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
  - Logic synthesis
    - Synthesis / manual
    - Phys. Implementation
  - Chip
- Software
  - Operating System
  - Driver
  - Application Software
  - Integration / Verification

- Custom & IP-Cores
- Synthesis / manual
- RTL Model
- Logic synthesis
- Phys. Implementation
- FPGA Prototype
- RTL Simulation
- Virtual Prototype
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

>--> clk
1

Result of expression
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

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:

- 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://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)
```

Ouch!

• Free variables are signals in waveforms
• Value depending on:
  ◦ Loaded waveform
  ◦ Time index in the waveform

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

<table>
<thead>
<tr>
<th>Step</th>
<th>clk Value</th>
<th>Counter Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>clk ⇒ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clk ⇒ (step 1)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>clk ⇒ 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>clk ⇒ (step 5)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>clk ⇒ 1</td>
<td>counter ⇒ 2</td>
</tr>
</tbody>
</table>

![Waveform Diagram]
Hands-On: Reading Signal Values

\[ \text{clk} \]
1
\[ \rightarrow \text{(step 1)} \]
#t
\[ \rightarrow \text{INDEX} \]
1
\[ \rightarrow \text{clk} \]
0
\[ \rightarrow \text{(step 5)} \]
#t
\[ \rightarrow \text{counter} \]
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)@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
  - \((\text{rising } x) \Rightarrow (\&\& ( = x \ 1) ( = x@-1 \ 0))\)
  - \((\text{falling } x) \Rightarrow (\&\& ( = x \ 0) ( = x@-1 \ 1))\)
  - \((\text{stable } x) \Rightarrow ( = x \ x@-1)\)

- Step over waveform and evaluate body whenever condition is true
  - Starts at the current INDEX
  - \((\text{whenever condition body+})\)

- Find all indices at which condition is true
  - \((\text{find condition})\)

- Count how often condition is true
  - \((\text{count condition})\)

- Step forward until condition is true
  - \((\text{step-until condition})\)
Hands-On: Whenever

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

clk
rst
counter

0 1 2 3 4 5 6
Hands-On: Find, Count

clk
rst

| counter | 0 | 1 | 2 | 3 | 4 | 5 | 6 |

>-> (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 = $|\text{waiting}| / |\text{sending}|$

(whenever clk 
... always evaluated when clk = 1 ...)

![Diagram of clk, req, and ack signals]
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 $= |\text{waiting}| / |\text{sending}|$

```
(define wait 0)
(define packets 0)
(whenever (rising clk)
    (when $(\&\& \text{req} (! \text{ack}))$) (inc wait))
    (when $(\&\& \text{req} \text{ack})$) (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
Hands-On: Groups

```plaintext
>> 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

\[
\text{(define wait 0)} \\
\text{(define packets 0)} \\
\text{(in-groups (groups req ack)} \\
\text{  \hspace{1cm} (whenever (rising clk)} \\
\text{    \hspace{1cm} (when (&& #req (! #ack)) (inc wait))} \\
\text{    \hspace{1cm} (when (&& #req #ack) (inc packets)))}} \\
\text{(print (/ wait packets))}
\]

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

• Data Structures
  ° Lists:
    ▪ (first list), (second list), (rest list), ...
    ▪ list[i], list[h:l]
    ▪ fold, map, for, ...
  ° Hashmaps:
    ▪ (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

```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>SmallAndProductiveICache</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
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
Papers


