Get the Most out of Your Waveforms

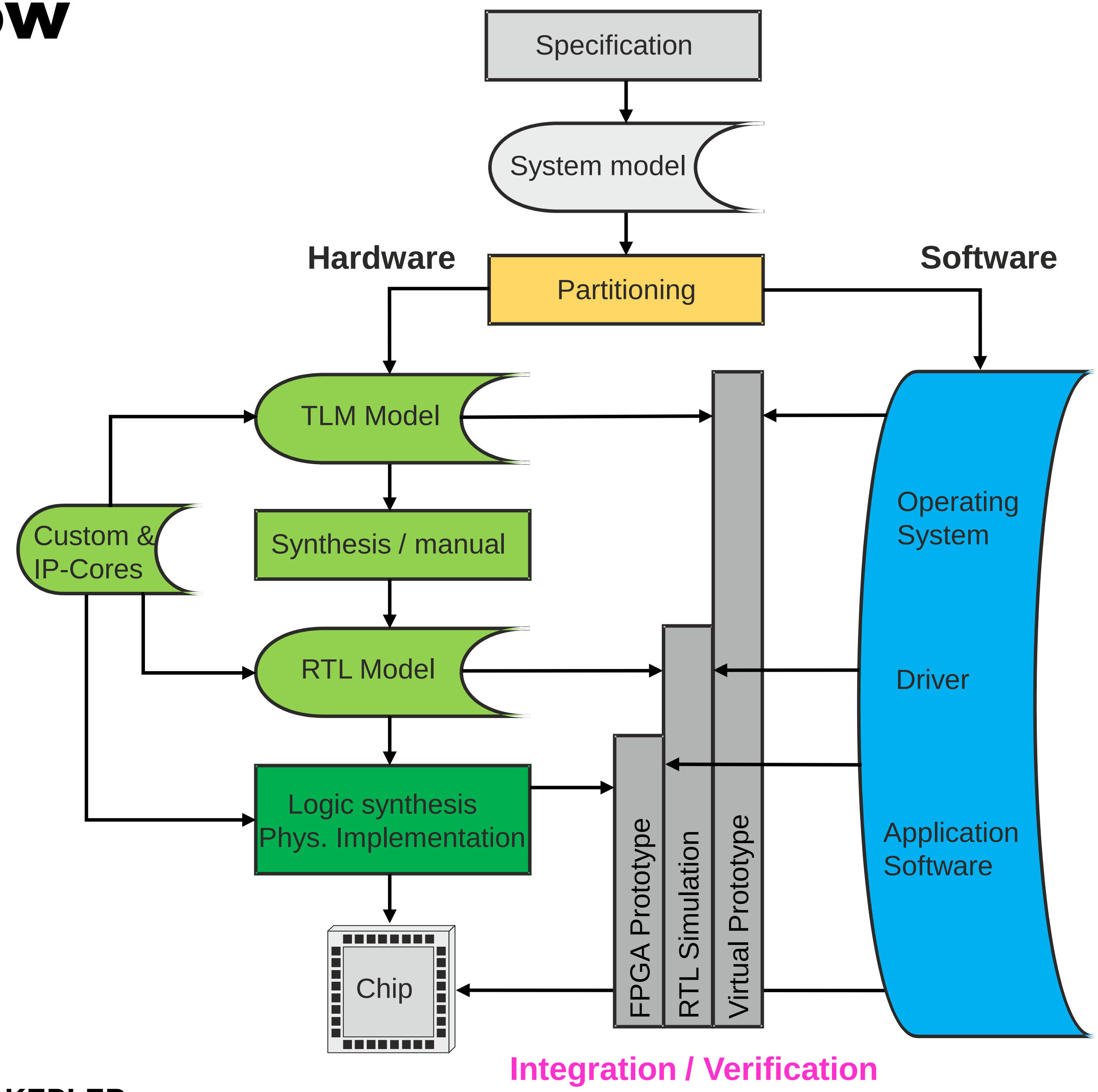
From Non-functional Analysis to Functional Debug via Programs on Waveforms

Daniel Große, Lucas Klemmer
Institute for Complex Systems (ICS)
Web: jku.at/ics   wal-lang.org
Email: daniel.grosse@jku.at, lucas.klemmer@jku.at

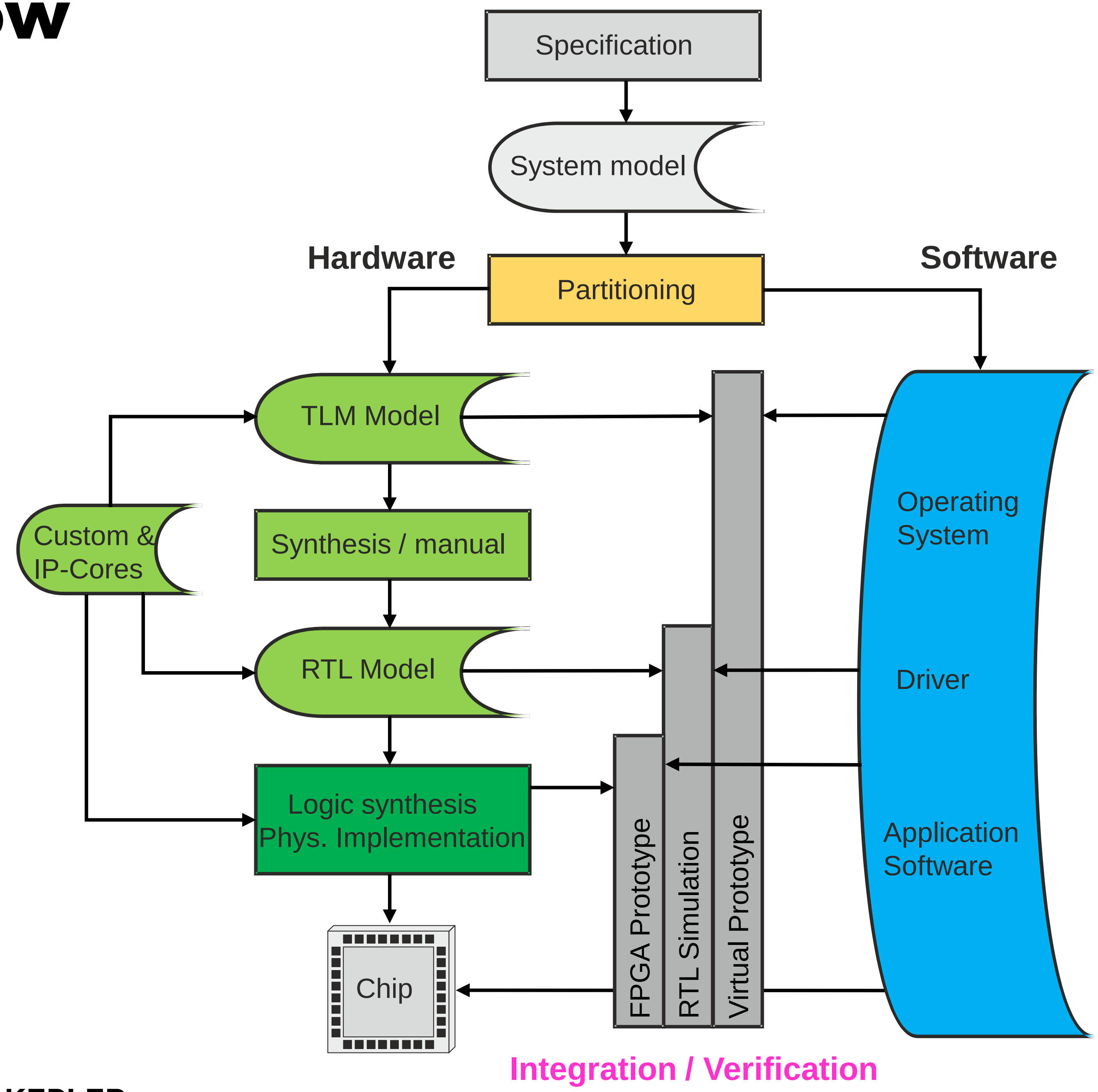
Why?

Get the Most out of Your Waveforms

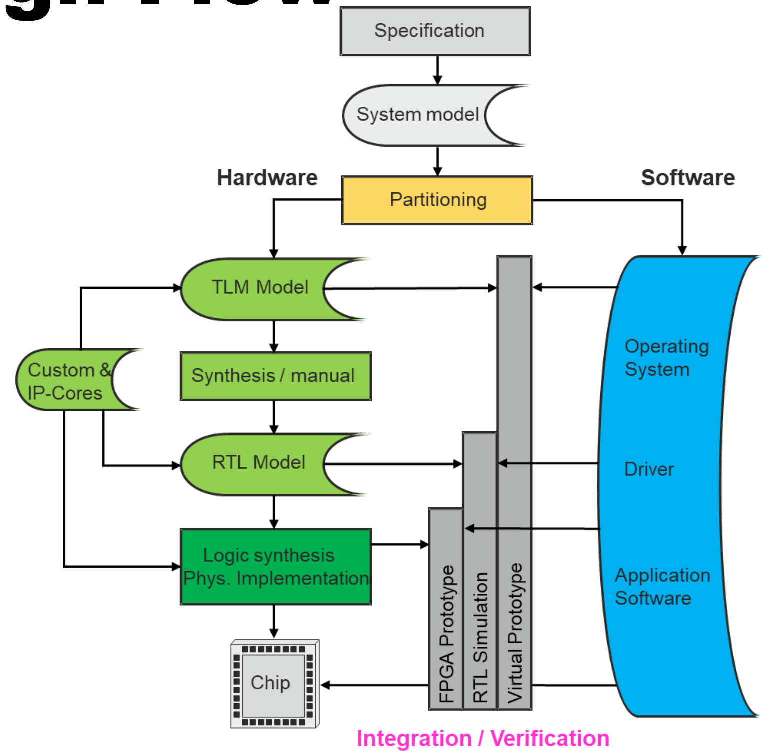
Design Flow


Specification
System model
Hardware
Partitioning
Software
TLM Model
Custom &
IP-Cores
Synthesis / manual
RTL Model
Logic synthesis
Phys. Implementation
Chip
FPGA Prototype
RTL Simulation
Virtual Prototype
Operating
System
Driver
Application
Software
Integration / Verification

Design Flow


Specification
System model
Hardware
Partitioning
Software
TLM Model
Synthesis / manual
Custom &
IP-Cores
RTL Model
Logic synthesis
Phys. Implementation
Chip
FPGA Prototype
RTL Simulation
Virtual Prototype
Operating
System
Driver
Application
Software
Integration / Verification

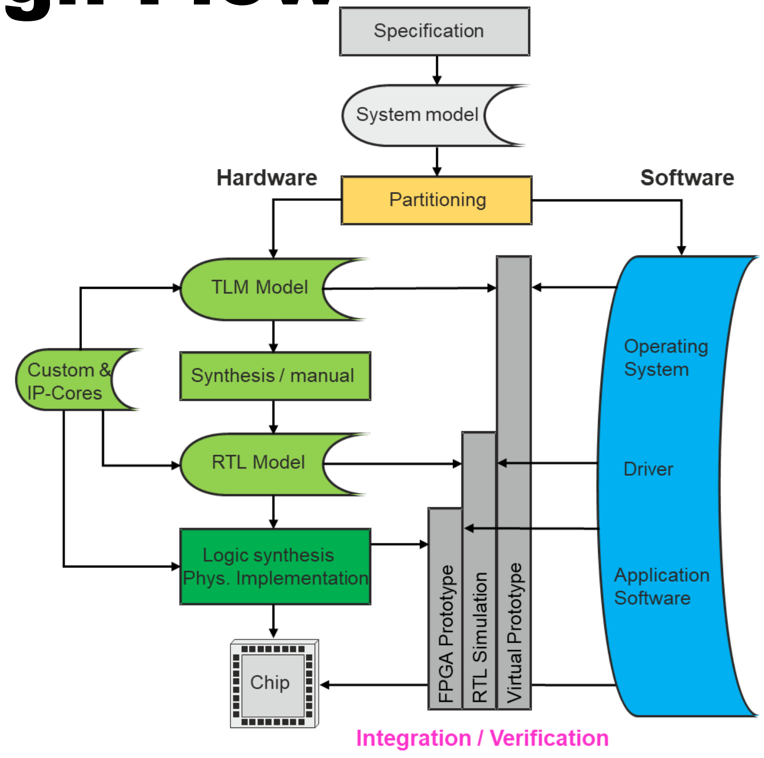
Design Flow

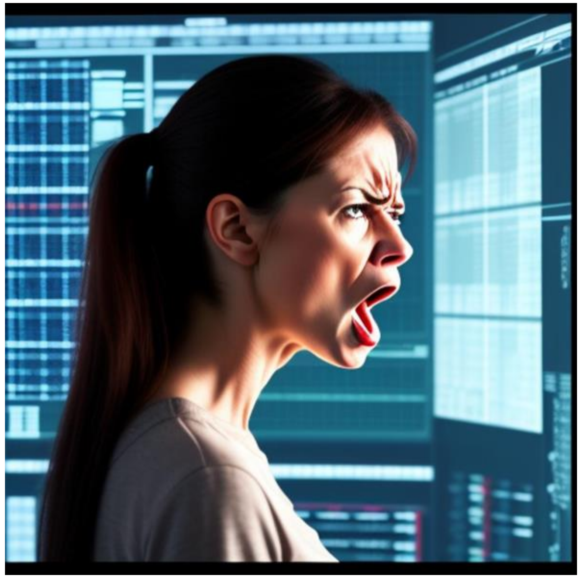


Waveforms

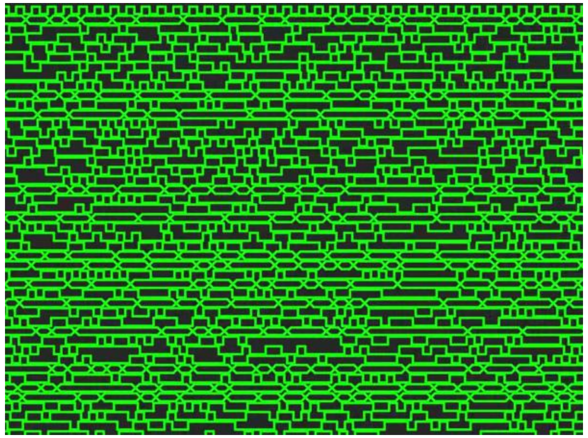
- HW block is alive
- HW shows expected behavior
- Communication works
- Assembler instructions run
- Performance as expected
- …

Design Flow





It's just too much data!

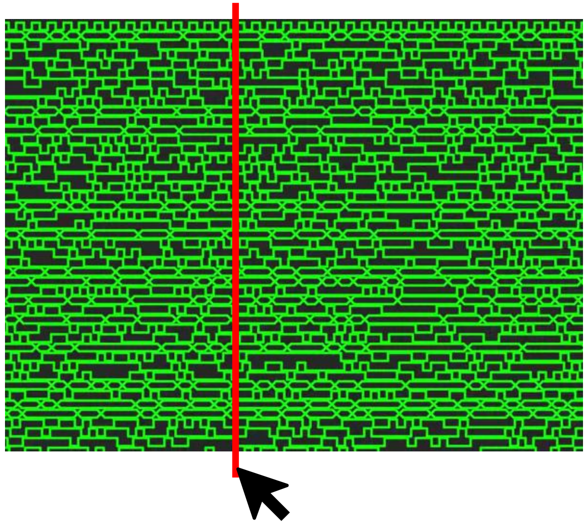


Waveforms

- HW block is alive
- HW shows expected behavior
- Communication works
- Assembler instructions run
- Performance as expected
- …

… Waveforms

- Waveforms are great!
- A central data format for HW development
  - Produced by simulators, formal tools, FPGAs, …
- They contain incredible amounts of information
  - performance, correctness, data/control flow, optimization, …
- However …
  - 100% manual process
  - Tedious and slow navigation
  - Only small slice of data visible at once
  - Only for “simple” signal relations
  - Analysis not 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!

How to Read WAL Expressions

- This is a **number**
  - 5
- These are also numbers
  - 0xff, 0b1101
- This is a **symbol**
  - my_var
- And these are also symbols
  - RD-START, top.core1.run
- This is a **string**
  - “hello, FDL!”

- The same in Python
  - 5
  - 0xff, 0b1101
  - my_var
  - RD-START, top.core1.run?
  - “hello, FDL!”

How to Read WAL Expressions (2)

- This is a **list**
  - `(5 1 abc)`
- If the first element is a function name the list is a function application
  - `(+ 1 2)`
  - `(+ 1 2 3 …)`
  - `(print “hello”)`
  - `(print “Sum: “ (+ 1 2))`

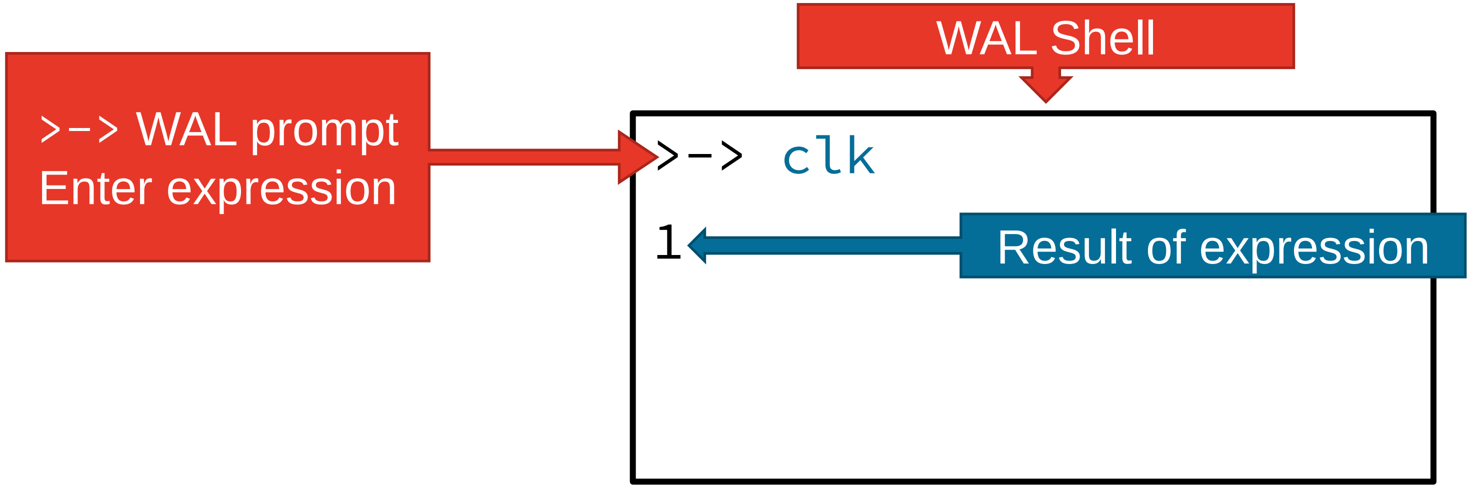
- The same in Python
  - `[5, 1, abc]`
  - `1 + 2`
  - `1 + 2 + 3 + . + ..`
  - `print(“hello”)`
  - `print(“Sum: “, 1 + 2)`

Arithmetic and Logic Operators

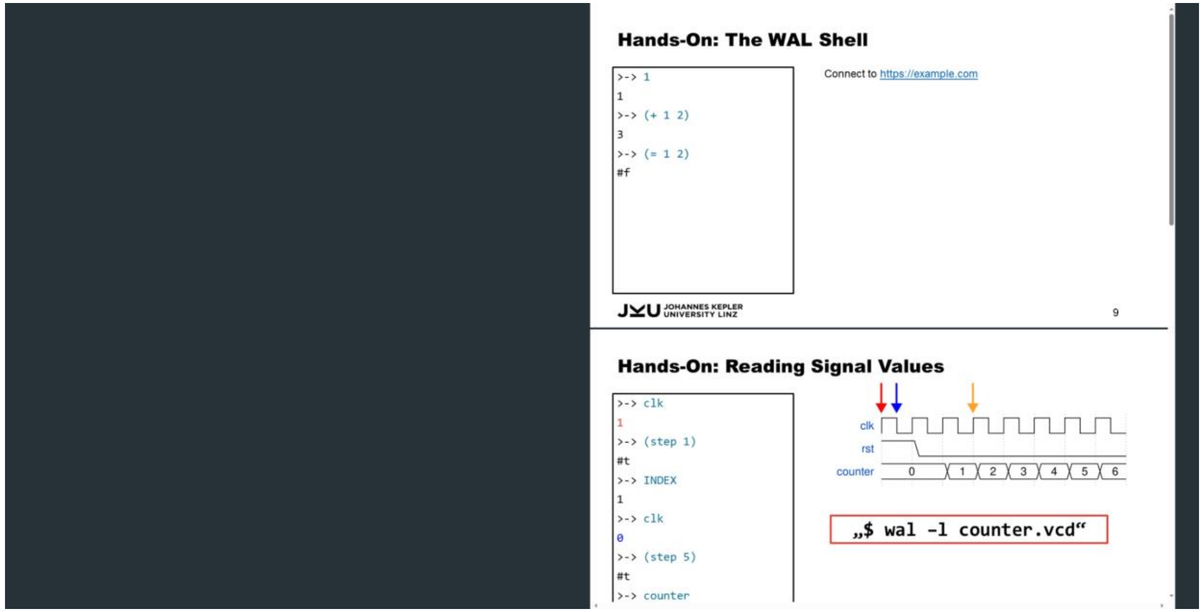
- Arithmetic Operators
  - `+`, `-`, `*`, `/`
  - `(+ 1 2)` => 3
  - `(+ 1 (- 4 2))` => 3

- Logic Operators
  - `!`, `&&`, `||`, `=`, `!=`, `>`, `<`, `>=`, `<=`
  - `(&& #t #t)` => #t
  - `(! (&& #t #t))` => #f
  - `(> 5 4)` => #t

Hands-On


WAL Shell
>-> WAL prompt
Enter expression
>-> clk
1
Result of expression

Hands-On: FDL Tutorial Website



Visit: Oh no, this link was only available to people at FDL ☹

- Left side
  - Linux environment
  - Nano, vim
  - Everybody has their own instance
  - Deleted when page is closed

- Right side
  - The Tutorial slides

Hands-On: The WAL Shell

```
user@e25a:~$ wal -l fdl.vcd
>-> 1
1
>-> (+ 1 2)
3
>-> (= 1 2)
#f
```

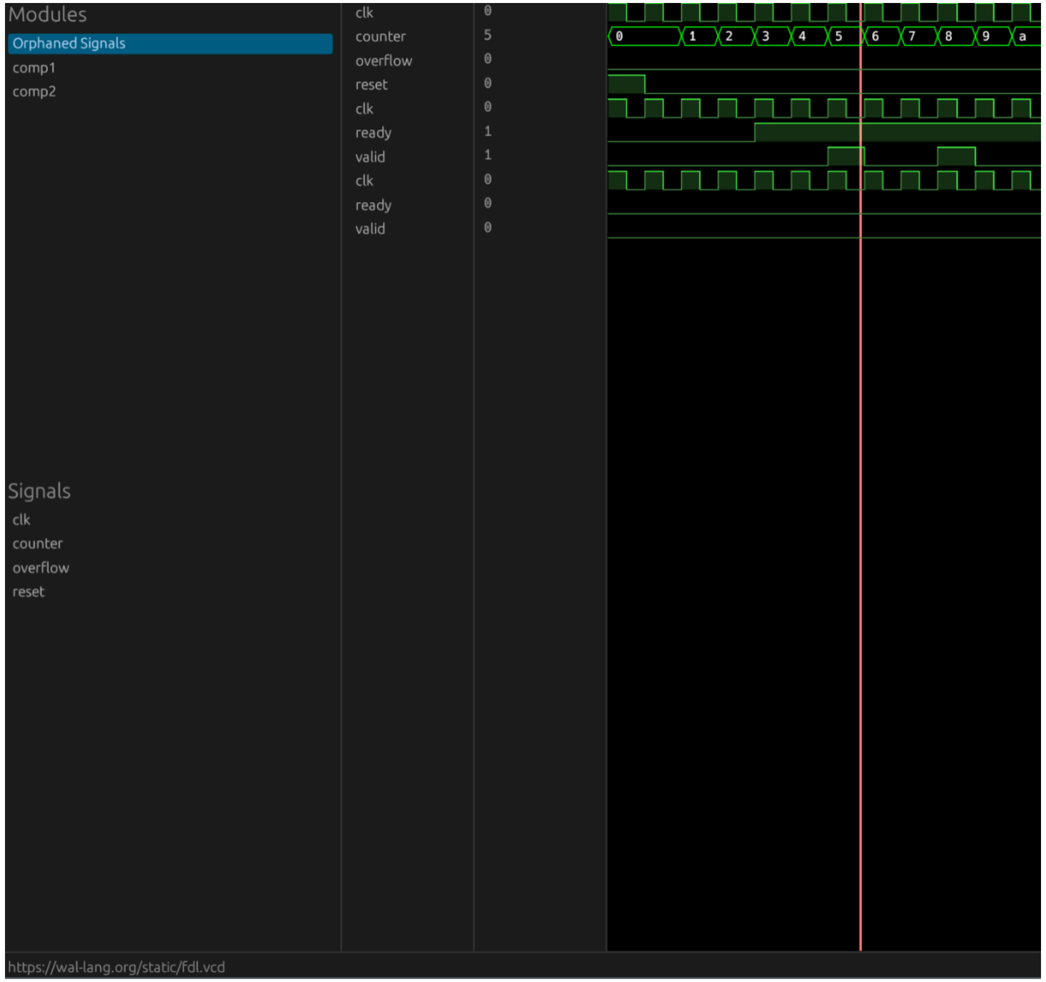
To follow this tutorial:

1. Install wal (wal-lang.org)
2. Download fdl.vcd (wal-lang.org/static/fdl.vcd)

To start WAL with example trace type:

„$ wal -l fdl.vcd“

Hands-On: Surfer Waveform Viewer



Visit: Oh no, this link was only available to people at FDL ☹

- Loads fdl trace automatically
- Select signals to show them
- OR press <SPACE>
  - add_signal …
  - add_scope …

Check out Surfer at:
https://surfer-project.org/

Reading Signal Values

- This is a signal access!

```
(define a 5)
(print (+ a b)
```

- What does this do?

```
a = 5
print(a + b)
```

```
Traceback (most recent call last):
  File "error.py", line 2, in <module>
    print(a + b)
NameError: name 'b' is not defined
```

Ouch!

- **Free variables are signals in waveforms**
- Value depending on:
  - Loaded waveform
  - Time index in the waveform

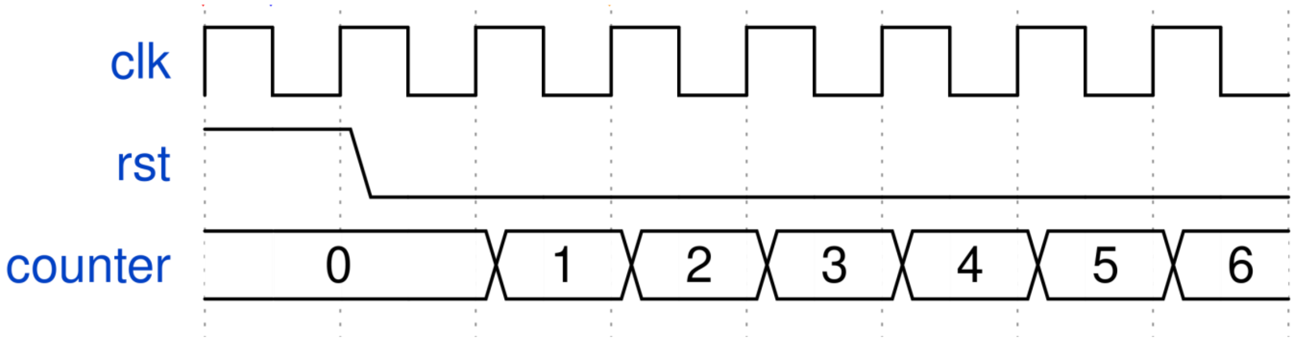
Reading Signal Values (Example)

- We have a simple counter
- **index = 0**, after waveform is loaded
- Read a signal by typing it's name
- Move the index with `(step)`

0: `>-> clk` ⇒ 1
`>-> (step 1)`

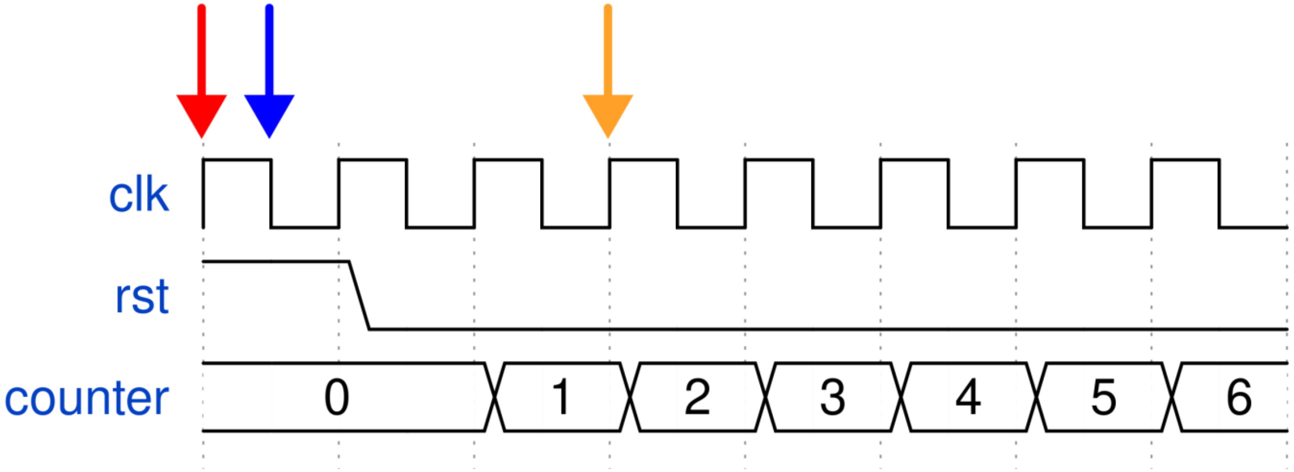
1: `>-> clk` ⇒ 0
`>-> (step 5)`

6: `>-> clk` ⇒ 1
`>-> counter` ⇒ 2



Hands-On: Reading Signal Values

```
>-> clk
1
>-> (step 1)
#t
>-> INDEX
1
>-> clk
0
>-> (step 5)
#t
>-> counter
2
```

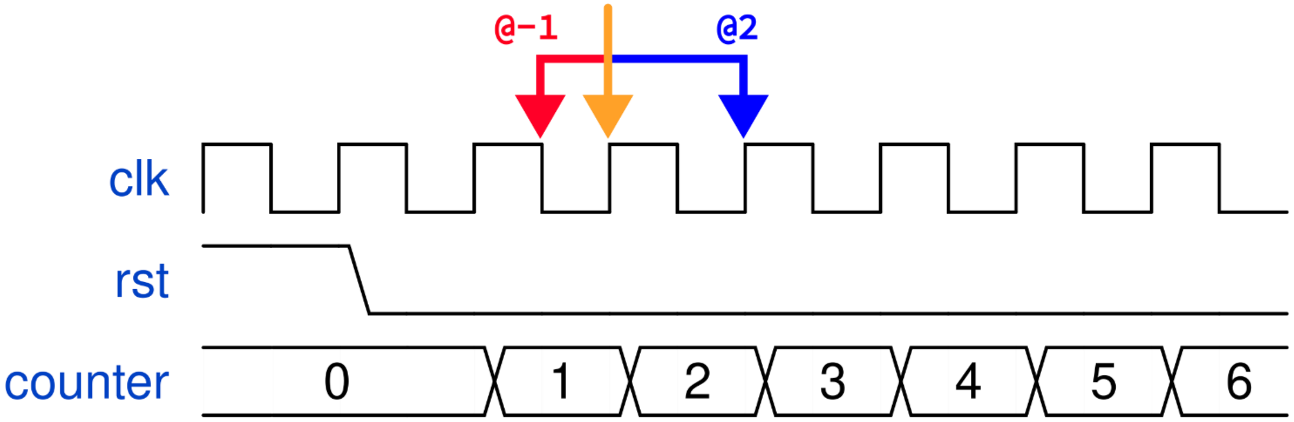


Relative Evaluation

- Index can be locally modified with `expr@offset` syntax
  - evaluated at INDEX + 1: `signal@1`
  - Signal value change: `(!= signal signal@1)`
  - @ can be applied to every expression (not just signals)
  - Is `x` larger than 5 two indices ahead?: `(> x 5)@2`

Hands-On: Relative Evaluation

```
>-> counter
2
>-> counter@-1
1
>-> counter@2
3
>-> (= counter 4)@2
#f
```



Variables

- Define a new variable using `define`
  - `(define x 5)`
- Change variables using `set`
  - `(set [x 22])`
- Create local bindings using `let`
  - `(let ([x 10]) x)`
  - `(let ([x 10] [y 20]) (+ x y))`

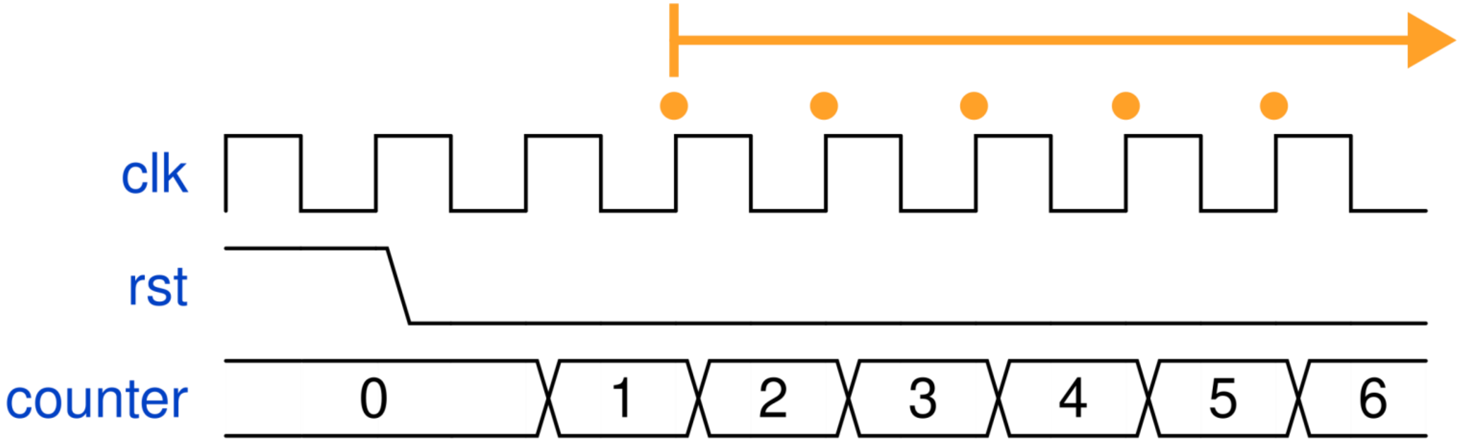
Hands-On: Variables

```
>-> (define x 5)
5
>-> x
5
>-> (+ x 1)
6
>-> (set [x “FDL”])
“FDL”
>-> x
“FDL”
>-> (+ x 1)
“FDL1”
```

Special Functions

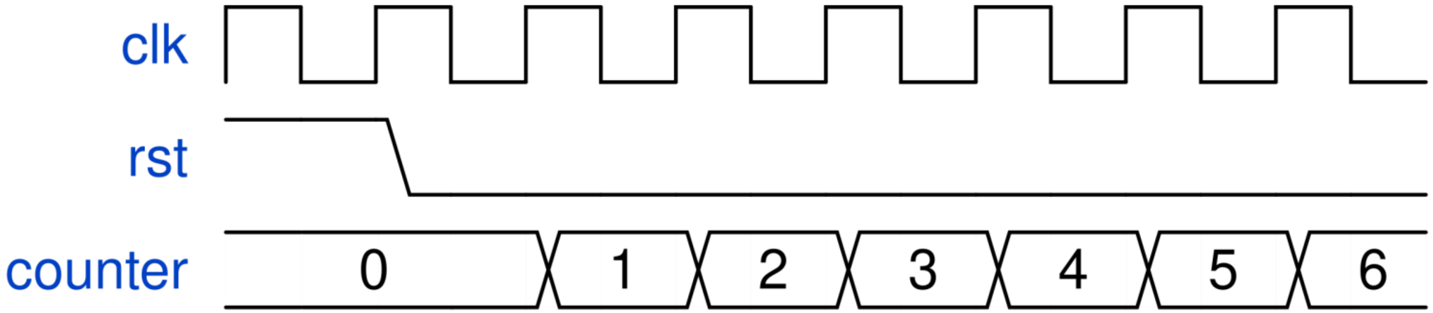
- Signal events
  - `(rising x)` => `(&& (= x 1) (= x@-1 0))`
  - `(falling x)` => `(&& (= x 0) (= x@-1 1))`
  - `(stable x)` => `(= x x@-1)`
- Step over waveform and evaluate body whenever condition is true
  - Starts at the current `INDEX`
  - `(whenever condition body+)`
- Find all indices at which condition is true
  - `(find condition)`
- Count how often condition is true
  - `(count condition)`
- Step forward until condition is true
  - `(step-until condition)`

Hands-On: Whenever




```
>-> (whenever clk (print INDEX " " counter))
6 2
8 3
10 4
…
```

Hands-On: Find, Count

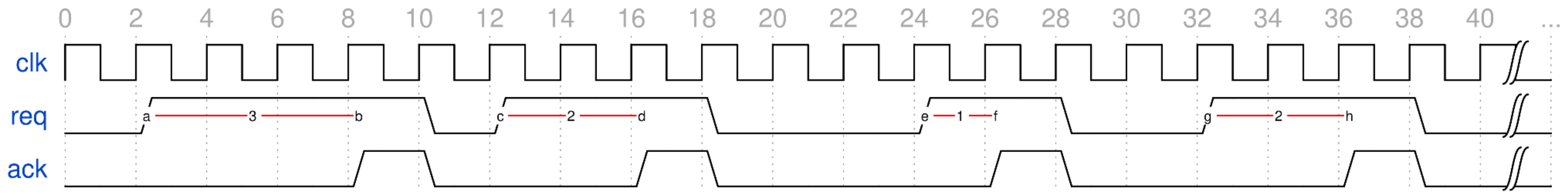



```
>-> (find (= counter 2))
(6 7 38 39 70 71)
>-> (count (= counter 2))
6
```

Example: Average Delay

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

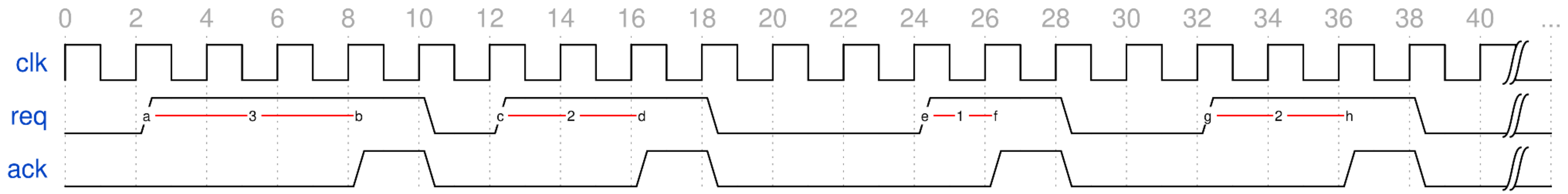
```
(whenever clk
    … always evaluated when clk = 1 …)
```



Example: Average Delay (1)

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

```
(define wait 0)
(define packets 0)
(whenever (rising clk)
     (when (&& req (! ack)) (inc wait))
     (when (&& req ack) (inc packets)))

(print (/ wait packets))
```



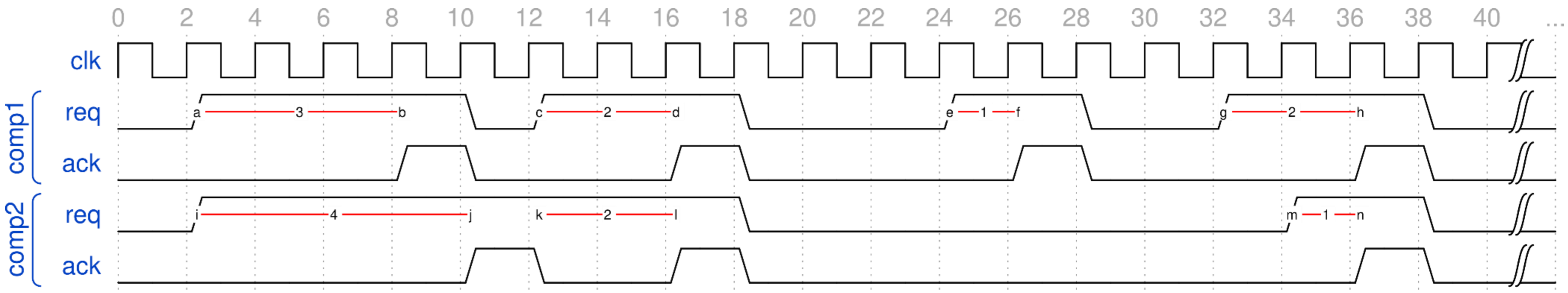
(3+2+1+2) / 4 = 8/4 = 2

Groups

- HW designs ideal for writing generic code!
  - Handshaking is common
  - Standardized interfaces (AXI, AHB, Wishbone, SPI, …)
- For example, two instances of the handshaking bus
- Write expressions only using the shared suffix of the name
- Expand `#suffix` to full name
  - `#req` => either `comp1.req` or `comp2.req`

- `clk`
- `comp1.req`
- `comp1.ack`
- `comp2.req`
- `comp2.ack`

- `comp1.`
- `comp2.`



Hands-On: Groups

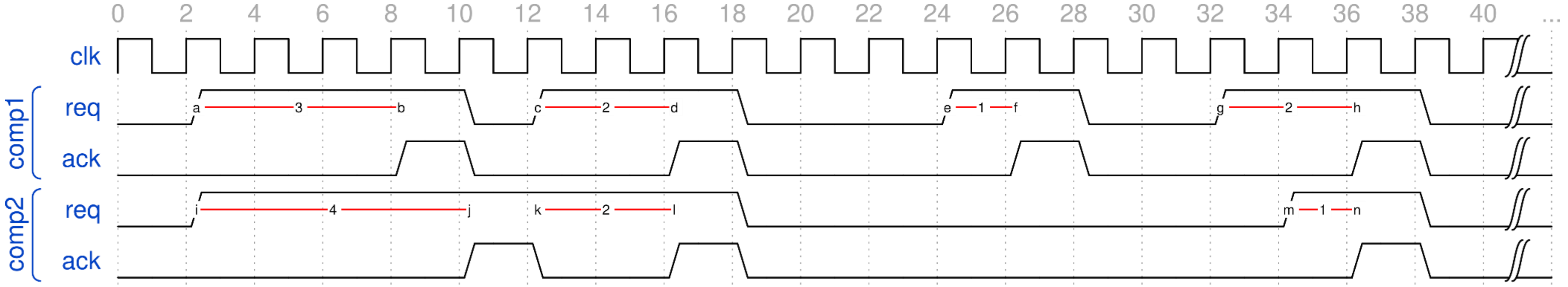
```
>-> SIGNALS
(… "comp1.clk" "comp1.ready" "comp1.valid"
   "comp2.clk" "comp2.ready" "comp2.valid")
>-> (groups clk ready valid)
("comp1." "comp2.")
>-> (groups clk)
("" "comp1." "comp2.")
```

Example: Average Delay (2)

- Wrap analysis in `in-groups` function
- Expression evaluated in each group
- `#signal` expanded to full name

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

```
(define wait 0)
(define packets 0)
(in-groups (groups req ack)
  (whenever (rising 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`, `map`, `for` , …
  - Hashmaps:
    - (geta symbol key1 key2 …)
    - (seta symbol key1 key2 … data)
- Extracting bits from signals
  - `signal[i]`, `signal[h:l]`
- WAL as a compilation target from other languages

Python in the WAL World

- WAL can call Python functions
- You can use all your beloved packages

```
(import riscv)

(call riscv.decode instruction)
```

```python
from riscvmodel import code
from riscvmodel.variant import Variant

variant = Variant('RV32G')

def decode(instr):
   try:
        return str(code.decode(instr, variant))
    except Exception:
        return 'Invalid: ' + str(instr)
```

WAL in the Python World

- Python can run WAL

```
>>> from wal.core import Wal
>>> w = Wal()
>>> w.eval('(print "Hello!")')
Hello!
```

Applications: Pipeline Explorer

Time	fetch	decode	execute	memory	writeback
23	sw x7, 84(x3)	sub x7, x7, x2	add x7, x4, x5	slt x4, x7, x2	-
24	lw x2, 96(x0)	sw x7, 84(x3)	sub x7, x7, x2	add x7, x4, x5	slt x4, x7, x2
25	add x9, x2, x5	lw x2, 96(x0)	sw x7, 84(x3)	sub x7, x7, x2	add x7, x4, x5
26	jal x3, .+8	add x9, x2, x5	lw x2, 96(x0)	sw x7, 84(x3)	sub x7, x7, x2
27	jal x3, .+8	add x9, x2, x5	-	lw x2, 96(x0)	sw x7, 84(x3)
28	addi x2, x0, 1	jal x3, .+8	add x9, x2, x5	-	lw x2, 96(x0)
29	add x2, x2, x9	addi x2, x0, 1	jal x3, .+8	add x9, x2, x5	-
30	add x2, x2, x9		-	jal x3, .+8	add x9, x2, x5
31	sw x2, 32(x3)		-	-	jal x3, .+8

pc: 108
rd: 2
rs1: 0
rs2: 1
flush: 1
stall: 0
value: 1048851
state: flush

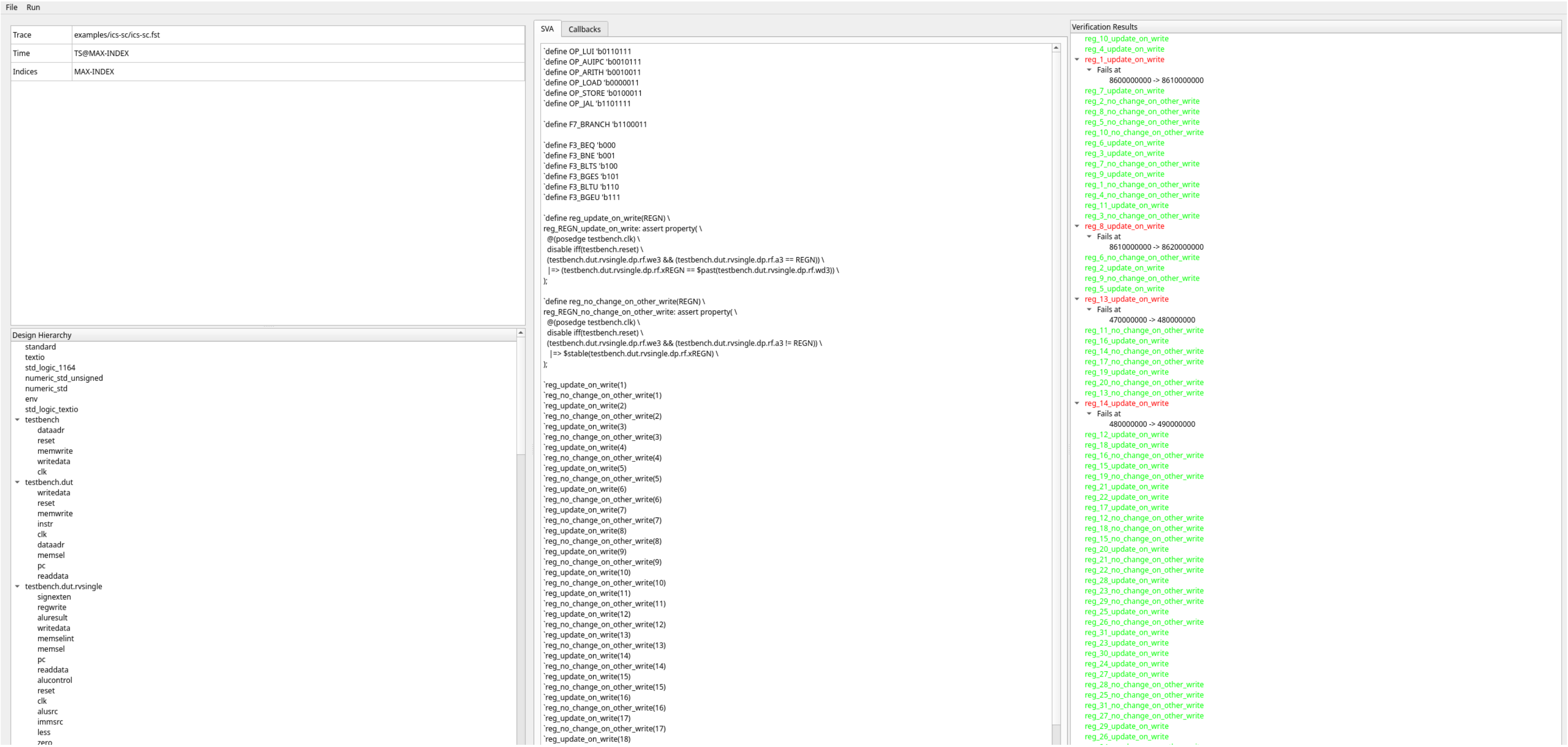
Enter Command

```
(require pipeline)

(stage fetch
       (value tb.dut.dp.instrf@1)
       (stall tb.dut.dp.stallf)
       (log stallf tb.dut.dp.stallf)
       (log pc tb.dut.dp.pcf))

(stage decode
       (update (! tb.dut.dp.stalld))
       (stall tb.dut.dp.stalld)
       (flush tb.dut.dp.flushd)

       (log pc fetch-pc@-1)
       (log rd tb.dut.dp.rdd)
       (log rs1 tb.dut.dp.rs1d)
       (log rs2 tb.dut.dp.rs2d))

(stage execute
       (update (! tb.dut.dp.flushe))
       (flush tb.dut.dp.flushe)
       (log pc decode-pc@-1))

(stage memory)

(stage writeback)
```

Applications: Processor Analysis

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%

Applications: SVA -> WAL Compiler



Conclusion

- WAL enables programmable waveform analysis
  - Data aggregation
  - Data visualization
  - Complex queries
- WAL availability
  - GitHub: https://github.com/ics-jku/wal
  - Documentation: https://wal-lang.org
  - Support: support@wal-lang.org

Get the Most out of Your Waveforms

From Non-functional Analysis to Functional Debug via Programs on Waveforms

Daniel Große, Lucas Klemmer
Institute for Complex Systems (ICS)
Web: jku.at/ics wal-lang.org
Email: daniel.grosse@jku.at, lucas.klemmer@jku.at

JOHANNES KEPLER
UNIVERSITY LINZ

Papers

- Lucas Klemmer and Daniel Große. WAL: a novel waveform analysis language for advanced design understanding and debugging. In *ASP-DAC*, pages 358-364, 2022. https://ics.jku.at/files/2022ASPDAC_WAL.pdf
- Lucas Klemmer and Daniel Große. Waveform-based performance analysis of RISC-V processors: late breaking results. In *DAC*, pages 1404-1405, 2022. https://ics.jku.at/files/2022DAC_LBR-Waveform-based-Performance-Analyisis-for-RISC-V.pdf
- Lucas Klemmer, Eyck Jentzsch, and Daniel Große. Programmable analysis of RISC-V processor simulations using WAL. In *DVCon Europe*, 2022. https://ics.jku.at/files/2022DVCon_Programmable_Analysis_of_RISC-V_Processor_Simulations_using_WAL.pdf
- Lucas Klemmer and Daniel Große. A DSL for visualizing pipelines: A RISC-V case study. In *RISC-V Summit Europe*, 2023. https://ics.jku.at/files/2023RISCVSummit_DSLforVisualizingPipelines.pdf
- Frans Skarman, Lucas Klemmer, Oscar Gustafsson, and Daniel Große. Enhancing compiler-driven HDL design with automatic waveform analysis. In *FDL*, 2023. https://ics.jku.at/files/2023FDL_Enhancing-Compiler-Driven-HDL-Design-with-WAL.pdf
- Lucas Klemmer and Daniel Große. Towards a highly interactive design-debug-verification cycle. In *ASP-DAC*, 2024.