naturally implements all basic programming constructs (e.g. variables, loops, user functions). Table I summarizes most WAL functions and in the following we always refer to this table.

$$\begin{array}{rcl}
\langle\text{expression}\rangle & \models & \langle\text{atom}\rangle \mid \langle\text{list}\rangle \mid \langle\text{quoted}\rangle \mid \\
 & & \langle\text{sliced}\rangle \mid \langle\text{timed}\rangle \mid \langle\text{resolved}\rangle \mid \lambda \\
\langle\text{atom}\rangle & \models & \textit{number} \mid \textit{symbol} \mid \textit{string} \\
\langle\text{quoted}\rangle & \models & \texttt{'}\langle\text{expression}\rangle \mid \\
 & & \texttt{`}\langle\text{expression}\rangle \mid \\
 & & \texttt{,}\langle\text{expression}\rangle \mid \\
 & & \texttt{,@}\langle\text{expression}\rangle \\
\langle\text{resolved}\rangle & \models & \texttt{\#}\langle\text{expression}\rangle \mid \sim\langle\text{expression}\rangle \\
\langle\text{sliced}\rangle & \models & \langle\text{expression}\rangle\texttt{[}\langle\text{expression}\rangle\texttt{]} \\
 & & \langle\text{expression}\rangle\texttt{[}\langle\text{expression}\rangle\texttt{:}\langle\text{expression}\rangle\texttt{]} \\
\langle\text{timed}\rangle & \models & \langle\text{expression}\rangle\texttt{@}\langle\text{expression}\rangle \\
\langle\text{list}\rangle & \models & \texttt{(}\langle\text{expression}\rangle\langle\text{element}\rangle\texttt{)} \\
 & & \texttt{[}\langle\text{expression}\rangle\langle\text{element}\rangle\texttt{]} \\
 & & \texttt{\{}\langle\text{expression}\rangle\langle\text{element}\rangle\texttt{\}} \\
\langle\text{element}\rangle & \models & \textit{whitespace}\langle\text{expression}\rangle\langle\text{element}\rangle \mid \lambda
\end{array}$$

Fig. 2: Simplified BNF of WAL.

1) Waveform Handling: First, a waveform must be loaded in a WAL program in order to have access to the signal values. The **`load`** operator reads the waveform specified by the first argument and registers it with the optional *id* given as the second argument. Assume the waveform data from Fig. 1 has been dumped to the file `"waveform.vcd"`, as a first step we load this file into WAL under the id `t` as following: `(load "waveform.vcd" 't)`.

After a waveform has been loaded, its time index is set to the beginning to 0, and it is available to WAL expressions. The **`step`** operator can be used to step the time index forward and backward by a variable amount.

For example, `(step 2)` increases the time index of all loaded waveforms by 2, while `(step -1)` decreases the time index of all loaded waveforms by 1.

2) Signal Access: After loading the waveform containing the data in Fig. 1, we can start writing our WAL solution to determine the average latency. In a first basic version of this program, we want to detect when `comp1` requests the bus and when there is the corresponding acknowledgment. As mentioned before, waveform signals are first-class citizens of the WAL language. Therefore, to access the signal value at the current time index of a waveform, it is sufficient to write the full signal name (i.e. a global name of the form `top.sub.signal`).[3]

In our problem, a request is said to be acknowledged when both the `req` and `ack` signals are high. This condition can be expressed by a Boolean conjunction of the signals `comp1 .req` and `comp1.ack` using the following WAL expression: `(&& comp1.req comp1.ack)`.

In the same way we describe pending requests using the next WAL expression, when the `req` signal is high but the `ack` signal is low: `(&& comp1.req (! comp1.ack))`. This time, the signal `comp1.ack` is inverted using the `!` function since the component has not yet processed the request and thus `comp1.ack` is set to 0.

```
1 (load "waveform.vcd")
2 (define packets 0)
3 (define wait 0)
4 (while (step 2)
5   (when (&& comp1.req comp1.ack) (inc packets))
6   (when (&& comp1.req (! comp1.ack)) (inc wait)))
7 (print (/ wait acks))
```

Listing 1: Average Latency for comp1.

If multiple waveforms are loaded, signal name ambiguities must be resolved by specifying the waveform id in front of the full signal name (e.g. `w$comp1.req` vs. `w2$Top.sig`). The id in front of the name can be omitted if only one waveform is loaded.

Average latency for Component 1 in WAL: Now, we can combine the presented WAL functions to solve the average latency problem of Section III for `comp1`. The WAL program is shown in Listing 1. First, in Line 1 the waveform is loaded. As we are interested in the average latency wrt. the complete waveform, in Line 4 we step forward[4] until the end of the waveform is reached. The "core detection" of requests and acknowledgments is performed in Line 5-6. To compute the average latency we have to determine the number of all acknowledged packets. This is done in Line 5, where the variable `packets` is incremented when a request is acknowledged. For this condition, we inserted the previously introduced expression for acknowledged requests. In Line 6, the `wait` variable is incremented when the component has a pending, unacknowledged request using the other previously introduced expression. Finally, after the end of the waveform has been reached, we calculate the average latency using division and print it to the standard output in Line 7.

3) Timing: Often, interesting signal relations are not limited to a single time index. For example, detecting a value change on a signal requires observing two values of the same signal at different time indices. This could be achieved by temporarily storing the first signal value in a variable, but this quickly becomes inconvenient. WAL overcomes this problem by allowing to modify the time index of a waveform locally for a specific expression. For this, we introduce the *relative-eval* operator **`reval`** which takes a target expression and an offset expression that must evaluate to a signed integer, and evaluates the target expression with a locally changed time index according to the evaluated offset expression. The integer specifies the time offset at which the expression is evaluated relative to the current time index. For instance, detecting a signal value change can be expressed using the following WAL expression: `(!= (reval sig -1) sig)`. As relative evaluation is commonly needed, it can be abbreviated by appending an `@` followed by an offset to any expression. Using this shorthand syntax, the expression `(reval sig -1)` can be written as `sig@-1`.

As an example we assume HW designers have to check a worst-case requirement and therefore have to find pending

[3]It is also possible to extract specific bits from a signal value using slicing functions.

[4]The step size is 2 since we only want to sample data at positive clock edges which in this trace are located on every other index.

TABLE I: Overview of WAL functions. This overview is not complete, some functions are left due to space limitations.

Name	Description	Example
load	Load trace into WAL, with optional trace id t	`(load "trace.fst" 't)`
unload	Remove trace with trace id t from WAL	`(unload t)`
step	Add n to INDEX	`(step n)`
eval-file	Evaluate file f and make definitions available	`(eval-file f)`
alias	Creates an alias for a signal name	`(alias short very.long.signal.name)`
unalias	Removes an alias	`(unalias short)`
get	Get value from signal specified by string or symbol	`(get "top.sub.ready")`
reval	Locally modifies INDEX and evaluates body expression	`(= top.signal (reval top.signal -1))`
import	Import external functions (e.g., from Python)	`(import riscv)`
call	Call function from imported file	`(call riscv.decode tb.top.instr)`
+,-,*,/,&&,||	Arithmetic and logical functions	`(* a b (+ 1 2))`
<,>,=,<=,>=	Comparison functions	`(< 1 2)`
set	Define x and bind value v to it	`(define x v)`
set	Set variable x to value v	`(set [x v])`
let	Bind value v to x and evaluate *body*	`(let ([x v]) (+ x 4))`
if	Conditional Branch	`(if (< x 5) "small" "large")`
when	`(when c b ...)` equivalent to `(if c (do b ...))`	`(when is-sending (+ acc 1))`
unless	`(unless c b ...)` eqv. to `(if (! c) (do b ...))`	`(unless overflow "ok")`

cond	Tests multiple conditions and evaluates first satisfied condition
	else can be used as a default condition
		`(cond [sending (+ acc data)]`
		`      [receiving (print "Received:" data)]`
		`      [else (print "waiting")])`

while	Evaluate body as long as condition evaluates to true	`(while (! receiving) (print "."))`

do	Evaluates all expressions in body and returns result of last one	`(do (set [x (+ 1 2)])`
		`    (print x)`
		`    (+ (calc-delay) x))`

defun	Define a new function	`(defun times-two [x] (* 2 x))`

defmacro	Define a new macro	`(defmacro rev-args [xs]`
		`` `(,(first xs) ,@(reverse (rest xs))))``

lambda	Define a new anonymous function	`((lambda [x] (* 2 x)) 4)`
print	Prints the arguments to the standart output	`(print "hello" (+ 1 2))`
printf	Prints the arguments to the standart output depending on format string	`(printf "%s: %d" "Res" (+ 1 2))`
list	Evaluate arguments and put into new list	`(list 1 2 (+ 1 2) "abc")`
length	Returns the length of a list	`(length (list 1 2 3))`
min, **max**	Returns the min or max value of a list	`(min (list 1 2 3 4))`
sum, **average**	Returns the sum or average value of a list	`(sum (list 1 2 3 4))`
map	Creates a new list by applying a function to each element in a list. Function must take exactly one argument.	`(map (lambda [x] (* x 2)) (list 1 2 3))`
reverse	Returns the passed list in reversed order	`(reverse '(1 2 3))`
array	Create a new array, entries can be passed are tuples	`(array ["a" 1] ["b" (+ 1 1)])`
seta	Update value in an array	`(seta array-var "b" 5)`
geta	Get value from an array	`(geta array-var "b")`
geta/default	Get value from an array, if key not in array return default value	`(geta array-var "default" "b")`
mapa	Returns a list by applying function to every key-value pair in an array. The function must take 2 arguments.	`(mapa (lambda [k v] (* 2 v)) array-var)`
resolve-scope	Takes the argument and appends it to the current scope to form a full signal name	`(resolve-scope signal)`
	The shorthand for **resolve-scope** is ~	`~signal`
in-scope	Evaluates the body expression in a scope. Inside signals can be scope-resolved	`(in-scope 'top.sub ~ready)`
in-scopes	Evaluates the body expression in all scopes	`(in-scopes '(top.sub1 top.sub2) ~ready)`
resolve-group	Takes the argument and appends it to the current group to form a full signal name	`(resolve-group signal)`
	The shorthand for **resolve-group** is #	`#signal`
groups	Returns a list of all groups that contain the signals passed as arguments	`(groups clk)`
in-group	Evaluates the body expression in a group. Inside signals can be group-resolved	`(in-group 'tb.dut (print #clk))`
in-groups	Evaluates the body expression in multiple groups	`(in-groups (groups clk) (print #clk))`
find	Returns a list of all indices at which expression evaluates to true	`(find (&& top.ready top.valid))`
count	Counts at how many indices the expression evaluates to true	`(count (&& top.ready top.valid))`
whenever	Evaluates the body expressions at each index where the condition is true	`(whenever (&& a (! b)) (print a))`
slice	Returns a slice of bits from a signal.	`(slice top.sig 1 0)`
	The shorthand for **slice** is `[h:l]`	`top.sig[1:0]`

timeframe	Evaluates body expressions in a new local timeframe.
	When the timeframe is exited INDEX is restored
		`(timeframe (step 10)`
		`           (print INDEX))`

sample-at	Sets the sampling points to the indices passed as an argument	`(sample-at (find (&& (! clk@-1) clk)))`

```
(&& (&& comp1.req (! comp1.ack))
    (&& comp1.req (! comp1.ack))@1
    (&& comp1.req (! comp1.ack))@2)
```

Listing 2: Detecting continuous pending requests.

TABLE II: Special shorthand syntax in WAL .

Special Syntax	Transformed into
'expr	(**quote** expr)
`expr	(**quasiquote** expr)
,expr	(**unquote** expr)
expr@off	(**reval** expr off)
~symbol	(**resolve-scope** symbol)
#symbol	(**resolve-group** symbol)
expr[i]	(**slice** expr i)
expr[h:l]	(**slice** expr h l)

requests that are persisting at least three consecutive cycles. This can be expressed as shown in Listing 2.

D. Macro System

One of the defining features of programming languages in the Lisp family is that most of them have very sophisticated macro systems. Compared to macros in other languages, they are more than mere text substitution, and since they work on S-expressions, they can operate on the real structure of a program. Additionally, at expansion time, Lisp macros can utilize the full language to generate code as they can be thought of as regular functions that just return program code which is inserted in place of the macro call. This powerful macro system can be used to expand the language to ones needs, for example, to define new language constructs. We will give examples for this in Section IV-D and Section IV-E3, respectively.

Before a WAL expression is evaluated, it is analyzed if it contains any macro applications. If this is the case, the macro function is called, and the macro application is replaced by the result of the macro function.

Using the **defmacro** keyword, new macros can be defined. Defining macros is very similar to defining functions, as the **defmacro** function also expects a name followed by a list of arguments while all additional arguments form the body of the macro.

Listing 3 shows a macro that can be used to step forward until a condition is true. Since in WAL code and data are represented by the same data structure[5], generating code is as simple as generating some nested lists. It is important to think about when an expression should be evaluated. If it should be a part of the code that is generated by the macro and not be evaluated instantly, it must be *quoted* using the **quote** function or its shorthand '. For example, we have to quote the (**step**) expression in Line 4 or else the step function would be evaluated immediately leading to a wrong INDEX[6]. However, evaluating (**step**) like this during the macro expansion is not what we want, and therefore we quote it to put the expression as a datum into the list. This way, we are able to place the

[5]This is called homoiconicity.

[6]See also Section IV-E3

```
1 (defmacro step-until [condition]
2   "Step forward until condition is true"
3   (list 'while (list '&& (list '! condition)
4                          '(step))
5                'INDEX))
```

Listing 3: Defining WAL macros with the **defmacro** function.

```
(defmacro step-until [condition]
  `(while (&& (! ,condition) (step)) INDEX))
```

Listing 4: The step-until macro with quasiquote.

(**step**) expression in the code the macro generates which is now evaluated only when the generated code is evaluated and not at expansion time. This resulting list can now be evaluated just as if it would be read from a WAL file.

To expand a macro without evaluating the resulting code, for example for debugging purposes, the **macroexpand** function can be used. For example, expanding (step-until overflow) results in the following expression (**while** (**&&** (**!** overflow) (**step**)) **INDEX**).

Since WAL expressions are represented by lists, macros that expand to more complex expressions generate deeply nested list structures. The above macro definition is very short, however, the expression generated by the macro is hard to read, since all nested lists are constructed manually and so, the macro contains many calls to the **list** function and many required quotations. For example, if we want to generate an expression of the form (**+** x (***** y z)), were x, y, and z are variables that contain other expressions which should be inserted into the expression, we would have to create a nested list by hand: (**list** '**+** x (**list** '***** y z)). This is hard to read since it contains noise (e.g., the **list** function calls) and it does not look like the desired result (e.g., the **+** operator is not the first element in the list).

To improve the readability of macro definitions, WAL implements the **quasiquote** function known from other Lisp languages. A quasiquoted expression is evaluated like a normal quoted expression except that sub-expressions can be evaluated using the **unquote** function. Similar to the normal **quote** function, **quasiquote** and **unquote** have shorthands (c.f. Table II). Using those two functions, the previous nested list can be generated with `(**+** ,x (***** ,y ,z)). The *step-until* macro can now be defined much more readable, as shown in Listing 4. Here, the quasiquoted list will be returned as is, with the exception that the condition expression will be evaluated and replaced by the result that was computed.

Reducing WAL core using Macros: Using the macro system we can now implement a large part of WALs functionality as WAL macros and functions. Instead of implementing these functions inside a WAL interpreter, they are now implemented in a standalone WAL program that is evaluated before a WAL program is run. This *WAL standard library* significantly reduces the complexity of WAL interpreters, and hence WAL core, since the number of functions that have to be implemented can be reduced.

Listing 5 shows two examples for WAL functions that were

```
(defmacro count [condition]
  `(length (find ,condition)))

(defmacro cond args
  (fold
    (lambda [acc branch]
      (let ([condition (if (= branch[0] 'else)
                            #t
                            branch[0])]
            [then (rest branch)])
           (if acc
               `(if ,condition (do ,@then) ,acc)
               `(if ,condition (do ,@then)))))
    '()
    (reverse args)))
```

Listing 5: Exemplary macros from the WAL standard library.

```
(cond [(= state 0x8) (print "Sending")]
      [(= state 0x4) (print "Received:" data)]
      [else (print "waiting")])
```

Listing 6: Examplary usage of the **cond** macro.

previously implemented inside the interpreter, but which are now implemented as macros in the WAL standard library file. The first macro, **count**, counts how often a condition is true on the complete waveform by combining the **length** and **find** functions of WAL. It is expanded to calculating the length of the list returned by a call to the **find** function with the given condition.

The second macro, **cond**, constructs nested *if-else* expressions by iterating over the arguments given to the **cond** macro. These arguments are lists where the first element is a condition and the rest are expressions that get evaluated if the condition evaluates to true. This macro is a good example to showcase how macro expansions can take advantage of the full WAL language as it uses several advanced WAL functions such as **fold**, **lambda**, and variable assignments to generate the resulting code.

An exemplary usage of the **cond** macro is shown in Listing 6. There, the **cond** macro is used to print a log message based on a *state* register. Listing 7 shows the nested if-else expressions generated by the **cond** macro. Every condition of the **cond** macro is translated into an if-else expression except the last one where the *else* condition is translated into an always true statement, therefore, the if expression can be optimized away.

E. Advanced WAL Operations

The expressiveness of WAL, based on the essential operations as introduced in the previous section, is sufficient for a wide range of analysis problems. However, the applicability of WAL can be significantly improved by adding advanced features. This allows to write much more compact, more generic and much easier to read WAL programs.

1) Calling External Code: WAL enables developers to write concise, powerful and easy-to-use programs for waveform analysis. On the other side, many problems not related to waveform analysis (such as UI or databases) are already available in libraries for other programming languages. Combining WAL with other programming languages saves time and helps to integrate WAL into complex work-flows. Therefore, WAL enables users to tap into the large ecosystems of other programming languages. Using the **import** function, external code in another language[7] can be imported into running WAL programs. After importing, external functions can be called using the **call** function.

2) Logical Grouping: Our example design contains two components connected to the bus, `comp1` and `comp2`. Both components share wrt. the bus communication interface a structural similarity (e.g., same signals). Coming back to our example, the problem *"How is the latency of `comp1`?"* is also valid for `comp2` or any other component attached to the bus. An elegant solution requires the separation of the core problem and the concrete signal names. WAL supports writing these separated *generic* expressions through a set of concepts and functions.

First, we introduce the concept of a *group*. A group is a set of signals which are semantically connected (e.g., the signals of a bus). Groups are defined by a prefix (a partial signal name) and a set of postfixes for which the combination *prefix* + *postfix* results in an existing signal name for every postfix. For example, the waveform in Fig. 1 contains two groups, `"comp1."` and `"comp2."`, for the set `req` and `ack`. Users can search the design for groups using the **groups** function (i.e., `(`**groups** `"req" "ack")` for the example).

To make use of a group, it has to be *captured* first. Capturing a group, defines this group as the current active group and allows accessing signals in the group using just the postfixes. Groups are captured using the **in-group** operator, which takes a `group` and then evaluates the `body` expression. The **in-groups** function works in the same way, but expects a list of groups and evaluates the body expression once for each of these groups. During the evaluation of the body expression the specified group is marked as the current group (the active Current Group is available using the **CG** special variable). Accessing signals in a group just by a postfix is called *resolution*. In the body of an **in-group[s]** expression, signal names can be resolved using the **resolve-group** function. This function takes a symbol and appends it to the captured group. If the resulting symbol refers to an existing signal, the value of this signal at the current time is returned. As wrapping all signals in **resolve-group** function calls leads to increased verbosity, a shorthand for this function is to add a **#** in front of a symbol (cf. Table II).

WAL expressions can make use of the hierarchical information of waveform data. The *scoping* concept allows evaluating WAL expressions in selected scopes (i.e., the submodules of the design). Scoping is available via the ~ shorthand and the

```
(if (= state 0x8)
    (print "Sending")
    (if (= state 0x4)
        (print "Received:" data)
        (print "waiting")))
```

Listing 7: Expansion of the **cond** expression in Listing 6.

[7]This is for example code in the host language of the WAL interpreter.

```
1 (whenever bus_ready
2   (timeframe
3     (let ([id bus_id])
4       ;; step until next ready
5       (print "Request from " INDEX)
6       (step-until (&& bus_ack (= id bus_id)))
7       (print " to " INDEX))))
```

Listing 8: By using a timeframe, `step-until` does not interfere with the **`whenever`** function.

scoping functions. Since scoping works similar to grouping, we omit further details.

3) Timeframes: In WAL, each trace has a global **`INDEX`** that points into the trace and thus influences which value is returned when a signal is evaluated. Often however, programs need to modify the **`INDEX`** while walking over the trace, but then need to restore the original **`INDEX`**. Of course, this behavior can also occur nested arbitrarily deep, which in turn results in the creation of a stack-like structure. This functionality could be implemented without additional functions by manually storing and restoring the current **`INDEX`**, however, WAL aims to make describing analysis problems as easy and the analysis code as descriptive as possible.

Take for example a bus on which multiple transactions can be "live" at the same time, i.e. a new request can be generated before the previous transactions are acknownleged. For this, each transaction is tagged with a transaction id.

A WAL program to analyze a system like that is shown in Listing 8; please ignore the **`timeframe`** function in Line 2 for now. This program visits every timepoint at which a new transaction is seen on the bus. It then stores the transaction id and steps forward until the transaction with this id is acknowledged and prints a debug message. Now, after the program reaches the time at which the transaction is acknowledged it returns to the outer **`whenever`** loop, however, as the **`INDEX`** was modified inside the previous iteration it does not continue at the timepoint after the `bus_ready` signal was high but after the transaction was acknowledged. This is a problem, since new transactions could have been started between these two timepoints.

To handle situations like this in a convenient way we propose so-called timeframes. A **`timeframe`** is an environment that stores the current **`INDEX`** for each loaded trace (accessible inside the timeframe via `TIMEFRAME-START`), evaluates its body expressions and then restores the indices back to their original values. Now, the problem in Listing 8 can be solved by wrapping the **`whenever`** body inside a timeframe. By this, the **`INDEX`** is restored back to its original value when the body of the **`whenever`** function was entered.

4) Modifying the Sampling Points: WAL makes no assumptions about the semantics of signals. Therefore, for example, *clk* or *reset* signals have no special meaning that change the behavior of WAL. However, a lot of analysis problems are best formulated using the clock-sampled synchronous semantics known from RTL modeling or from assertion languages. Please recall, that the **`INDEX`** is incremented, whenever a value changes inside a trace. Compare this to clock-sampled semantics where users might expect that the **`INDEX`** is only incremented for example on each new rising clock edge. This also influences the semantics of relative evaluation. For example, `signal@-1` should now return the value `signal` had at the last clock cycle.

To achieve this, we present the **`sample-at`** function which changes at which time points signal values are sampled from the trace and how the **`INDEX`** is calculated. This function expects a list of indices that has to be a subset of all indices of the loaded trace. After **`sample-at`** is called, signals can only be read at time points that are included in this subset of indices. Additionally, the new time points are given new indices starting from 0. This process of changing the sample points of a trace is shown in Fig. 3. The trace with the original sample points above the waveform is shown in Fig. 3a (see arrows). Please note, how each change of a signal introduces a new sample point and a new index (see **`INDEX`** 2 for an asynchronous new index). In this example, we want to sample from the trace at each rising edge of the *clk* signal, therefore, we get a list of all indices at which this condition is true by evaluating `(`**`find`**` (rising clk))`. The indices that are contained in the returned list are shown in Fig. 3b. Finally, by passing this list to the **`sample-at`** function, we update the sample points of the trace which results in the new indices shown in Fig. 3c. Now, the expressions `data@-1` has the meaning of the value of the *data* signal at the previous rising clock edge instead of the value of the *data* signal at the time of the last signal change.

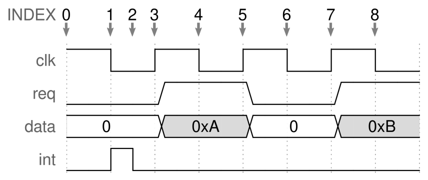
The **`sample-at`** function is not only limited to sampling at rising clock edges. For example, in Fig. 4 we sample only when both *clk* and *req* signals have rising edges thus, modeling a valid transaction on a simple bus. Now, the expressions `data@-1` has the meaning of the *data* signal at the last valid transaction.

V. WAL Interpreter

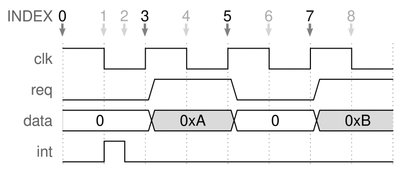
We have implemented a Python-based interpreter for WAL which we refer to as the *WAL core*. WAL core is a Python package that contains an API for WAL integration in Python applications, a standalone interpreter to run WAL programs in a terminal, and a *Read-Eval-Print-Loop* (REPL) shell for interactive WAL programming. WAL core can analyze waveforms in the *VCD* and *FST* [27] formats.

The architecture of WAL core is depicted in Fig. 5. The inputs to WAL core are user-defined analysis programs (see top of the figure). These programs are either WAL programs or programs in a different language that are transpiled to WAL. In the first case, the WAL core **Reader** transforms WAL programs from text form to *Internal Code*. The same flow via the Reader is also possible for a **Custom Frontend**. Alternatively, a frontend has the option to utilize its own reader, generating WAL internal code directly without relying on the Reader module in the WAL core. An example of this approach is our *Waveform AWK* (WAWK) language which is presented in detail in [10].

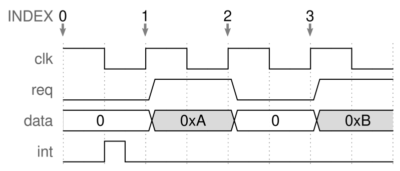
Next, the internal code is run through passes. In Fig. 5, these are **Macro Expansion** followed by **Optimization**. The macro system embedded within WAL empowers users to construct



(a) Original sample points.



(b) New sample points.



(c) Updated samplepoints and new indices.

Fig. 3: Changing the sample points of a trace with `(sample-at (find (rising clk)))`.

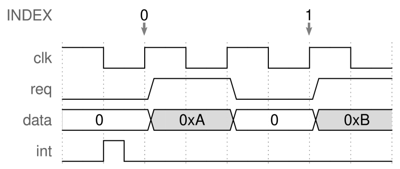


Fig. 4: Sampling at `(&& (rising clk) (rising req)))`.

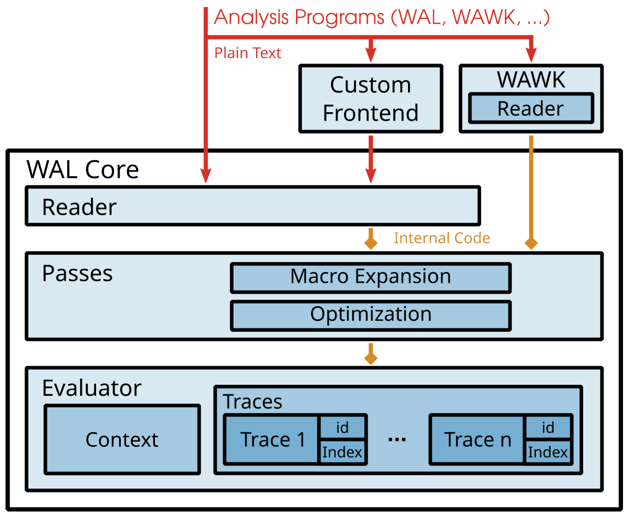


Fig. 5: Architecture of WAL Interpreter (referred as WAL core).

complex operations by composing simpler operations, yielding numerous advantages across all language levels. Furthermore, this feature facilitates the creation of libraries, amplifying the language's capabilities.

The result of the passes is then evaluated, i.e. executed, by the **Evaluator**. The evaluator is also the place where the execution state is stored. The execution state consists of the loaded traces and the context which contains all the defined variables.

WAL core is not limited to just executing WAL program files, it also allows embedding WAL into Python programs. Using the API of WAL core, non-trivial analysis tasks can be easily performed in Python applications, which we demonstrate in Section VI. In addition, WAL core provides access to the internal WAL structures, which in combination with the API, enable a clean interface for WAL-IR applications. Currently WAL core is not optimized for performance, however, it is intended as an executable specification that is both easy to understand and easy to extend to enable further research.

Further, with the macro system we presented in Section IV-D we shifted more functionality out of WAL core towards the standard library. This makes WAL core even smaller and provides a lot of functionality defined in WAL itself which is now ready to use in alternative WAL implementations.

VI. Case Studies

In this section, we explore case studies that represent different WAL use cases. First, in Section VI-A, we analyze the performance of several RISC-V processors using WAL. We utilize WALs strong abstraction capabilities to apply the same generic WAL code to a wide range of RISC-V processors with only a small amount of additional glue code. Then, in Section VI-B we expand the scope of the analysis to the profiling of SW running on RISC-V processors using WAL and the symbol information contained in the application binaries. Next, in Section VI-C we present an *APB* monitoring function which can be used to perform various tasks by providing different callback functions. Finally, in Section VI-D, we utilize WALs macro system to create a new SQL inspired DSL which is embedded in WAL that allows easily gathering large amounts of data with simple queries.

A. RISC-V Performance Analysis

Since RISC-Vs introduction, we have seen an explosion of, often freely, available RISC-V cores. However, this development brings its set of challenges since the huge number of available RISC-V cores, which are often highly configurable and extensible, makes it very hard and time-consuming for both, designers and users, to compare different cores and core configurations against each other [28], [29]. One method of comparing different cores are benchmark applications that perform a set of representative computing tasks (see e.g. [30] or [31]). They measure the run time of these tasks and typically give a score that indicates how well a core handled the tasks. Yet, these benchmarks are SW applications that can only measure the performance of the core indirectly, for example via the run time, without the possibility of directly accessing the microarchitectural state of the core. RISC-Vs performance counters offer another way to acquire performance information directly from the core. However, not every core implements the same counters, and it is difficult to add counters since this requires changes to the microarchitecture of the core itself.

When it comes to performance analysis, waveforms can be considered the ground truth since all metrics of either SW benchmarks or performance counters can be recreated from the waveform data. The waveform captures all microarchitectural information of the processor over the complete run time of the executed application. Based on this comprehensive data, arbitrarily complex performance metrics can be constructed that can measure the performance of each component of the core in detail (e.g., the pipeline, caches, or the bus interface).

In this section, we demonstrate the use of WAL to analyze performance metrics of various RISC-V cores. These cores range from extremely area efficient ones [32] to pipelined cores with higher performance and many configuration options [33], [34]. With WAL, it is possible to split the analysis problem into a general and a core-specific part. Thus, we can write the code in such a way that the core analysis logic is independent of any core-specific information. In this section, we present a generic WAL riscv-library, thanks to which the analysis and comparison of various significantly different RISC-V cores is possible with just a few lines of processor-specific glue code.

1) Instructions Per Cycle (IPC): First, we analyze the raw performance of each core in terms of executed IPC. Since all analyzed cores are single core architectures, the best theoretical IPC score in this case study is 1.0 instructions per cycle. This means that the core executes and commits one instruction in each clock cycle. However, this is almost impossible to achieve, for example, due to branching and memory induced delays.

The WAL program for IPC analysis is split into two separate parts, a generic and core-independent analysis part and the core-specific code, which has to be provided by the user. Listing 9 shows the code for the generic IPC analysis program in WAL. This function performs the IPC analysis for all waveforms passed in the *traces* parameter. For each trace, first, the trace is loaded in Line 3 and then the optional *setup* function is called in Line 4. The optional *setup* and *clean-up* functions can be defined by the users to perform core-specific setup and clean operations. Then, the number of executed instructions is calculated in Line 5 using the user-supplied *is-valid* and *instr-done* functions (see below). The idea is to count how often the predicates *is-valid* and *instr-done* evaluate to 1 on the waveform. Next, the resulting IPC value is calculated in Line 6. We divide the number of total valid cycles by the number of executed *instructions*, take the reciprocal value, and print it in Line 7. Finally, the optional *clean-up* function is called, and the trace is unloaded from the WAL environment in Line 9.

To perform the IPC analysis on a new RISC-V core, users only have to provide the two *is-valid* and *instr-done* functions. Lines 2-5 in Listing 10 show the implementations of these functions for the IBEX processor [33]. The IBEX processor always sets the *instr_done* signal inside the *id_stage_i* module to 1 whenever an instruction is completed. Therefore, the *instr-done* function only has to return the value of this signal. The IBEX core executes instructions when the clock is rising and reset is low, which is checked in the *instr-valid* function.

2) Pipeline Stall Analysis: Our WAL riscv-library also provides a function to calculate the relative number of cycles at which at least one stage of the pipeline was stalled. This metric is useful, for example, to assess how efficient the branch prediction is working. Similar to the IPC analysis the pipeline stall function is also written in a generic way such that new cores can be easily analyzed by providing certain functions. Listing 11 shows the implementations for the required `is-stalled` and `is-valid` functions for the VexRiscv core. In Line 1, we first create a variable that will be used to store the names of all pipeline stages that were found in the core. This variable will be updated by the `setup` function (Line 3) whenever a new trace is loaded. The number of pipeline stages of a VexRiscv configuration pipeline can be changed, however, our `setup` function will work for every number of pipeline stages automatically since each pipeline stage contains an “isMoving” signal.

The `is-stalled` function (Line 6 in Listing 11) should return true whenever at least one stage of the pipeline is stalled. To check this, we create a list of the states of each pipeline stage by iterating over all stages and getting the signal value of the *isMoving* signal. If this list contains a 0, we know that one of the stages is stalled since it is not moving forward.

Finally, the `is-valid` function (Line 10) should return true at each valid cycle, i.e., a rising clock edge and no reset.

3) Results: Table III shows the analysis results for multiple open-source and commercial RISC-V cores. The name of the core is shown in the first column and the analyzed configuration is shown in the second column. The last two columns contain the analysis results.

With our generic WAL riscv-library we were able to analyze two detailed performance metrics of cores, ranging from extremely small bit-serial cores all the way to highly configurable commercial cores [35]. Further, only a few lines of code are required for each core to be made compatible with our WAL riscv-library and, if written accordingly, this glue code is applicable even to very different configurations of the same core.

TABLE III: Analysis results.

Core	Configuration	IPC	Stalled Cycles
SERV	servant	0.02	*not pipelined*
PicoRv32	default	0.24	*not pipelined*
VexRiscv	microNoCsr	0.33	63%
VexRiscv	smallest	0.33	66%
VexRiscv	smallAndProductive	0.42	54%
VexRiscv	smallAndProductiveICache	0.47	51%
VexRiscv	twoThreeStage	0.47	48%
VexRiscv	secure	0.57	42%
VexRiscv	linux	0.59	38%
VexRiscv	full	0.57	35%
VexRiscv	fullNoMmuMaxPerf	0.63	33%
IBEX	default	0.63	48%
IBEX	icache	0.89	19%
TGC	3-Stage	0.61	65%
TGC	4-Stage v1	0.72	49%
TGC	4-Stage v2	0.70	45%
TGC	4-Stage v3	0.70	44%
TGC	4-Stage v4	0.68	43%
TGC	5-Stage	0.78	40%

B. Waveform-based Profiling of RISC-V Binaries

In this section, we present how RISC-V binaries, that are executed on simulated cores, can be profiled using WAL. For this we utilize the symbol information contained in the binary and match this information against the instructions which were executed during simulation. The profiling is implemented as a

```
(defun calc-ipc [traces]
  (for [trace traces]
       (load trace t)                                       ;; load the trace
       (when (defined? 'setup) (setup))                     ;; if defined, run setup function
       (let ([instructions (count (&& (is-valid) (instr-done)))] ;; count num of executed instructions
             [ipc (/ 1 (/ (count (is-valid)) instructions))])    ;; calculate ipc value
         (printf "%40s: %15.2f\n" trace ipc)                ;; print results
         (when (defined? 'clean-up) (clean-up))             ;; if defined, run clean-up function
         (unload t))))                                      ;; unload the trace
```

Listing 9: Generic WAL function for the IPC analysis.

```
(defun is-valid []
  (&& (rising TOP.IO_CLK) TOP.IO_RST_N))

(defun instr-done []
  TOP.ibex_simple_system.u_top.u_ibex_top.
    ↪ u_ibex_core.id_stage_i.instr_done)
```

Listing 10: IBEX specific code for the IPC analysis.

```
(define stages '())

(defun setup []
  (set [stages (groups "isMoving")]))

(defun is-stalled []
  (in 0 (for/list [stage stages]
             (in-group stage #isMoving))))

(defun is-valid [] (&& (rising TOP.clk)
                       (! TOP.reset)))
```

Listing 11: VexRiscv specific code for the pipeline stall analysis.

```
1  functions = ranges(BIN)
2
3  wal = Wal()
4  wal.load(VCD)
5  # config script for core-specific names
6  wal.eval('(eval-file config)')
7  wal.eval('''
8  (defun count-function [addr]
9    (for [f funcs]
10     (when (&& (>= addr f[1]) (<= addr f[2]))
11       (seta dist
12          f[0]
13          (+ (geta/default dist 0 f[0]) 1)))))
14 ''')
15
16 # calculate the time spent in each function
17 dist = {}
18 instruction_executed = wal.eval('''
19 (whenever (fire)
20           (count-function (pc))
21           (inc ninstr))''',
22   funcs=functions, dist=dist)
```

Listing 12: Python code for function profiling using WAL.

```
(defmacro fire []
  `(&& (rising TOP.clk)
       (! TOP.reset)
       TOP.VexRiscv.lastStageIsFiring))
(defmacro pc [] 'TOP.VexRiscv.lastStagePc)
```

Listing 13: VexRiscv specific code for the function profiling.

Python application (an excerpt of this application is shown in Listing 12) that handles both, the extraction of symbol information, and the waveform analysis using WAL. First, the application extracts function address ranges from a given ELF binary using the *nm* command from the RISC-V toolchain (this happens in the function *ranges* in Line 1 of Listing 12 which we omit for brevity). All information about the functions are stored as triples consisting of the function name, the start address, and the end address of the function.

Next, the WAL interpreter is instantiated in Line 3 and the simulation trace is loaded in Line 4. The analysis is processor independent and relies on two definitions (that can be implemented as either functions or macros) that must be implemented for a specific core, *fire* and *pc*. The *fire* definition, when evaluated, should return true when the core finishes an instruction at the current **`INDEX`** and false otherwise. The *pc* definition, when evaluated, should return the program counter value that is associated with the currently finished instruction. Listing 13 shows macros that implement the required functionality for the *VexRiscv* processor. The core of the profiling analysis is the function *count-function* (Line 8-13 in Listing 12) which matches the current program counter against the function address ranges read from the binary. If the program counter lies within a functions address range, the counter associated with this function in the `dist` (short for function distribution) array is incremented.

This `dist` array is instanciated in the Python code in Line 17 and passed as an argument to the WAL interpreter when the analysis is performed (Line 22). By passing the array into the WAL interpreter like this, it is placed in the WAL execution environment and the WAL code can directly work on it. This showcases, how WAL and Python code can interact with each other, thus enabling developers to partition the code such that each sub problem can be implemented in the most natural language.

The main analysis loop is shown in Line 18-22 of Listing 12. This expression will traverse the full trace calling `count-function` whenever an instruction is finished by the processor. Additionally, it also counts (and returns to the Python code as the results of the whenever expression which is the last evaluated statement of the body) the number of instructions that have been executed overall.

Listing 14 shows an excerpt of the profiling results, listing the function, its address range, and the number of times an instruction from this range was executed.

```
bss_loop   [0x800000cc-0x800000d8]: 11273
puts       [0x80000e38-0x80000e78]: 9527
printf     [0x80000d34-0x80000e34]: 7362
__divsi3   [0x80002ec4-0x80002f44]: 6344
printf_s   [0x80000c40-0x80000c74]: 1565
printf_d   [0x80000c7c-0x80000d08]: 1325
main2      [0x800002f8-0x80000a78]: 400
__mulsi3   [0x80002ea0-0x80002ec0]: 343
...
```

Listing 14: Excerpt of profiling results.

C. Analyzing APB Interfaces

The *Advanced Peripheral Bus* (APB) protocol is a widely used communication interface for low-speed devices. It is a synchronous protocol with two independent buses, one for reading and one for writing, which, however, cannot be used at the same time.

In this section, we present a monitor function which can detect events (e.g., the end of a successful write transaction) on an APB interface. Users can specify a callback function for any of the supported events, which is called if this event is detected. By plugging in different kinds of callback functions the general APB monitor can be used to perform a wide range of tasks.

The code for the APB monitor function is shown in Listing 15. Five parameters are expected by the `monitor-apb` function, the group of the APB bus which should be analyzed and four callback functions.

```
 1  (defun monitor-apb [g setup wait read write]
 2    (in-group
 3     g
 4     (whenever (&& (rising #pclk) (= #presetn 1))
 5       (case (list #psel #penable #pready)
 6         [(1 0 0) (when setup (setup))]
 7         [(1 1 0) (when wait (wait))]
 8         [(1 1 1) (if #pwrite
 9                      (when write
10                        (write #paddr #pwdata))
11                      (when read
12                        (read #paddr #prdata)))]))))
```

Listing 15: Generic APB monitoring function.

Lines 4-12 contain the main logic of the function. Since APB is a clock synchronous protocol, the function visits every timestamp at which the APB clock is rising and the active-low APB resetn signal is high. At each timestamp at which this condition is true, the state of the bus can be detected by observing the `psel`, `penable`, and `pready` signals. On Line 5 the current values of these signals are sampled and collected into a list. Then, we pattern match this list using the **`case`** function against three constant lists that represent the *setup*, *wait*, and *enable* states of an APB bus. If the sampled signals now match one of the three lists, the corresponding callback function is executed, but only if a valid callback function for this event is available. Only for the *enable* state on Line 8-12 the code further checks if the transaction is a read or a write access. The main benefit of this pattern matching approach is that it reduces the number of nested if expressions and that it plainly documents the possible types of APB events. However, since the monitor function is not a protocol checker, the pattern matching also requires that the bus adheres to the specification.

Listing 16 presents three exemplary use cases for the `monitor-apb` function.

APB Stream Logging: First, in Listing 16 on Lines 1-6 a macro is defined which can be used to print a report of the transaction on the specified APB bus `g`. The macro expands to an application of the `monitor-apb` function, in which only the `read` and `write` callbacks are used. For them, two lambda functions are supplied that print a string with information about the transaction. The other callbacks are disabled by supplying them with zeros instead of functions.

APB Slave Memory Restoration: Often, memories are not dumped into waveforms due to space and efficiency reasons. Therefore, the memory module acts as a black box since the waveform never has its full state. However, by monitoring the transaction on a bus it is possible to reconstruct the memory from a certain initial state. Line 8-12 in Listing 16 show a function that, if passed the group to an APB bus, restores the memory content based on observed write transactions. This is done via the array defined on Line 9. This array keeps track of the current value of every address and is updated in the `write` callback on Line 11.

APB Transaction Delay: Lines 14-25 in Listing 16 present a function which, similar to the earlier example, calculates the average delay on a given APB bus. This is implemented using the `setup`, `write`, and `read` callbacks. The logic behind this code is that the current timestamp (**`TS`**) is stored in the `start` variable whenever the APB bus enters the *setup* phase (see `on-setup` function on Line 17). When, either a read transaction or a write transaction is detected the time that passed since the last setup phase is calculated (Lines 18-19) and appended to a list. Finally, the average of the list is computed and returned as the result of the `apb-avg-delay` function. Note, that by providing the `on-enable` function to only the `read` or `write` callbacks the delay analysis can be performed for only one transaction type if required.

D. SELECT: An Exemplary DSL Built On Macros

The ease of defining new embedded DSLs using macros is one of the key benefits of using Lisp-like languages. They allow lifting a problem to a much higher abstraction which can help solving the problem in a simple and efficient way. Further, DSLs created using macros are not to be used only in a standalone fashion but can be naturally embedded in host language programs.

In this section, we present a case study for SELECT, a DSL written as a WAL macro. This DSL presents a new, much more concise and declarative syntax for reading the values of multiple signals at specified timestamps. In particular SELECT, is highly inspired by the syntax and clauses of SQL languages. The main idea behind this DSL is, that declarative queries such as **`SELECT`** `s1 s2` **`FROM`** `tb.dut.core1` can be used to get signal values from a waveform, without having to write the WAL code that performs the required operations.

```
(defmacro apb-print [g]
  `(monitor-apb ,g 0 0
     (lambda [addr data]
       (printf "%4d: R(%s)=%s\n" TS addr data))
     (lambda [addr data]
       (printf "%4d: W(%s)=%s\n" TS addr data))))


(defun apb-restore [g]
  (define mem (array))
  (monitor-apb g 0 0 0
    (lambda [addr data] (seta mem addr data)))
  mem)

(defun apb-avg-delay [g]
  (define start TS)
  (define delays '())
  (defun on-setup [] (set [start TS]))
  (defun on-enable [addr data]
    (set [delays (append delays (- TS start))]))

  (monitor-apb g
               on-setup
               0
               on-enable
               on-enable)
  (average delays))
```

Listing 16: APB callback functions.

Select queries support the clauses **WHERE**, **ON**, and **LIMIT**. These clauses are parsed by the *SELECT* macro and get expanded into the required WAL code, thus hiding the complexity of writing the underlying WAL program to the user.

The code of the *SELECT* macro is shown in Listing 17. First, after the select clause, the macro reads the signals which should be contained in the result until another clause is detected (Lines 4-6). All clauses except the **SELECT** clause are optional and are initialized with sensible default values (e.g., when the **LIMIT** clause is omitted the number of rows returned is not limited) on Lines 8-11. The `resolve-signals` function is a helper function that takes a WAL expression and rewrites it such that all selected signals are resolved in the specified group. Then, `signaltuples`, a list of tuples containing the name and the value of all selected signals, is constructed. Afterward, the rest of the query is parsed and the remaining default clause are overwritten if they are specified (Lines 26-34). Finally, the code that implements the selection query is assembled (Lines 36-44) using the clause values parsed before.

Listing 18 demonstrates how the *SELECT* macro can be used to quickly get the values of some signals. In particular, the macro allows collecting quite complex data with very little code. In addition, the *SELECT* expression is much less deeply nested than the code produced by the macro and therefore much easier to write, especially in an interactive context like the WAL shell. `SHOW` is an utility function defined in the *SELECT* library which prints the result of a query as an ASCII table.

In general, the *SELECT* macro is only one example for a WAL DSL. Other examples could include a DSL that encodes processor instructions or one that encodes network packets. Depending on and utilizing the development context, custom DSLs can simplify expressing problems significantly.

```
 1 (defmacro SELECT args
 2   (define signals '())
 3   (while (&& args
 4              (! (in args[0] '(FROM WHERE ON LIMIT))))
 5     (set [signals (append signals args[0])])
 6     (set [args (rest args)]))
 7
 8   (define group "")
 9   (define condition #t)
10   (define sampling #t)
11   (define limit #t)
12
13   (defun resolve-signals [e]
14     (cond [(&& (list? e) (> (length e) 1))
15             `(,e[0] ,@(map resolve-signals (rest e)))]
16           [(&& (symbol? e)
17                (in e signals)
18             (! (in e '(TS INDEX SIGNALS CS CG))))
19             `(resolve-group ,e)]
20           [else e]))
21
22   (define signaltuples
23           (map (lambda [x] `(',x ,(resolve-signals x)))
24           signals))
25
26   (while args
27     (case args[0]
28      [FROM  (set [group args[1]])]
29      [WHERE (set [condition (resolve-signals args[1])])]
30      [ON    (set [sampling (resolve-signals args[1])])]
31      [LIMIT (set [limit
32                   (resolve-signals
33                     `(< (length result) ,args[1]))])])
34     (set [args (rest (rest args))]))
35
36  `(let ([result '()]
37         [default-sampling (find #t)])
38     (in-group ',group
39       (sample-at (find ,sampling))
40       (whenever (&& ,condition ,limit)
41         (set [result
42               (append result (array ,@signaltuples))]))
43       (sample-at default-sampling))
44      result))
```

Listing 17: Macro for the query language.

```
>-> (SHOW (SELECT INDEX counter
           FROM tb.dut.
           WHERE (> counter 3)
           ON #clk
           LIMIT 5))
+-----+-------+---------+
| Row | INDEX | counter |
+-----+-------+---------+
|   0 |    17 |       4 |
|   1 |    18 |       4 |
|   2 |    19 |       5 |
|   3 |    20 |       5 |
|   4 |    21 |       6 |
+-----+-------+---------+
```

Listing 18: Using the *SELECT* macro inside the WAL shell (>-> is the WAL shell prompt).

VII. Limitations and Future Work

Currently, WAL is not well integrated with established verification and analysis frameworks such as UVM. Utilizing WAL thus requires learning a significantly different new language to either SystemVerilog or VHDL. However, in future work we plan to develop a SystemVerilog bridge which will allow verification engineers to integrate WAL into existing or new verification setups.

Further, the current WAL reference implementation is not targeted at production size workloads. Developing an efficient interpreter targeted at waveform analysis opens up a range of interesting research questions, including optimizations and data structures, which we will address in future work.

VIII. Conclusion

We proposed WAL, a novel domain specific language for non-trivial automated waveform analysis. WAL is specifically designed for waveform analysis, featuring a wide set of functionalities that allow solving waveform analysis problems in a natural and expressive way. With a macro system, a standard library of commonly used functions, and the flexibility of the S-expression syntax, it is straightforward to adapt WAL to many use cases. We have demonstrated the capabilities of WAL for design understanding and debugging in four case studies: First, we analyzed performance metrics of several RISC-V processors. Next, we profiled a RISC-V binary on a waveform. Then, we presented how APB bus interfaces can be analyzed using a flexible and reusable monitoring function. Finally, we presented how WAL can be extended with an embedded DSL by presenting an SQL inspired query language.

Acknowledgments

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

References

[1] X. Lai, A. Balakrishnan, T. Lange, M. Jenihhin, T. Ghasempouri, J. Raik, and D. Alexandrescu, "Understanding multidimensional verification: Where functional meets non-functional," *Microprocessors and Microsystems*, vol. 71, p. 102867, 2019.

[2] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[3] H. D. Foster, "Trends in functional verification: a 2014 industry study," in *Design Automation Conf.*, 2015, pp. 48:1–48:6.

[4] B. Bailey, "Can debug be tamed?" https://semiengineering.com/bigger-debug-challenges-ahead, 2019.

[5] J. Bergeron, *Writing Testbenches Using SystemVerilog*. Springer, 2006.

[6] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer Science & Business Media, 2004.

[7] A. B. Mehta, *SystemVerilog Assertions and Functional Coverage*. Springer, 2019.

[8] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010.

[9] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, Part I," *Commun. ACM*, vol. 3, no. 4, p. 184–195, Apr. 1960.

[10] L. Klemmer and D. Große, "WAL: a novel waveform analysis language for advanced design understanding and debugging," in *ASP Design Automation Conf.*, 2022, pp. 358–364.

[11] ——, "Waveform-based performance analysis of RISC-V processors: late breaking results," in *Design Automation Conf.*, 2022, pp. 1404–1405.

[12] L. Klemmer, E. Jentzsch, and D. Große, "Programmable analysis of RISC-V processor simulations using WAL," in *Design and Verification Conference and Exhibition Europe*, 2022.

[13] S. Ray, I. G. Harris, G. Fey, and M. Soeken, "Multilevel design understanding: from specification to logic," in *International Conference on Computer-Aided Design*, 2016.

[14] J. Zhao and I. G. Harris, "Automatic assertion generation from natural language specifications using subtree analysis," in *Design, Automation and Test in Europe*, 2019, pp. 598–601.

[15] M. Michael, D. Große, and R. Drechsler, "Localizing features of ESL models for design understanding," in *Forum on Specification and Design Languages*, 2012, pp. 120–125.

[16] J. Malburg, A. Finder, and G. Fey, "A simulation-based approach for automated feature localization," *IEEE Transactions on Computer Aided Design of Circuits and Systems*, vol. 33, no. 12, pp. 1886–1899, 2014.

[17] S. Vasudevan, D. Sheridan, S. J. Patel, D. Tcheng, W. Tuohy, and D. R. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *Design, Automation and Test in Europe*, 2010, pp. 626–629.

[18] L. Schammer, J. Runge, P. Klimach, and G. Fey, "Design understanding: Identifying instruction pipelines in hardware designs," in *International Conference on Modern Circuits and Systems Technologies*, 2022, pp. 1–6.

[19] A. Gascón, P. Subramanyan, B. Dutertre, A. Tiwari, D. Jovanović, and S. Malik, "Template-based circuit understanding," in *Int'l Conf. on Formal Methods in CAD*, 2014, pp. 83–90.

[20] A. Mahzoon, D. Große, and R. Drechsler, "RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers," in *Design Automation Conf.*, 2019, pp. 185:1–185:6.

[21] M. Soeken, B. Sterin, R. Drechsler, and R. K. Brayton, "Reverse engineering with simulation graphs," in *Int'l Conf. on Formal Methods in CAD*, 2015, pp. 152–159.

[22] *IEEE Standard for Universal Verification Methodology Language Reference Manual*, IEEE Std. 1800.2-2020, 2020.

[23] *IEEE Standard Verilog Hardware Description Language*, IEEE Std. 1364-2001, 2001.

[24] "Racket language extensions," https://docs.racket-lang.org/guide/hash-languages.html, Accessed: 2023-07-03.

[25] "Common lisp hyperspec," http://www.lispworks.com/documentation/lw50/CLHS/Front/Contents.htm, Accessed: 2023-07-03.

[26] "R7rs scheme," https://small.r7rs.org/, Accessed: 2023-07-03.

[27] A. Bybell, "Implementation of an efficient method for digital waveform compression," https://gtkwave.sourceforge.net/gtkwave.pdf, Accessed: 2023-03-27.

[28] Dörflinger *et al.*, "A comparative survey of open-source application-class RISC-V processor implementations," 2021, p. 12–20.

[29] E. Sperling, "Which processor is best?" https://semiengineering.com/which-processor-is-best, 2022.

[30] D. Patterson, J. Bennett, C. G. P. Dabbelt, G. Madhusudan, and T. Mudge, "Embench™: A modern embedded benchmark suite," 2020.

[31] EEMBC, "Coremark," https://www.eembc.org/coremark/, 2022.

[32] "SERV," https://github.com/olofk/serv, Accessed: 2023-07-03.

[33] "IBEX," https://github.com/lowRISC/ibex, Accessed: 2023-07-03.

[34] "VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation," https://github.com/SpinalHDL/VexRiscv, Accessed: 2023-07-03.

[35] "Minres - The Good Core," https://www.minres.com/products/the-good-folk-series/, Accessed: 2023-07-03.

Lucas Klemmer (Studen Member, IEEE) is a PhD student at the Institute for Complex Systems at the Johannes Kepler University in Linz, Austria. His interests include (formal) hardware verification, computer architecture, open-source EDA tools, and novel applications of SMT solvers.

Daniel Große (Senior Member, IEEE) is a full professor at the Johannes Kepler University Linz, Austria, since 2020, where he is the head of the Institute for Complex Systems (ICS) as well as the head of the "LIT Secure and Correct Systems Lab". His current research interests include verification, virtual prototyping, debugging, synthesis and RISC-V. He published over 170 papers in peer-reviewed journals and conferences. He served in program committees of numerous conferences, including ASP-DAC, DAC, DATE, ICCAD, CODES+ISSS, GLSVLSI, FDL, and MEMOCODE. He received best paper awards (FDL 2007, DVCon Europe 2018, ICCAD 2018, FDL 2020 and FDL 2022) as well as business-related awards (IKT Innovativ Award 2013, Weconomy Award 2013, and Embedded Award 2014).