WAL: A Novel <u>W</u>aveform <u>A</u>nalysis <u>L</u>anguage for Advanced Design Understanding and Debugging

Lucas Klemmer   Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
lucas.klemmer@jku.at   daniel.grosse@jku.at

Abstract—**Starting points for design understanding and debugging 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. 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 evaluate WAL in two major case studies. This includes (i) a WAL-based communication analyzer reporting for example throughput or latency of AXI communication and (ii) the tracing of the instruction flow through the pipeline of a RISC-V processor as well as the extraction of software basic blocks via WAWK, which is based on the WAL-IR to make complex waveform analysis as easy as searching in text files.

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, which, broken down, is dominated by debugging [1]. Even worse, debugging is rated as least predictable since it requires a deep design understanding [2]. The cornerstone for design understanding is the waveform, which is generated during simulation of a *Hardware Description Language* (HDL) design. A waveform for a simulation run describes the circuit signals over time together with hierarchy information [3]. In both, the design phase and the verification phase, waveforms are heavily used. Initially, directed test stimuli are created to see that the currently designed hardware 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. Along this highly iterative process, waveforms demonstrating expected behavior or unexpected behavior (in case of a violated assertion) 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**.

While the above mentioned advanced verification techniques have introduced automation, and led to the generation of "better" waveforms (e.g. by employing formal methods, reducing their length, or minimizing the signals involved in a failing trace) there has been almost **no progress for automating the analysis of waveforms** (in related work we discuss this in more detail). The potential for automation becomes clear when looking at typical non-trivial analysis questions raised for waveforms:

- How are the *Finite State Machine* (FSM) states evolving during data processing?
- What is the latency of my bus interfaces?
- What throughput is my bus achieving?
- When is the processor pipeline flushed or stalled during software execution?
- Which software basic blocks 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) [4]. First, we 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]) [5]. S-expressions have three main advantages: an extremely regular syntax based on lists, code (and data) is represented as nested lists, and hence code can be generated in an arbitrary language 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, and (3) provide an *Intermediate Representation* (IR) well-suited as a compilation target from other languages.

We have developed a reference implementation of WAL in form of an interpreter in Python, which we call *WAL core* in the following. Moreover, to demonstrate the quality of WAL as an IR, we have developed a compiler that reads AWK-like programs[3] and maps them onto the WAL-IR. With the WAL core, less than 300 lines of code are needed to implement this new language *Waveform AWK* (WAWK) making complex waveform analysis as easy as searching in text files.

[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.

[3]AWK, originally developed at Bell Labs, is a data-driven language for text file processing.

Overall, WAL enables the user to create intuitive and easy-to-use programs for non-trivial waveform analysis tasks. The user can either write WAL programs directly or generate them dynamically. We consider both scenarios in two major case studies. At first, we present a WAL-based communication analyzer that generates a detailed report for complex AXI communication traced in waveforms. With only a few lines of embedded WAL code, this analyzer is able to generate advanced textual and graphical performance reports of a design to aid developers in debugging and optimization.

In the second case study, we present the aforementioned WAWK language which uses the WAL-IR as a compilation target. Using WAWK, we (i) visualize the flow of instructions through a RISC-V processor pipeline in the form of an interactive HTML website and (ii) extract the basic blocks of software running on the processor.

Both case studies demonstrate, that WAL enables sophisticated waveform analysis in complex scenarios where information is collected, aggregated and suitably presented.

II. Related Work

Design understanding and debugging is an active field of research. Several methods targeting specific problems as well as different abstraction levels have been considered [6]. For example, this includes natural language techniques to derive assertions from specifications [7], feature localization in ESL models [8] or in RTL descriptions [9], assertion mining at RTL [10], and reverse engineering at the gate-level [11], [12]. However, all these solutions focus on dedicated design understanding sub-problems and do not provide a generic user-programmable analysis for waveforms.

In the introduction assertion-based verification [13] has been mentioned. However, assertion-based verification aims to determine whether a 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 also complex signal relations can be caught, but then the user can perform arbitrary actions for all kinds of computations. In principle, a user could perform certain analysis in the testbench of a design. However, such an approach is extremely complex and would require significant effort, quickly reaching the limits of practicability.

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

III. Programmable Waveform Analysis Problem

As already stated in the introduction, after the generation of a waveform, 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. 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 often is reusable across projects.

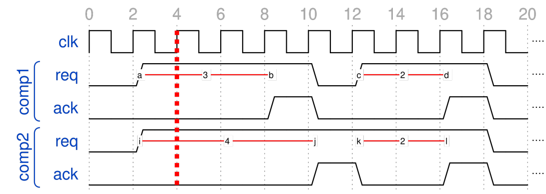


Fig. 1: Request-Acknowledge bus communication

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 with a waveform viewer is practically not possible. More precisely, the task is to walk through the trace and count consecutive requests as long as they are acknowledged. 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 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. Then, we briefly review symbolic expressions as proposed by Lisp (Section IV-B), which we extend by the specifics of symbolic expressions in WAL (Section IV-C). We continue with the essential WAL operations (Section IV-D). Thereafter, the advanced WAL operations are presented (Section IV-E). Finally, we close this section with our reference implementation of WAL which we have realized as an interpreter written in Python, called WAL core.

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 hardware 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 fitting 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 (e.g. [14]).

B. S-Expression Syntax

Symbolic expressions (abbrev. as S-expressions), are common in languages related to Lisp, such as Common Lisp [15] or Scheme [16]. Fundamentally, S-expressions can be of two kinds: *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`.

C. WAL Specific S-Expressions

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.

D. 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).

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 *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 `w` as following: `(load "waveform.vcd" w)`.

After a waveform has been loaded, its time index is set to the beginning at 0, and it is available to WAL expressions. The **`step`** operator can be used to step the time index forwards and backwards 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.

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`).[4]

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.

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 2 we step forward[5] until the end of the waveform is reached. The "core detection" of requests and acknowledgments is performed in Line 3-4. To compute the average latency we have to determine the number of all acknowledged packets. This is done in Line 3, 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 4 the `wait` variable is incremented when the component has a

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

[5]The step size is 2 since we only want to sample data at positive clk edges located on every other index.

pending, unacknowledged request using the other previously introduced expression. Finally, after the end of the waveform has been reached, we calculate the average latency and print it to the standard output in Line 5.

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

Listing 1: Average Latency for comp1

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 an expression and a signed integer, and evaluates the expression with a locally changed time index according to the integer argument. 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 a signed integer to an expression. Using this shorthand syntax, the expression `(reval sig -1)` can be written as `sig@-1`.

We assume the developers need to check a worst-case requirement and therefore have to find pending requests that are persisting at least three consecutive clock cycles. This can be expressed as shown in Listing 2.

```
(&& (&& comp1.req (! comp1.ack))
    (&& comp1.req (! comp1.ack))@2
    (&& comp1.req (! comp1.ack))@4)
```

Listing 2: Detecting continuous pending requests

However, this expression contains a lot of redundancy, which increases the code size and reduces readability. Expressions, which need to be evaluated at multiple time indices, are very common. Hence, we also defined a dedicated shorthand (a summary of all shorthands is listed in Table I); instead of a single number a list of signed integers can stand behind the `@` syntax (row two in Table I).

A better way to write this expression is: `(&& (&& comp1.req (! comp1.ack))@(0 2 4))`.

Here, the condition for a pending request is expanded three times using the `@` syntax. It must be noted that the expansion syntax does not evaluate the expressions after expanding them. Instead, the expressions are passed as arguments to the surrounding call to the `&&` function. This makes the `@` expansion syntax very flexible as it works together with any WAL or user-defined function.

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

TABLE I: Special Shorthand Syntax in WAL

Special Syntax	Transformed into
`expr@sint`	`(reval expr sint)`
expr@(sint$_0$ ... sint$_n$)	expr@sint$_0$ ... expr@sint$_n$
`~symbol`	`(resolve-scope symbol)`
`#symbol`	`(resolve-group symbol)`
`expr[int]`	`(slice expr int)`
expr[int$_0$:int$_1$]	(slice expr int$_0$ int$_1$)

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[6] 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 complete 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

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

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 I).

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 sub modules of the design). Scoping is available via the `~` shorthand and the scoping functions. Since scoping works similar to grouping, we omit further details.

Global Average Latency in WAL: With the advanced WAL functions we can finally generalize the solution to Section III from Listing 1. This allows to calculate the average latency for the complete design (or any other design containing `req-ack` interfaces) with only slight modifications. The resulting WAL program is shown in Listing 3. To enable generalization, the main expression from Listing 1 is wrapped into an `in-groups` function call in Line 2. The `(groups "req" "ack")` function call returns a list of all groups containing `req` and `ack` signals and thus, the expression is evaluated in the groups `"comp1."` and `"comp2."`. Additionally, the *while-step* combination in Line 2 of Listing 1 is changed to a call to `whenever` in Listing 3. The `whenever` function evaluates the second argument at each time index for which the first argument evaluates to true. As we only want to sample data at positive clock edges, we use the condition `clk` (Line 3).

Now the analysis works for all components in the design and, if later more components are attached to the bus, the WAL script automatically works for these added components too.

```
1 (load "waveform.vcd" w)
2 (in-groups (groups "req" "ack")
3   (whenever clk
4        (when (&& #req #ack) (inc packets))
5        (when (&& #req (! #ack)) (inc wait))))
6 (print (/ wait packets))
```

Listing 3: Global Average Latency

F. WAL core

As a reference implementation for WAL we developed 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 employs the Python library vcdvcd [17] for waveform parsing.

Using the API of WAL core, non-trivial analysis tasks can be easily performed in Python applications, which we demonstrate in Section V-A. 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 (cf. Section V-B). Currently, WAL core is not optimized for performance, however as we show in Section V-B it is able to process large waveforms in reasonable time.

V. Case Studies

In this section, we explore three exemplary applications organized in two case studies that demonstrate the capabilities of WAL to perform non-trivial analysis tasks. The case studies were chosen to represent two common utilization scenarios of WAL: in Section V-A, we consider the creation of analysis programs directly in WAL and present an WAL-based AXI communication analyzer. In Section V-B, we describe the dynamic generation of WAL code. As an example, we sketch the language *WAWK* for which we have implemented a compiler that maps AWK-like programs to the WAL-IR. Using WAWK, we perform pipeline instruction flow visualization and basic block detection for the well-known RISC-V processor VexRiscv.

A. WAL-based Communication Analyzer

Advanced bus structures are present in nearly any hardware design. A prominent example is the *Advanced eXtensible Interface* (AXI) as it has been widely adopted and became an industry standard.

In this section, we present a *WAL-based communication analyzer* for AXI4, AXI4-Stream and FSMs. This communication analyzer extracts detailed information about AXI interfaces and FSMs to display them in a *Graphical User Interface* (GUI). The application is not only fully automated, but WALs expressions make it also design agnostic, allowing users to analyze any waveform regardless of signal names, design hierarchy or complexity. While the application GUI has been developed in Python, all waveform analysis is done using WAL and the overwhelming majority (approximately 95%) of code is related to the GUI. All information shown in the GUI is extracted from the waveform data using WAL programs. This leads to a very clean separation of the analysis code from the application and GUI code. In general, for each type of information (such as contained interfaces, latency or FSM graphs) a specific WAL program was written. These programs are typically very concise and are thus directly embedded into the application code as string arguments[7].

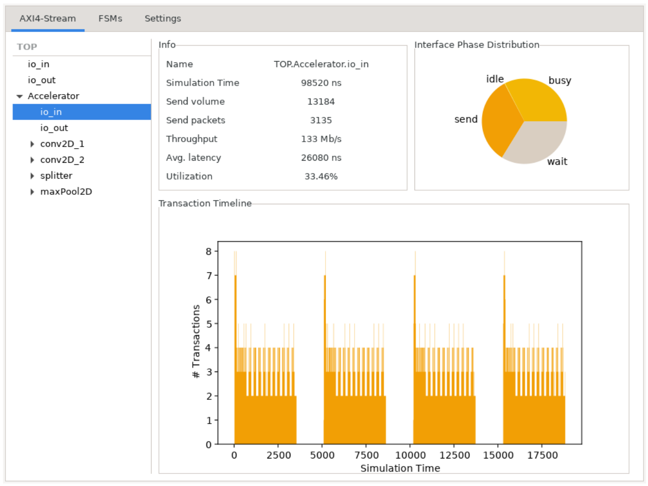
When the user opens a waveform in the WAL-based communication analyzer, it is first scanned for AXI4 or AXI4-Stream interfaces and FSMs. Fig. 2 shows the analysis results for the AI accelerator from [18]. Information about each interface type is divided into multiple tabs. For example, Fig. 2 shows the analysis results of one of the contained AXI4-Stream interfaces of the design. Using WAL, detailed information about the performance (e.g. throughput, avg. delay) of the interface is displayed textually and using graphical elements.

It is virtually impossible to produce the information shown in Fig. 2 manually with a waveform viewer. Implementing this analysis in testbench or design logic is also impractical as it would require very complex logic which would have to be programmed for every analyzed interface and for every analyzed design from scratch. In comparison, our communication analyzer is a one-time effort which is able to analyze every design without additional work.

B. Dynamic WAL-IR Generation from WAWK

In this case study we examine the application of WAL-IR as a compilation target from other languages. For this purpose we present *WAWK*, a new language inspired from the popular text-processing language *AWK*. Using the well-established programming paradigm of AWK, WAWK enables easy and

[7]Similar to the way SQL queries are embedded into other applications.



Name	TOP.Accelerator.io_in
Simulation Time	98520 ns
Send volume	13184
Send packets	3135
Throughput	133 Mb/s
Avg. latency	26080 ns
Utilization	33.46%

Fig. 2: AXI4-Stream Interfaces of AI Accelerator

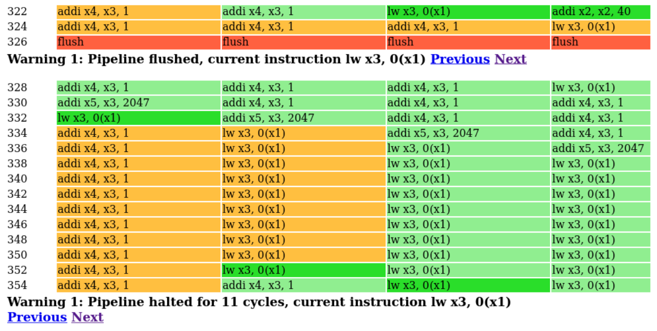
concise scripts for processing waveforms. From the technical viewpoint, we have implemented a compiler that maps WAWK programs to WAL used as IR.

In general, all WAWK scripts consist of multiple statements that follow a `condition: { action }` scheme. For each time index in a waveform, WAWK evaluates the `condition` of each statement, and, if satisfied, executes the associated `action`.

We have implemented the WAWK compiler in Python in less than 300 lines of code. After parsing, the WAWK constructs are mapped onto the WAL-IR. Since the syntax of WAL is based on S-expressions, this mapping is amazingly simple by combining existing WAL functions using the list utilities of Python.

In the following, we consider the VexRiscv processor [19] and demonstrate how advanced waveform analysis is possible with WAWK, i.e. we exemplify visualization of pipeline instruction flow and basic block extraction.

1) RISC-V Pipeline Instruction Flow Visualization: The VexRiscv cores are highly configurable from a bare-bones minimal CPU to a full-fledged Linux capable processor using a very flexible plugin concept offered by SpinalHDL, a high-level HDL implemented in Scala. For a concrete core configuration eventually Verilog is generated. However, this modular structure can make it significantly more challenging to understand the functionality since the one-to-one relation of traditional RTL to the hardware is somewhat "blurred" by the more advanced and abstract concepts of SpinalHDL. While working with a VexRiscv core, we needed a deep insight of the VexRiscv pipeline in order to understand how instructions move through the pipeline and how this effects other parts of the core. Since the relation of complex data is best understood in a structured view and the large amount of data must be manageable, we have created a WAWK script that generates an interactive HTML document. A screenshot of the generated HTML is shown in Fig. 3: Based on the color of the cells, and the assembler code in the cells, we can clearly see the instructions advancing through the pipeline. In addition, we print warnings about exceptional events that occurred during the simulation. For example, a warning is shown whenever the pipeline is halted for more than 5 clock cycles or if the



322	addi x4, x3, 1	addi x4, x3, 1	lw x3, 0(x1)	addi x2, x2, 40
324	addi x4, x3, 1	addi x4, x3, 1	addi x4, x3, 1	lw x3, 0(x1)
326	flush	flush	flush	flush

Warning 1: Pipeline flushed, current instruction lw x3, 0(x1) Previous Next

328	addi x4, x3, 1	addi x4, x3, 1	addi x4, x3, 1	lw x3, 0(x1)
330	addi x5, x3, 2047	addi x4, x3, 1	addi x4, x3, 1	addi x4, x3, 1
332	lw x3, 0(x1)	addi x5, x3, 2047	addi x4, x3, 1	addi x4, x3, 1
334	addi x4, x3, 1	lw x3, 0(x1)	addi x5, x3, 2047	addi x4, x3, 1
336	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	addi x5, x3, 2047
338	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
340	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
342	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
344	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
346	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
348	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
350	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
352	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)	lw x3, 0(x1)
354	addi x4, x3, 1	addi x4, x3, 1	lw x3, 0(x1)	lw x3, 0(x1)

Warning 1: Pipeline halted for 11 cycles, current instruction lw x3, 0(x1)
Previous Next

Fig. 3: Pipeline Instruction Flow Visualization

complete pipeline is flushed. Adding a new case in which a warning is displayed is as easy as detecting the event and calling a function from the script.

Using WAWK, both, the data extraction and the generation of the HTML code required only about 50 lines of WAWK code. The decoding of RISC-V instructions is handled via the external "riscvmodel" [20] Python package. With the help of just 8 lines of Python code our WAWK script is able to profit from the large Python ecosystem.

2) RISC-V Basic Block Extraction: For code analysis, we had to find the basic blocks of software running on the VexRiscv processor. A basic block is a sequence of continuous instructions that will always execute in the given order with no jumps in or out. We collected this information using a WAWK script that traces instructions and detects the beginnings of new blocks and the transitions between those blocks. The script (Listing 4) is composed of four statements.

Initially, *Stmt 1* is executed once before the waveform is processed, which is specified by the special `BEGIN` condition. There, we define some shorter names for the most used signals using the **`alias`** function.

In *Stmt 2*, the *pc* of the first block is stored in the variable *bstart* when *clk* and *fire* are set to 1 and *bstart* is 0 (which is true at the beginning since all variables are initialized to 0 at first use).

Stmt 3 handles detecting the starts of new basic blocks. A new basic block starts if one of the following conditions is met: (1) the last instruction was a jump instruction or branch instruction. Since, the RISC-V ISA [21] encodes jumps and branches by setting the bits 6 and 5 of an instruction to 1, we test this condition in Line 13 and store the result in the variable `was_jump`. (2) the sequential execution of instruction reaches an instruction that is already contained in a block. This is the case when `starts[pc]` is not 0 and the start of the current block is not the same as the block of the previous instruction (i.e. the instruction at `pc - 4` assuming an RV32 ISA). We test this condition and store it in the variable `run_into` in Line 14. In Line 15 we check if one of the above conditions is true, and if satisfied store a new block and block transition. A new transition from the previous block into the current block is stored in the associative array `trans` in Line 16. In Line 17-18 we store the previous instruction as the end of a block in the associative array `ends` and set the current `pc` as the start of the current block. Next, in Line 21-24 we mark the current instruction as belonging to the current block and save the current `pc` and `inst` for the next instruction.

```
// Stmt 1: create aliases
BEGIN: {
  alias(fire, TOP.VexRiscv.lastStageIsFiring);
  alias(pc, TOP.VexRiscv.lastStagePc);
  alias(inst, TOP.VexRiscv.lastStageInstruction);
}

// Stmt 2: initialize pc
TOP.clk, fire, !bstart: { bstart = pc; }

// Stmt 3: detect block boundaries
TOP.clk, fire: {
  was_jump = (lasti[6:5] == 3);
  run_into = (starts[pc] && (starts[pc]!=starts[last]));
  if (was_jump || run_into) {
    trans[last, pc] = [last, pc];
    ends[last] = last;
    bstart = pc;
  };
  // save start of block
  starts[pc] = bstart;
  // save current instruction
  last = pc;
  lasti = inst;
}
// Stmt 4: render graphviz file
END: { ...
}
```

Listing 4: Extracting and visualizing software basic blocks

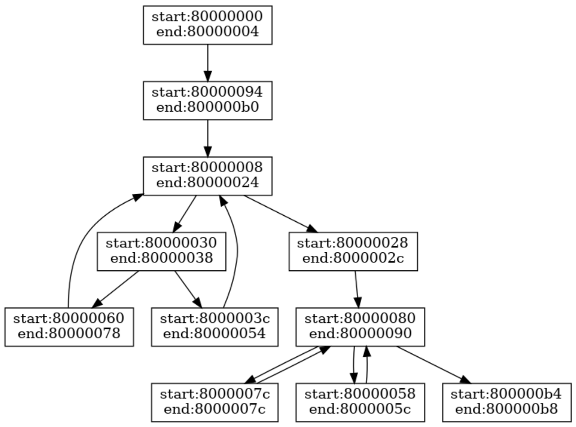

start:80000000 end:80000004
start:80000094 end:800000b0
start:80000008 end:80000024
start:80000030 end:80000038
start:80000028 end:8000002c
start:80000060 end:80000078
start:8000003c end:80000054
start:80000080 end:80000090
start:8000007c end:8000007c
start:80000058 end:8000005c
start:800000b4 end:800000b8

Fig. 4: Basic blocks for GCD calculation implemented in C

Finally, *Stmt 4* generates and prints the graphviz code to render the basic blocks and the control flow of the software. The condition of *Stmt 3* consists of the special symbol END and thus is only executed once after the waveform was completely processed. For brevity, we left out the body of *Stmt 3* since it only contains some loops and print statements to render the graphviz file.

As an example, we analyzed the basic block of a *Greatest Common Divisor* (GCD) implementation in C. The resulting basic block graph is depicted in Fig. 4. The WAWK script needed less than a second for processing the 4 MB large waveform file. To check how the script scales with larger waveform files we evaluated waveforms of the Dhrystone benchmark. The analysis of a medium-sized 265 MB waveform needed around 42 seconds and found 207 basic blocks connected by 298 edges. For the second run, we increased the amount of Dhrystone iterations to produce a much larger waveform file, i.e. more than 4 GB. Running the same WAWK script as on the previous case took 586 seconds and thus the run time scaled approximately linear to the waveform size.

To summarize, we have demonstrated the collection, aggregation and suitable result generation for non-trivial waveform analysis tasks via WAWK programs.

VI. Conclusions

We proposed WAL, a novel domain specific language for non-trivial automated waveform analysis. We have demonstrated the capabilities of WAL for design understanding and debugging in two major case studies: First, we implemented a WAL-based communication analyzer that reports throughput, average latency, and FSM statistics for AXI bus communication. Second, we implemented WAWK on top of WAL, i.e. we developed a compiler that maps AWK-like programs to the WAL-IR. This enabled pipeline instruction flow visualization and basic block detection for the well-known RISC-V processor VexRiscv via WAWK programs making waveform analysis as easy as searching in text files.

Acknowledgments

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

References

[1] H. D. Foster, “Trends in functional verification: a 2014 industry study,” in *DAC*, 2015, pp. 48:1–48:6.
[2] B. Bailey, “Can debug be tamed?” https://semiengineering.com/bigger-debug-challenges-ahead, 2019.
[3] J. Bergeron, *Writing Testbenches Using SystemVerilog*. Springer, 2006.
[4] M. Fowler, *Domain Specific Languages*. Addison-Wesley Professional, 2010.
[5] 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.
[6] S. Ray, I. G. Harris, G. Fey, and M. Soeken, “Multilevel design understanding: from specification to logic,” in *ICCAD*, 2016.
[7] J. Zhao and I. G. Harris, “Automatic assertion generation from natural language specifications using subtree analysis,” in *DATE*, 2019, pp. 598–601.
[8] M. Michael, D. Große, and R. Drechsler, “Localizing features of ESL models for design understanding,” in *FDL*, 2012, pp. 120–125.
[9] J. Malburg, A. Finder, and G. Fey, “A simulation-based approach for automated feature localization,” *TCAD*, vol. 33, no. 12, pp. 1886–1899, 2014.
[10] 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 *DATE*, 2010, pp. 626–629.
[11] A. Mahzoon, D. Große, and R. Drechsler, “RevSCA: Using reverse engineering to bring light into backward rewriting for big and dirty multipliers,” in *DAC*, 2019, pp. 185:1–185:6.
[12] M. Soeken, B. Sterin, R. Drechsler, and R. K. Brayton, “Reverse engineering with simulation graphs,” in *FMCAD*, 2015, pp. 152–159.
[13] H. Foster, A. Krolnik, and D. Lacey, *Assertion-based design*. Kluwer, 2004.
[14] “Racket language extensions,” https://docs.racket-lang.org/guide/hash-languages.html, Accessed: 2021-07-26.
[15] “Common lisp hyperspec,” http://www.lispworks.com/documentation/lw50/CLHS/Front/Contents.htm, Accessed: 2021-07-026.
[16] “R7rs scheme,” https://small.r7rs.org/, Accessed: 2021-07-26.
[17] “GitHub - Python Verilog value change dump (VCD) parser library,” https://github.com/cirosantilli/vcdvcd, Accessed: 2021-07-25.
[18] L. Klemmer, S. Froehlich, R. Drechsler, and D. Große, “XbNN: Enabling CNNs on edge devices by approximate on-chip dot product encoding,” in *ISCAS*, 2021, pp. 1–5.
[19] “GitHub - VexRiscv: A FPGA friendly 32 bit RISC-V CPU implementation,” https://github.com/SpinalHDL/VexRiscv, Accessed: 2021-07-25.
[20] “riscv-model python package,” https://pypi.org/project/riscv-model, Accessed: 2021-07-25.
[21] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA*, SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019.