WAL: A Novel Waveform Analysis Language for Advanced Design Understanding and Debugging

Lucas Klemmer, Daniel Große
Institute for Complex Systems (ICS)
Web: jku.at/ics
Email: lucas.klemmer@jku.at

JOHANNES KEPLER UNIVERSITY LINZ

Bio

- Lucas Klemmer
- PhD student at the Institute for Complex Systems
- Johannes Kepler University Linz Austria

- Interests:
  - Verification
  - EDA tools
  - RISC-V

Waveform Debugging: Problem or Solution?

- Waveforms are the fundamental data format for HW development
  - Produced by simulators and formal tools
- Proven technique for design understanding and debugging
- Waveforms contain incredible amounts of information
  - performance, correctness, data/control flow, optimization, …
- Problems
  - 100% manual process
  - Tedious and slow navigation
  - Only small slice of data visible at once
  - Only for “simple” signal relations
  - Cannot be automated

WAL: Waveform Analysis Language

- WAL is *Domain Specific Language* (DSL) to express HW analysis problems
- Specialized language constructs for HW domain:
  - Waveform signals
  - Time
  - Hierarchy (modules, submodules)
  - Signal relations (bus interfaces)
- Not just true/false expressions, much more than SVA, PSL, …
- Full capabilities of scripting languages (functions, external libraries, …)
- Implemented in Python
  - Access to a billion Python packages!

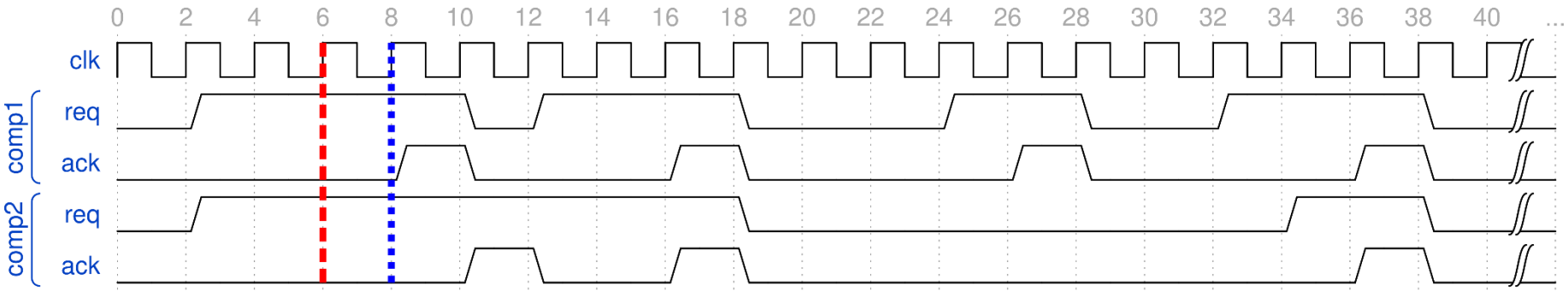
WAL Expressions

- Inspired by LISPs S-Expressions
- WAL programs consist of expressions
  - Constants: `1, „hello, world!“`
    - `1` ⇒ `1`
  - Symbols: `var1, var2`
    - `var1` ⇒ e.g. `5`
  - Lists: `(1 2 3)`
  - Operators: `(+ 1 2 var1)` ⇒ `8`
  - User functions: `(defun foo [bar] …) (foo 5)`
  - Waveform signals: `Top.module.data_o` ⇒ ?

Accessing Waveform Data

- What is the result of evaluating a signal?
- Depending on:
  - Loaded waveform
  - Timepoint in the waveform
- Move index using `(step)` function

```
   (step 6)
6: comp1.ack ⇒ 0
   (step 2)
8: comp1.ack ⇒ 1
```



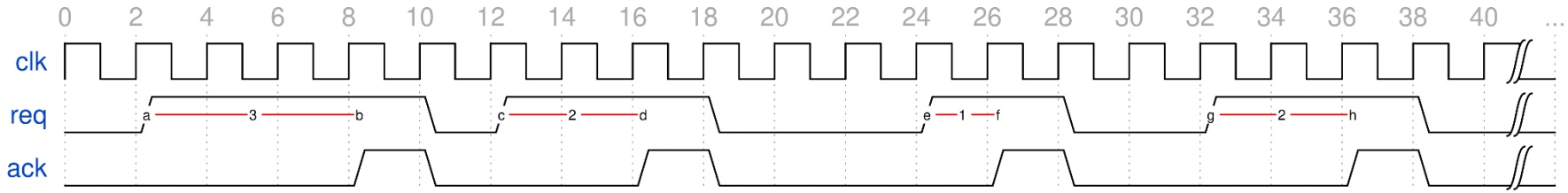
Timed Signal Access

- Time index can be locally modified with `expr@offset` syntax
  - Example: `signal@1` evaluated at INDEX + 1
  - Signal value change: `(!= signal@-1 signal)`
  - Rising clock edge: `(&& (! clk@-1) clk)`
- @ can be applied to every expression (not just signals)
  - Is `data` larger than 5 two indices ahead?: `(> data 5)@2`
- @ can also be used with multiple offsets: `signal@<offset1 offset2 …>`
  - `(&& (> data 5)@<1 2 3 4>)` ⇒ `(&& (> data 5)@1 (> data 5)@2 …)`

Example: Avg. Delay (1)

- Calculate avg. delay on handshaking bus
- Two states:
  - Waiting: `(&& req (! ack))`
  - Sending: `(&& req ack)`
- Count states
- Result = |waiting| / |sending|

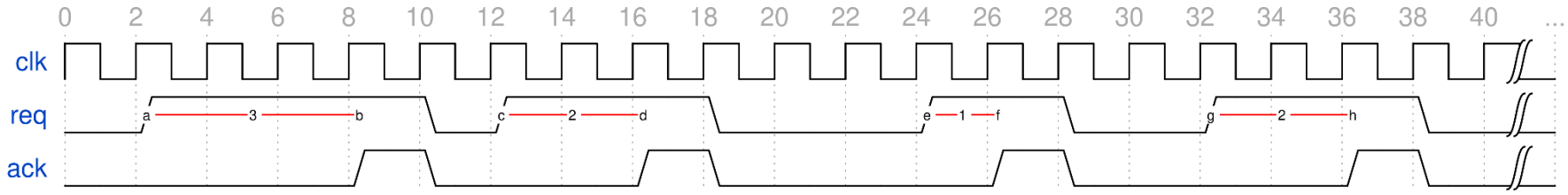
```
(whenever clk
    … always evaluated when clk = 1 …)
```



Example: Avg. Delay (2)

- Calculate avg. delay on handshaking bus
- Two states:
  - Waiting: `(&& req (! ack))`
  - Sending: `(&& req ack)`
- Count states
- Result = |waiting| / |sending|

```
(whenever clk
    (when (&& req (! ack)) (inc wait))
    (when (&& req ack) (inc packets)))

(print (/ wait packets))
```



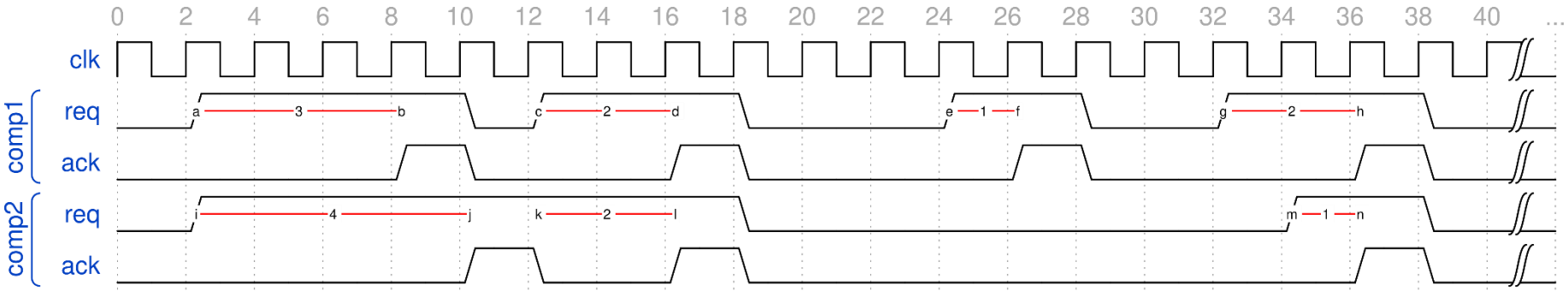
(3+2+1+2) / 4 = 8/4 = 2

Example: Avg. Delay (3)

```
(groups “req” “ack”) ⇒ (comp1. comp2.)
```

- HW designs ideal for writing generic code!
- Generic expressions with groups
- Expression evaluated in each group
- `#signal` expanded to full name

```
(in-groups (groups "req" "ack")
  (whenever clk
    (when (&& #req (! #ack)) (inc wait))
    (when (&& #req #ack) (inc packets))))

(print (/ wait packets))
```



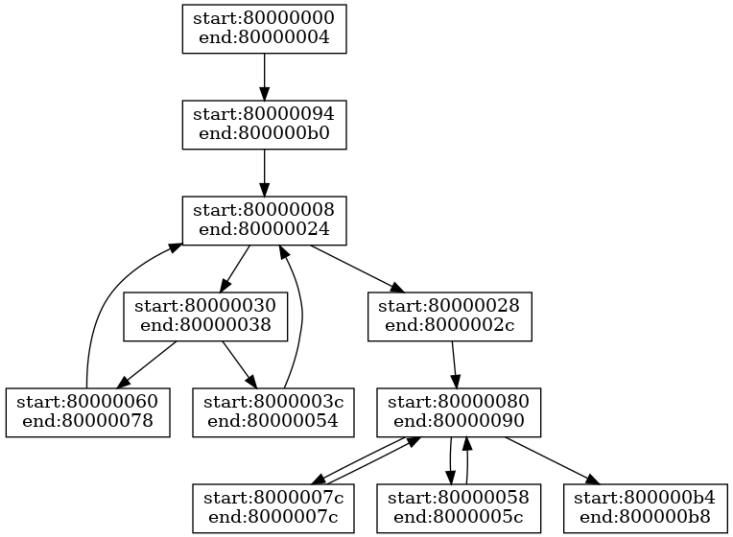
((3+2+1+2) + (4+2+1)) / 7 = (8 + 7) / 7 = 15/7 ≈ 2.1

Other WAL Features

- Data Structures
  - Lists:
    - `(first list)`, `(second list)`, `(rest list)`, …
    - `list[i]`, `list[h:l]`
    - `Fold`, `fold/signal` , `map`, `for` …
  - Hashmaps:
    - (geta symbol key1 key2 …)
    - (seta symbol key1 key2 … data)
- Extracting bits from signals
  - `signal[i]`, `signal[h:l]`
- Finding indices at which condition is true
  - `(find cond)` => `(40 56 102 …)`
- WAL as a compilation target from other languages

Example Applications (1)

Basic block/CFG extraction



VexRiscv Pipeline Visualization

318	lw x3, 0(x1)	addi x2, x2, 40	auipc x2, 1	addi x1, x1, 0
320	addi x4, x3, 1	lw x3, 0(x1)	addi x2, x2, 40	auipc x2, 1
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

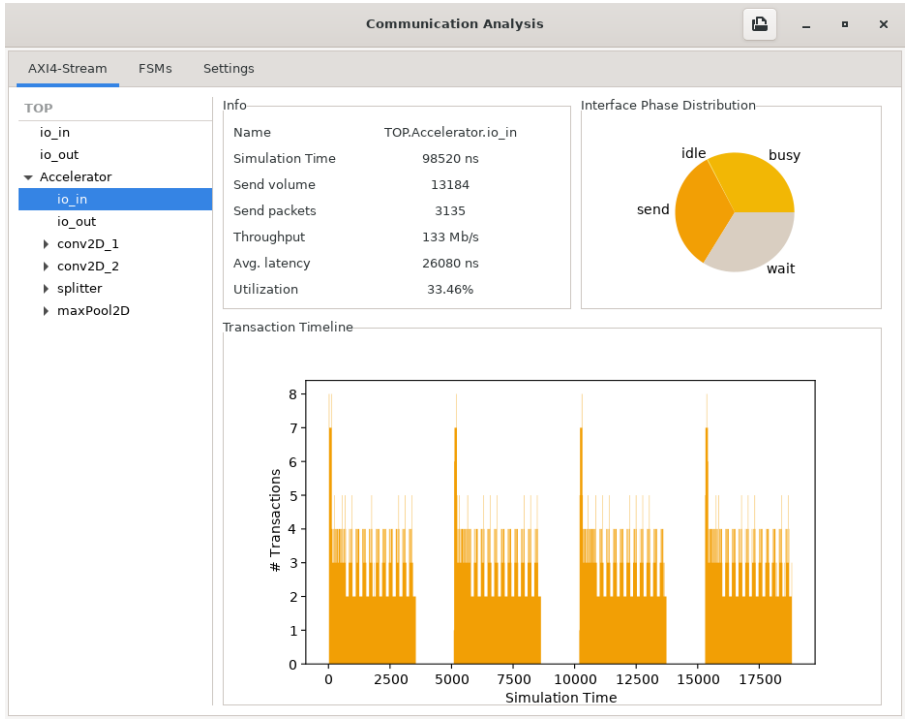
356	addi x4, x3, 1	addi x4, x3, 1	addi x4, x3, 1	lw x3, 0(x1)

Example Applications (2)

Interactive Debugging REPL

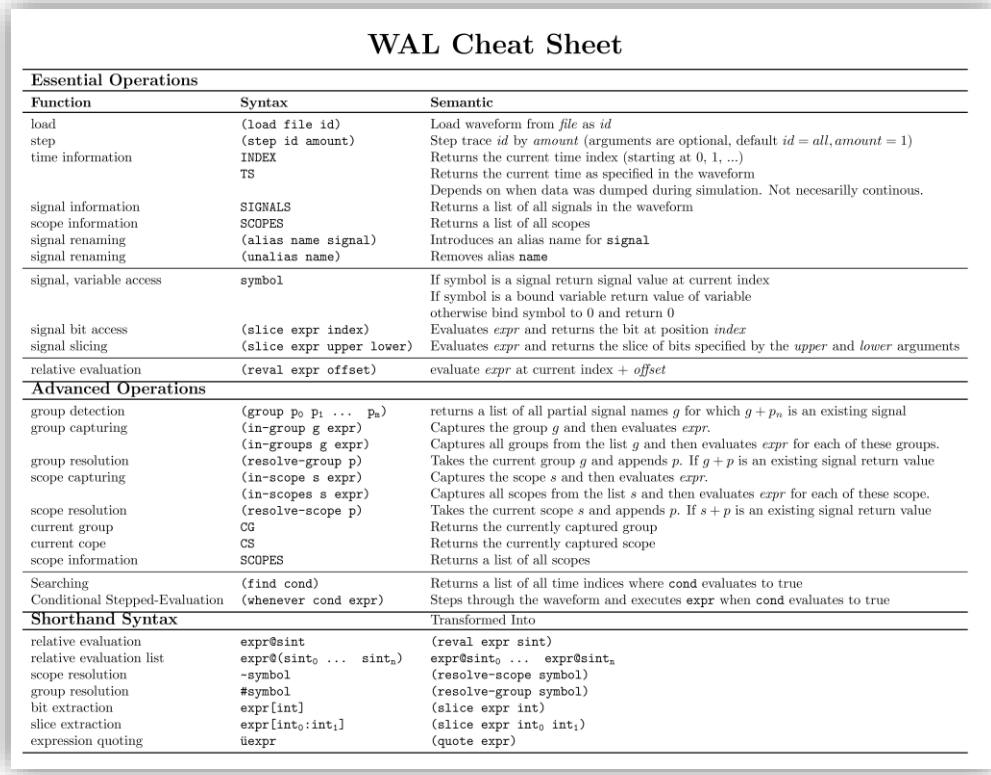
```
uart ) wal
WAL 0.6.0

>-> (load "uart.vcd" uart)
uart >-> (alias clk uart_tx_test.clk)
uart >-> (alias state uart_tx_test.test_uart.uart_rx.sta
te)
uart >-> (find (&& clk (= state@-2 0) (= state 1)))
(1737 11313 20889 30465 40041 49617 59137 68713 78289 87
865 97441 107017 116593 126169 135745 145265)
uart >-> (step 1737)
#t
uart >-> state
1
uart >-> uart_tx_test.test_uart.uart_rx.data
0
uart >->
```

Throughput, Latency on AXI bus



Conclusion

- Waveform viewing is highly manual
- WAL enables programmable waveform analysis
  - Data aggregation
  - Data visualization
  - Complex queries
- WAL availability
  - GitHub: https://github.com/ics-jku/wal
  - Soon also on PyPi



WAL Cheat Sheet

Essential Operations		
Function	**Syntax**	**Semantic**
load	`(load file id)`	Load waveform from *file* as *id*
step	`(step id amount)`	Step trace *id* by *amount* (arguments are optional, default $id = all, amount = 1$)
time information	`INDEX`	Returns the current time index (starting at 0, 1, ...)
	`TS`	Returns the current time as specified in the waveform Depends on when data was dumped during simulation. Not necesarilly continous.
signal information	`SIGNALS`	Returns a list of all signals in the waveform
scope information	`SCOPES`	Returns a list of all scopes
signal renaming	`(alias name signal)`	Introduces an alias name for `signal`
signal renaming	`(unalias name)`	Removes alias `name`
signal, variable access	`symbol`	If symbol is a signal return signal value at current index If symbol is a bound variable return value of variable otherwise bind symbol to 0 and return 0
signal bit access	`(slice expr index)`	Evaluates *expr* and returns the bit at position *index*
signal slicing	`(slice expr upper lower)`	Evaluates *expr* and returns the slice of bits specified by the *upper* and *lower* arguments
relative evaluation	`(reval expr offset)`	evaluate *expr* at current index + *offset*
Advanced Operations		
group detection	`(group` p_0 p_1 ... p_n`)`	returns a list of all partial signal names g for which $g + p_n$ is an existing signal
group capturing	`(in-group g expr)`	Captures the group g and then evaluates *expr*.
	`(in-groups g expr)`	Captures all groups from the list g and then evaluates *expr* for each of these groups.
group resolution	`(resolve-group p)`	Takes the current group g and appends p. If $g + p$ is an existing signal return value
scope capturing	`(in-scope s expr)`	Captures the scope s and then evaluates *expr*.
	`(in-scopes s expr)`	Captures all scopes from the list s and then evaluates *expr* for each of these scope.
scope resolution	`(resolve-scope p)`	Takes the current scope s and appends p. If $s + p$ is an existing signal return value
current group	`CG`	Returns the currently captured group
current cope	`CS`	Returns the currently captured scope
scope information	`SCOPES`	Returns a list of all scopes
Searching	`(find cond)`	Returns a list of all time indices where `cond` evaluates to true
Conditional Stepped-Evaluation	`(whenever cond expr)`	Steps through the waveform and executes `expr` when `cond` evaluates to true
Shorthand Syntax		Transformed Into
relative evaluation	`expr@sint`	`(reval expr sint)`
relative evaluation list	`expr@(`$sint_0$ ... $sint_n$`)`	`expr@`$sint_0$ ... `expr@`$sint_n$
scope resolution	`-symbol`	`(resolve-scope symbol)`
group resolution	`#symbol`	`(resolve-group symbol)`
bit extraction	`expr[int]`	`(slice expr int)`
slice extraction	`expr[`int_0`:`int_1`]`	`(slice expr` int_0 int_1`)`
expression quoting	`üexpr`	`(quote expr)`