2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

Programmable Analysis of RISC-V Processor Simulations using WAL*

Lucas Klemmer, Johannes Kepler University Linz, Institute for Complex Systems, Linz, Austria, (lucas.klemmer@jku.at)

Eyck Jentzsch, MINRES Technologies GmbH, Munich, Germany (eyck@minres.com)

Daniel Große, Johannes Kepler University Linz, Institute for Complex Systems, Linz, Austria, (daniel.grosse@jku.at)

Web: https://wal-lang.org & https://github.com/ics-jku/wal

Abstract*—With RISC-V's growing traction, both researchers and companies race to bring their RISC-V implementations to the public. Here, especially RISC-V's extensibility has created a very diverse ecosystem with RISC-V cores ranging from low power to high performance and superscalar architectures. In this diverse ecosystem, knowing the performance specifications of a RISC-V core is essential for both, designers and users when placing the core on the market or selecting a suitable RISC-V core. In this paper, we demonstrate the use of the open-source domain specific language WAL to analyze performance specifications of multiple configurations of open-source as well as a commercial RISC-V core. WAL programs analyze the cores based on waveforms generated during simulation and thus can easily be integrated into standard development processes. The presented WAL programs are flexible and generic, and can be easily adapted to different RISC-V cores.

***Keywords*—RISC-V; simulation; optimization; debugging; performance; programmable analysis**

I. INTRODUCTION

Recently, RISC-V has been gaining enough traction to become a serious competitor to the few proprietary *Instruction Set Architectures* (ISAs) that dominate the market today. RISC-V is an open and royalty free ISA [1] striving for innovation through collaboration, thus enabling even small companies as well as community projects to develop their own processors which take advantage from RISC-V's permissive license and its extensibility to explore new ideas and markets with often highly specialized hardware.

However, this openness and extensibility of RISC-V brings its own set of challenges, since the sheer 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 [2, 3]. A sophisticated analysis of the cores is needed to obtain relevant performance metrics. Since a wide range of cores has to be evaluated, the analysis solution must satisfy several requirements: (1) the analysis must be powerful enough to cover complex analysis tasks, (2) it must be implementation agnostic and easy to port to new cores, and (3) it must be easy to integrate into existing workflows.

In this paper, we use the **open-source *Waveform Analysis Language* (WAL)** [4] to analyze performance metrics for multiple configurations of an industrial RISC-V core. WAL has been realized as a *Domain Specific Language* (DSL) [5]. The language allows creating **analysis programs using the values from the waveforms** generated during simulation of a RISC-V core **in form of program variables**. Our contributions are flexible WAL

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

programs for different performance metrics including the analysis of complex processor pipelines. The programs can be adapted and used on a wide variety of RISC-V microarchitectures.

Our experimental results demonstrate that the WAL-based analysis can clearly highlight the differences between the analyzed cores and configurations. In addition, we can quantify the performance improvements of different core configurations that can be set by enabling additional features, such as instruction caches, branch prediction, or faster ALU implementations.

II. Related Work

In the context of processor architecture research several processor simulators have been proposed. Prominent examples are gem5 [6], multi2sim [7] or virtual prototypes such as [8, 9] for RISC-V. A complimentary direction are emulators, such as qemu [10] or OVPSim [11]. Both simulators, and emulators can partially be used to calculate (performance) metrics (see e.g. [12]). However, they are not as flexible as the WAL DSL for the problems considered. In WAL, we can create programs and tailor them to the exact needs, such as IPC count, pipeline analysis and more.

III. WAL: The Waveform Analysis Language

In this section we review the core concepts of WAL following [5, 13][1]. In comparison to other programming languages, WAL programs have direct access to all signal values of a waveform. Accessing signals in WAL is similar to accessing variables, with the difference that the value returned depends on the loaded waveform and the time at which the signal is accessed. Consider the waveform in Figure 1. The WAL expression *(&& clk instr_done)* returns true at a given time point in the waveform if and only if the *clk* and *instr_done* signals are both set to 1. In Figure 1, all time points at which the expression evaluates to true are highlighted in green. WAL provides a large collection of functions that can be used to analyze waveforms. For example, the count function can be used to count how many instructions are executed on the waveform with the WAL expression *(count (&& clk instr_done))*.

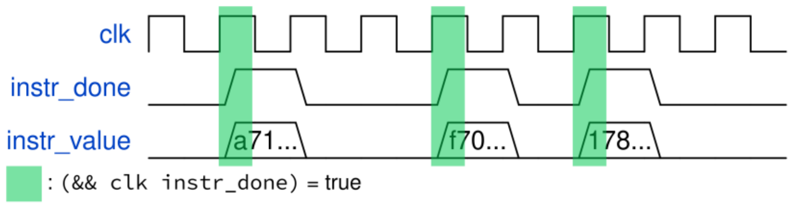

Figure 1: Exemplary Waveform of an Instruction Bus

The **WAL syntax** is based on **Symbolic expressions** (abbrev. as S-expressions), which are common in languages related to Lisp, such as Common Lisp or Scheme. In general, 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*.

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. 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, we introduce some of these functions.

[1] For the Programmer Manual of WAL, we refer the reader to https://wal-lang.org

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. WAL can load waveforms from the industry standard “vcd” format and the faster and more space efficient “fst” format. Assume the waveform data has been dumped to the file *"waveform.vcd"*, as a first step we load this file into WAL under the id that is 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 by 1.

2) Signal Access

After a waveform has been loaded its signal values can be accessed by WAL programs. 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). By accessing signal values just through the name of the signal and the current index of the waveform, WAL programs are very concise and easy to write since virtually no boilerplate code (i.e. getter functions for signal access and bookkeeping of indices) is needed. Signal names can be used inside WAL programs just like variables in other languages. For example, printing a debug message if the signal *top.sub.signal* is not *0* can be written using the following WAL expression: *(if top.sub.signal (print “msg”))*.

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. This can be expressed by appending an @ followed by a signed integer to an expression. Using this shorthand syntax, the value of a signal at the previous index can be accessed by *signal@-1* (this expression will evaluate to the value of *signal* at the time point index – 1) while the value of the same signal but 4 steps ahead can be accessed by *signal@4*. Please note that the @ operator can be applied to every expression and not only to a signal, for example *(+ a b)@2* would evaluate the expression *(+ a b)* at index + 2.

While the focus of this paper is on the analysis of performance metrics for RISC-V cores, we refer the reader to [5] for other applications of WAL, e.g. a WAL-based communication analyzer reporting for example throughput or latency of AXI communication, visualization of the flow of instructions through a RISC-V processor pipeline in the form of an interactive HTML website or the extraction of the basic blocks of software running on the processor.

IV. Experiments

In this section, we present three exemplary processor analysis problems and how they can be analyzed using the waveform analysis language WAL. Initial results of the first two problems have been published in [13]; here we extend them wrt. to several configurations of the commercial RISC-V *The Good Core* (TGC) core from the *The Good Folk Series* (TGFS), which is a family of Generator-Based hardware IP available from MINRES [14] (Section IV.A and Section IV.B). Since the TGC core is highly configurable, we present an extension to determine the execution time of instructions inside the pipeline (Section IV.C).

A. Processor Performance in Instructions per Cycle

In this section, we analyze the raw performance of different RISC-V cores in terms of executed *Instructions Per Cycle* (IPC). Since all analyzed cores are single core architectures, the best theoretical IPC score is 1.0. This means that a 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.

```
1   (defun calc-ipc [traces]
2     (for [trace traces]
3        (load trace t)
4        (setup)
5        (set [instructions (count (&& (is-valid) (instr-done)))])
6        (set [ipc (fdiv 1 (fdiv (count (is-valid)) instructions))])
7        (printf "%40s:␣%15.2f\n" trace ipc)
8        (clean-up)
9        (unload t)))
```

Listing 1: Generic WAL Function for IPC Analysis

```
1   ;; --------------------    General    --------------------
2   (defun setup [(s s) (clk-group clk-group)]
3          (set [s (groups isMoving)]
4               [clk-group (first (groups clk reset lastStageIsFiring))]))
5
6   ;; --------------------      IpC      --------------------
7   (defun is-valid [(clk-group clk-group)]
8          (in-group clk-group (&& (! #clk)@-1 #clk (! #reset))))
9
10  (defun instr-done [(clk-group clk-group)]
11         (in-group clk-group #lastStageIsFiring))
12
13  ;; --------------------   Pipeline   --------------------
14  (defun is-stalled [(s s) (clk-group clk-group)]
15         (in 0 (map (lambda [x] (in-group x #isMoving)) s)))
```

Listing 2: Processor Specific Code for the TGC Processor

The generic WAL program to perform the IPC analysis is shown in Listing 1. The 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 and then to assign the result to the variable instructions via the set function of WAL. 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. First, however, an optional *setup* function is defined in Listing 2 Lines 2-4 which finds all pipeline stages and the location of the global *clk* and *reset* signals. This is done, since different configurations of the core have varying numbers of pipeline stages as well as different locations for the clk and reset signals. Then, in Line 7 the *is-valid* function is implemented. This function should check if the *clk* signal is rising and that *reset* is low. It also is evaluated in a group since, as mentioned before, the actual location of these signals can change. Lines 7-11 show the implementations of these functions for the TGC processor. This processor always sets the *lastStageIsFiring* module to 1 whenever an instruction is completed. Therefore, the *instr-done* function only has to return the value of this signal. After the definition of the required functions, the IPC analysis can be started.

B. *Pipeline Stall Analysis*

In addition to the IPC analysis, we present how the percentage of cycles with stalled pipeline stages can be analyzed with WAL. This metric is useful for example to access how efficient the branch prediction is working. The pipeline stall analysis works similar to the IPC analysis in the sense that users can use a generic function and only have to provide certain processor specific functions themselves. The users have to provide the function *is-valid* and *is-stalled*. Listing 2 also shows the processor specific code required for the pipeline stall analysis on the TGC processor. The *is-stalled* function should return true at each time-point where some part of the pipeline is stalled. Consider the pipeline architecture of the analyzed core: Each stage has an *isMoving* signal which can be used to determine if the stage is currently stalled. We get a list of all groups associated with the pipeline stages in the *setup* function, which is called before the main analysis. The *is-stalled* function is defined in Line 14. This function creates a list with the values of each *isMoving* signal and checks if this list contains a 0 which indicates that this pipeline stage is currently stalled. By using the *in-group* function, we can get the value of all *isMoving* signals even if the actual location of the signal changes or the number of pipeline stages varies.

Table 1 lists the results of the analyzed performance metrics for multiple open-source RISC-V cores and some configurations of the TGC core. It can be seen that TGC, even in the medium sized *3-Stage* configurations, achieves the second-best IPC value of all cores. Also, even the small 3-Stage configuration achieves competitive IPC values. IBEX has to be set to a large cache-enabled configuration to achieve better performance. The commercial core also achieves one of the lowest values for the relative amount of cycles in which at least one of the pipeline stages is stalled.

Table 1: Performance Metrics of RISC-V Cores

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	64%
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%

C. *Execution Time Analysis of Single Instructions*

The highly modular nature of RISC-V is reflected in the extremely high configurability of many RISC-V processors. Many of the available cores today can not only be configured to support a wide range of the RISC-V extensions but they allow a mix and match of nearly every piece of the processor including but not limited to arithmetic components, the number of pipeline stages, different branch prediction implementations, caches and even to completely different bus architectures. Configuring a parameter of a processor can have a significant impact on the performance of the core. Developers might be able to make a rough estimate of this impact however, considering all possible ways these changes can impact the pipeline reliable measurements are needed. For this, only an extensive simulation at the RTL level can be the ground truth.

To get a more accurate understanding of the performance impact of various configurations of the TGC processor we implemented an additional analysis program. This program computes the time each instruction type (e.g. *add*, *mul*, or *srli*) takes on average to be executed. With this program we analyzed performance critical instructions for the six configurations of the TGC processor. Especially arithmetic instructions are suitable for optimization, often at the cost of increased hardware size. In this section, we analyze two arithmetic instructions, *div* and *mul,* from the RISC-V M extension and two shift instructions, *slli* and *srli*.

The *run* function which computes the execution time of instructions is shown in Listing 3. The general idea behind this function is, that instructions are tracked through the pipeline of the processor. Whenever a new instruction enters the pipeline in the first stage (in the following entry stage) the current simulation time is stored in an array (i.e. a hash map). Then, when the instruction leaves the last pipeline stage (in the following exit stage) the execution time is determined using the previously stored start time. However, since instructions can be removed from the pipeline at any stage, we also have to delete instructions from the array if they are removed.

```
1  (defun run [trace entry exit]
2    (load trace t)
3    (set [alive (array)] ;; tracks instructions inside the pipeline
4         [stats (array)] ;; contains a list of lifetimes for each opcode
5         ;; groups for all pipeline stages
6         [stages (groups arbitration_removeIt INSTRUCTION)])
7
8    (whenever (&& clk (! reset))
9        (in-groups stages ;; for each pipeline stage and check instruction status
10           (let ([instr #INSTRUCTION]
11                 [valid #arbitration_isValid])
12
13                ;; when in the first stage and a new instruction enters
14                (if (&& (= CG entry) (!= #INSTRU@<-1 0>) valid)
15                        (seta alive instr INDEX))
16
17                ;; when in last stage and an instruction is done
18                (if (&& (= CG exit) #arbitration_isFiring valid)
19                    ;; calculate how long the instruction was alive
20                    (set [t (/ (- INDEX (geta alive instr)) 2)])
21                    ;; call python code to get the RISC-V instruction opcode
22                    (set [opcode (call extern.decode_opcode instr)])
23                    ;; remove the instruction from list of alive instructions
24                    (dela alive instr)
25                    ;; add lifetime to list of lifetimes for this opcode
26                    (seta stats opcode (+ (geta/default stats '() opcode) t)))
27
28                ;; instruction can be flushed in every stage
29                (when #arbitration_removeIt
30                      ;; remove from alive instructions if flushed
31                      (dela alive instr)))))
32   (unload t))
```

Listing 3: Instruction Execution Time Analysis for TGC

Since the number of pipeline stages varies, the function is written in a generic way. The user has to supply the name of the trace file and the names of the first and last stages to the function (cf. Line 1 in Listing 3). Then, the trace file is loaded (Line 2). After that, the arrays for tracking the instructions (*alive*) and the final results (*stats*) are initialized (Lines 3-4). Also, the groups of the pipeline stages are searched for (each stage has an *arbitration_removeIt* and an *INSTRUCTION* signal) (Line 6). The main loop of the function is evaluated at each index at which the clk is high and reset is low (Line 8). Now, the function goes through each pipeline stage (Line 9) and checks if it is the entry or exit stage (Line 10-31). In each stage, first the current instruction and the validity of this instruction is stored since they are required multiple times in later lines (Lines 10-11). If a new valid instruction enters the entry stage it is added to the *alive* array (Lines 14-15). If an instruction is executed in the exit stage, the execution time is calculated based on the stored start time of this instruction (Line 18-20). Then, the opcode of the instruction is determined with the help of a call to the "riscv-model" Python package (Line 22)[2]. The instruction is removed from the *alive* array (Line 24) and the result is stored to the list in the *stats* array (Line 26). If the RISC-V core removes an instruction from the pipeline (e.g. in case of mistaken branch), we also have to remove it from the *alive* array (Line 29-31). Please not this scenario is possible in each pipeline stage. Finally, the trace is unloaded from WAL (Line 32). Please note, that the code was slightly simplified to better illustrate the idea behind the analysis, also the printing of the result at the end is not shown.

Table 2: Average Execution Time of Instructions (in number of cycles)

Instruction	3-Stage	4-Stage v1	4-Stage v2	4-Stage v3	4-Stage v4
div	6	6	68	68	14
mul	4	4	62	62	7
slli	1	2	12	4	12
srli	1	2	21	5	25

Table 2 shows the resulting execution times for the four analyzed instructions in number of cycles. All waveforms were generated by running the Dhrystone benchmark on the specified core configurations. The first

[2] Please note: WAL allows to interface with other languages. In the example, we use a Python package to decode the RISC-V instruction.

column gives the instruction while the remaining columns show the processor configuration. The data shows, that the runtime of the *div* and *mul* instructions is significantly lower in the configurations 3-Stage and 4-Stage *v1*. The same also applies for the runtime of the shift instructions *slli* and *srli* which is much slower in the configurations *4-Stage v2* and *4-Stage v4*. With these fine-grained measurements of the hardware cost associated with each configuration developers can decide which configuration makes the most sense for a given application and its requirements.

V. CONCLUSIONS

In this paper we calculated relevant performance metrics of several RISC-V cores using WAL, the Waveform Analysis Language. We analyzed the performance of a core in terms of raw numbers of executed instructions per cycle, the frequency of stalled pipeline stages, and the execution time of different instructions. Our experiments show that thorough analyses of RISC-V cores are possible with WAL. Automatic and programmable core analysis makes it possible to identify performance bottlenecks and optimization potential at an early stage. The information gathered by the WAL analysis enables developers to leverage the high configurability of modern processor workflows to create fine-tuned processors that are tailored to the needs of specific applications.

VI. REFERENCES

[1] Andrew Waterman and Krste Asanović. The RISC-V Instruction Set Manual; Volume I: Unprivileged ISA. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, 2019

[2] Alexander Dörflinger et al. A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations. In *CF*, pages 12–20, 2021

[3] Ed Sperling. Which Processor Is Best? https://semiengineering.com/which-processor-is-best. 2022

[4] WAL the Waveform Analysis Language. https://github.com/ics-jku/wal

[5] Lucas Klemmer and Daniel Große. WAL: A Novel Waveform Analysis Language for Advanced Design Understanding and Debugging. In *ASP-DAC*, pages 358–364, 2022.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (May 2011), 1–7. 2011

[7] Rafael Ubal, Julio Sahuquillo, Salvador Petit, and Pedro Lopez. Multi2sim: A simulation framework to evaluate multicore-multithreaded processors. In *SBAC-PAD*, pages 62–68, 2007

[8] Vladimir Herdt, Daniel Große, Hoang M. Le, and Rolf Drechsler. Extensible and configurable RISC-V based virtual prototype. In *FDL*, pages 5–16, 2018.

[9] Vladimir Herdt, Daniel Große, Pascal Pieper, and Rolf Drechsler. RISC-V based virtual prototype: An extensible and configurable platform for the system-level. *JSA*, 109:101756, 2020.

[10] “QEMU a generic and open source machine emulator and virtualizer”, https://www.qemu.org/

[11] “Technology OVPsim”, https://www.ovpworld.org/technology_ovpsim

[12] Vladimir Herdt, Daniel Große, and Rolf Drechsler. Fast and accurate performance evaluation for RISC-V using virtual prototypes. In *DATE*, pages 618–621, 2020.

[13] Lucas Klemmer and Daniel Große. Waveform-based performance analysis of RISC-V processors: late breaking results. In *DAC*, pages 1404–1405, 2022.

[14] MINRES Technologies, https://www.minres.com/