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
Get the Most out of Your Waveforms
Design Flow

- Specification
- System model

Hardware:
- Partitioning
  - TLM Model
  - RTL Model
  - Custom & IP-Cores
  - Synthesis / manual
  - Logic synthesis
  - Phys. Implementation
  - Chip

Software:
- Operating System
- Driver
- Application Software

Integration / Verification
Design Flow

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

Synthesis / manual
Design Flow

Waveforms

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

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

Result of expression

clk

1
Hands-On: FDL Tutorial Website

Visit:

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

- **Right side**
  - The Tutorial slides

Oh no, this link was only available to people at FDL 😞
Hands-On: The WAL Shell

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:

```
user@e25a:~$ wal -l fdl.vcd

>-> 1
1
>-> (+ 1 2)
3
>-> (= 1 2)
#f
```

"$ wal -l fdl.vcd"
Hands-On: Surfer Waveform Viewer

Visit:

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

Oh no, this link was only available to people at FDL 😞

Check out Surfer at:
https://gitlab.com/surfer-project/surfer
Reading Signal Values

• This is a signal access!

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

• Free variables are signals in waveforms

• Value depending on:
  ◦ Loaded waveform
  ◦ Time index in the waveform

• 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!
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: \(\rightarrow\) clk \(\Rightarrow 1\)  
\(\rightarrow\) (step 1)

1: \(\rightarrow\) clk \(\Rightarrow 0\)  
\(\rightarrow\) (step 5)

6: \(\rightarrow\) clk \(\Rightarrow 1\)  
\(\rightarrow\) counter \(\Rightarrow 2\)
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

```scheme
;; counter
2
;; counter@-1
1
;; counter@2
3
;; (= counter 4)
#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))`
<table>
<thead>
<tr>
<th>Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&gt;-&gt; (define x 5)</code></td>
<td>5</td>
</tr>
<tr>
<td><code>&gt;-&gt; x</code></td>
<td>5</td>
</tr>
<tr>
<td><code>&gt;-&gt; (+ x 1)</code></td>
<td>6</td>
</tr>
<tr>
<td><code>&gt;-&gt; (set [x “FDL”])</code></td>
<td>“FDL”</td>
</tr>
<tr>
<td><code>&gt;-&gt; x</code></td>
<td>“FDL”</td>
</tr>
<tr>
<td><code>&gt;-&gt; (+ x 1)</code></td>
<td>“FDL1”</td>
</tr>
</tbody>
</table>
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
...```

 clk

 rst

 counter 0 1 2 3 4 5 6
Hand-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: \( \text{req} \land \neg \text{ack} \)
  - Sending: \( \text{req} \land \text{ack} \)
- Count states
- Result = \(|\text{waiting}| / |\text{sending}|\)
Example: Average Delay (1)

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

\[
(3 + 2 + 1 + 2) / 4 = 8 / 4 = 2
\]
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 \texttt{comp1.req} or \texttt{comp2.req}

\begin{itemize}
  \item \texttt{clk}
  \item \texttt{comp1.req}
  \item \texttt{comp1.ack}
  \item \texttt{comp2.req}
  \item \texttt{comp2.ack}
  \item \texttt{comp1.}
  \item \texttt{comp2.}
\end{itemize}
Hands-On: Groups

```scala
>>> 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 \texttt{in-groups} function
• Expression evaluated in each group
• \texttt{#signal} expanded to full name

\begin{verbatim}
(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))
\end{verbatim}

\[
\frac{(3+2+1+2) + (4+2+1)}{7} = \frac{8 + 7}{7} = \frac{15}{7} \approx 2.1
\]
Other WAL Features

• Data Structures
  ◦ Lists:
    ▪ (first list), (second list), (rest list), ...
    ▪ list[i], list[h:l]
    ▪ fold, map, for, ...
  ◦ Hashmaps:
    ▪ (get symbol key1 key2 ...)
    ▪ (set 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

```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

```python
>>> from wal.core import Wal
>>> w = Wal()
>>> w.eval('(print "Hello!")')
Hello!
```
Applications: Pipeline Explorer

(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

<table>
<thead>
<tr>
<th>Core</th>
<th>Configuration</th>
<th>IPC</th>
<th>Stalled Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERV</td>
<td>Servant</td>
<td>0.02</td>
<td>Not pipelined</td>
</tr>
<tr>
<td>PicoRv32</td>
<td>Default</td>
<td>0.24</td>
<td>Not pipelined</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>MicroNoCsr</td>
<td>0.33</td>
<td>63%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>Smallest</td>
<td>0.33</td>
<td>66%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>SmallAndProductive</td>
<td>0.42</td>
<td>54%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>SmallAndProductiveICache</td>
<td>0.47</td>
<td>51%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>TwoThreeStage</td>
<td>0.47</td>
<td>48%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>Secure</td>
<td>0.57</td>
<td>42%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>Linux</td>
<td>0.59</td>
<td>38%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>Full</td>
<td>0.57</td>
<td>35%</td>
</tr>
<tr>
<td>VexRiscv</td>
<td>FullNoMmuMaxPerf</td>
<td>0.63</td>
<td>33%</td>
</tr>
<tr>
<td>IBEX</td>
<td>Default</td>
<td>0.63</td>
<td>48%</td>
</tr>
<tr>
<td>IBEX</td>
<td>Icache</td>
<td>0.89</td>
<td>19%</td>
</tr>
<tr>
<td>TGC</td>
<td>3-Stage</td>
<td>0.61</td>
<td>64%</td>
</tr>
<tr>
<td>TGC</td>
<td>4-Stage v1</td>
<td>0.72</td>
<td>49%</td>
</tr>
<tr>
<td>TGC</td>
<td>4-Stage v2</td>
<td>0.70</td>
<td>45%</td>
</tr>
<tr>
<td>TGC</td>
<td>4-Stage v3</td>
<td>0.70</td>
<td>44%</td>
</tr>
<tr>
<td>TGC</td>
<td>4-Stage v4</td>
<td>0.68</td>
<td>43%</td>
</tr>
<tr>
<td>TGC</td>
<td>5-Stage</td>
<td>0.78</td>
<td>40%</td>
</tr>
</tbody>
</table>
Applications: SVA -> WAL Compiler

<table>
<thead>
<tr>
<th>File Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVA files</td>
<td>Description of SVA files</td>
</tr>
<tr>
<td>WAL files</td>
<td>Description of WAL files</td>
</tr>
</tbody>
</table>

**Design Hierarchy**
- [ ] standard
- [ ] terms
- [ ] obj
- [ ] core
- [ ] unsigned
- [ ] unsigned
- [ ] obj
- [ ] core
- [ ] unsigned
- [ ] unsigned
- [ ] obj
- [ ] core
- [ ] unsigned
- [ ] unsigned

**Compilation Results**
- [ ] compilation successfully completed
- [ ] compilation not complete
- [ ] compilation error
- [ ] compilation warning

**Test Results**
- [ ] test successfully completed
- [ ] test not complete
- [ ] test error
- [ ] test warning

**Notes**
- [ ] additional notes
- [ ] important notes
- [ ] technical notes
- [ ] user notes

**References**
- [ ] relevant references
- [ ] additional references
- [ ] important references
- [ ] user references
Conclusion

• WAL enables **programmable** waveform analysis
  ◦ Data aggregation
  ◦ Data visualization
  ◦ Complex queries

• WAL availability
  ◦ GitHub: [https://github.com/ics-jku/wal](https://github.com/ics-jku/wal)
  ◦ Documentation: [https://wal-lang.org](https://wal-lang.org)
  ◦ Support: [support@wal-lang.org](mailto: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
Papers


