EmLogic

ALDEC
THE DESIGN VERIFICATION COMPANY

The most error prone corner cases

- and how to detect and avoid them

Aldec Webinar, 14 October 2021

EmLogic

- Independent Design Centre for Embedded Systems and FPGA
- Established 1st of January 2021. **Extreme ramp up**
  - January: 1 person
  - September: → 18 designers (SW:7, HW:1, FPGA:10) - **And still growing…**
- Continues the legacy from bitvis
  - All previous Bitvis technical managers are now in EmLogic
- Verification IP and Methodology provider **UVVM**
- Course provider within FPGA Design and Verification
  - Accelerating FPGA Design (Architecture, Clocking, Timing, Coding, Quality, Design for Reuse, …)
  - Advanced VHDL Verification – Made simple (Modern efficient verification using UVVM)

Agenda

- Corner cases
  - What why, how, when?
  - Probability Occurence & Detection
  - Error prone?
- Detection
  - How and at what stage?
  - Testbench requirements
  - Detection using UVVM
  - Brief intro to UVVM
- Mitigation and Summary

Excerpts from our courses:

'Accellerating FPGA and Digital ASIC Design'

and

'Advanced VHDL Verification – Made simple'

EmLogic

What is a Corner Case?

- 'Edge case' & 'Boundary case'
  - Typically minimum and maximum values
- 'Corner case'
  - Often defined as: A combination of minimum two edge cases
  - Wikipedia:
    '*Situation that occurs only outside normal operating parameters - specifically one that manifests itself when multiple environmental variables or conditions are simultaneously at extreme levels*'

Digital design - de facto corner case:

A particular situation - that is allowed acc. to specification
- combined or single,
- not occurring as a normal situation,
- or a functional event that is likely to be missed.

that **could** be a design problem

Corner Case categories

- Value related
  - Data, control, addresses, ...
- Inter value related
  - Two or more values
- Single-end Cycle related
  - Cycles between events
- Multi-end Cycle related
  - Any comb. of inputs and states
  - Multi-cycle issues
- Value **and** cycle related

A corner case is not
a problem in itself.

It is only a problem when the
design does not work
for this case.

AND this is detected late
in the FPGA development

OR even worse
- not detected until delivered

Corner cases could however create more verification work...

AND - they are definitely Error prone!

EmLogic

Value related corner cases

- Single value
  - E.g. Payload size of 0,1 or maximum
- Multiple values together
  - Payload size AND neighbouring field of Source address
- Value and action sequence
  - FIFO was full when attempting to load new data

→ Have been the focus for
Constrained Random and Functional Coverage for years

→ Lots of tools and methodologies are available

→ **UVVM Constrained Random and Functional Coverage**
will be released next week

➔ Great for value related, but not sufficient for cycle related…

** Several examples of these in our Design course, but this presentation is about cycle related CC*
https://emlogic.no/wp-content/uploads/2021/02/Accellerating_FPGA_Design.pdf

Cycle related corner cases

Occur when the cycle in which an event happens is not fixed – and this event is related to another event – fixed or not

Typically - events in different FSMs (explicit or implicit) that affect each other or affect the same objects

- Simple inter FSM example (e.g. streaming data):
  FSM-1: valid + last, FSM-2: ready (or ack)
  - ready could be delayed by suddenly one more cycle – or more
  - last could suddenly take one more cycle to generate

  → **Lots of possible corners**
- **Any** communication interface:
  - e.g. Read-out vs New data entry
- Any two or more events combining into an FSM
  - E.g. Interrupt or DMA controller receiving two triggers 0-N cycles apart
- Could also be combinations of cycle and value related corner cases

Example 3b: Cycle related (i)

```
if (rx_data_reg read from cpu) then
   <something>
elsif (new byte received from rx) then
   <something>
end if;
```

If-then-else
for non-exclusive actions
e.g. inside single process in UART:

Bug if both are true in the same cycle

```
Different coding style
- Same scenario:

if (rx_data_reg read from cpu) then
   <something>
end if;
.....
if (new byte received from rx) then
   <something>
end if;
```

Bug if <something> is

a) go to new state, or

b) assigning to same object

if both ifs are true in the same cycle

EmLogic

Example 3b: Cycle related (ii)

```
if (rx_data_reg read from cpu) then
   <something>
elsif (new byte received from rx) then
   <something>
end if;
```

If-then-else
for non-exclusive actions
e.g. inside single process in UART:

For clock @ 100 MHz and bit rate @ 100 kHz (→ byte @ 10 kHz)

- Probability of bug = 1:10.000 per byte
- Probability of detection in sequential simulation:
  - For one single byte: 1:10.000
  - For thousands of bytes: 1:10.000 – in lots of testbenches
    > 1:10.000 – in quite a few testbenches but not necessarily much better…
- Probability of detection in the lab: Depends on setup.
- Probability of bug in final product : **High!**

How do we avoid corner cases?

- Most corner cases are given by specification
  - These cannot be avoided in the design
  - Unless specification can be modified
- Some corner cases are added by implementation
  - Some of these cannot be avoided
  - Some can definitely be avoided

A corner case is not a problem in itself.

It is only a problem when the design doesn't work for this case.

AND this is detected late in the FPGA development

OR even worse - not detected until delivered

EmLogic

How detect buggy corner cases?

1. Should be aware of problem during design
2. Must know where to search
3. Must know how to search
4. Must pin-point corner case
5. Must verify they are not buggy

- **Detection and Verification:**
  - If possible - Avoid or reduce
  - Review!
  - Simulation
  - Test

Let's check this against the given example....

EmLogic

Detection - for example 3b

Most important: Design structure

2nd most important: Testbench structure

Example 3b: Cycle related

```
if (rx_data read from cpu) then
   <something>
elsif (new byte received from rx) then
   <something>
end if;
```

If-then-else
for non-exclusive actions
e.g. inside single process in UART:

Bug if both are true in the same cycle

Different coding style

For clock @ 100 MHz and bit rate @ 100 kHz (→ byte @ 10 kHz)

- Probability of bug = 1:10.000 per byte
- Probability of detection in sequential simulation: Approx 1:10.000
- Probability of detection in the lab: Depends on setup.
- Probability of bug in final product : **High!**

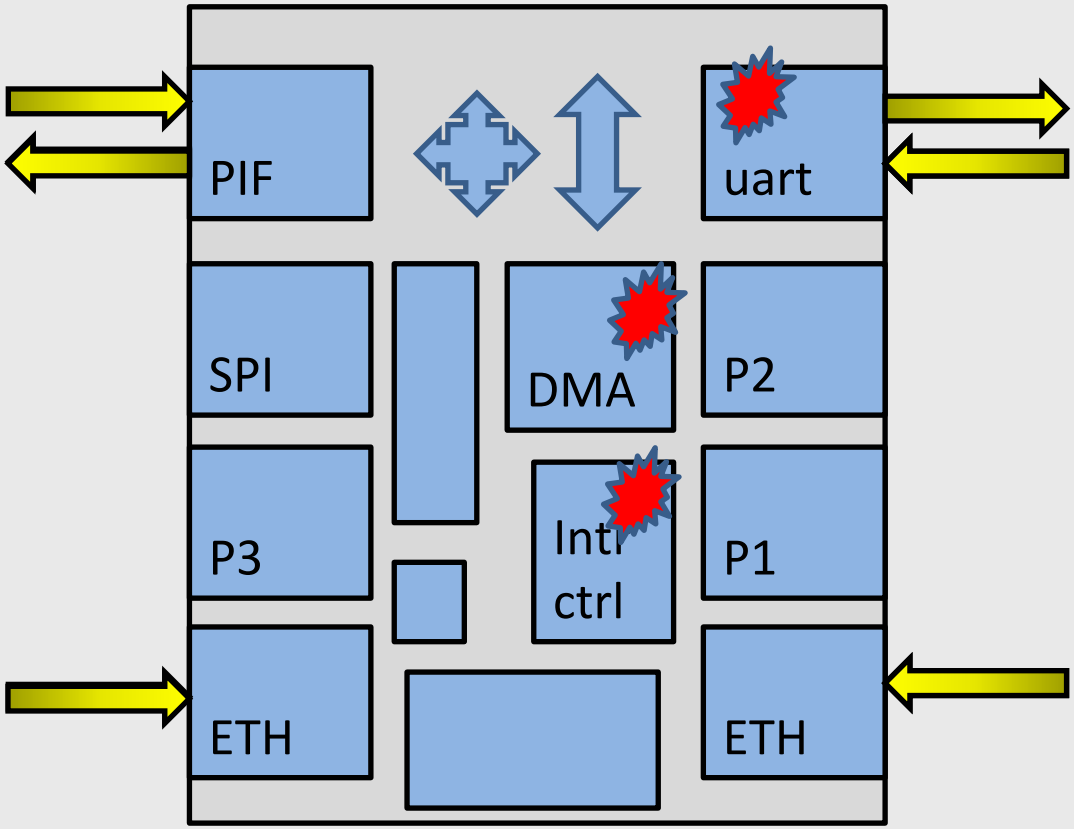
```
   <something>
end if;
```

- Avoid / reduce (spec/arch)
- Simulation
- Review
- Test

- Awareness and experience
  - Generally very low
  - Will easily miss such bugs
- Simulation may detect all
  - Experience important
  - Approach matters
  - Need a good TB architecture - to skew interface stimuli
- Review?
  - Important, but complex
- Lab tests?
  - Lots of test cases possible, but...
  - Will seldom test for cycle relations
  - Often restricted by appl. SW

EmLogic

Chip level functional scenario



Sequential testing

➔ Find some bugs?

➔ Then behaves fine

➔ Value related corners?

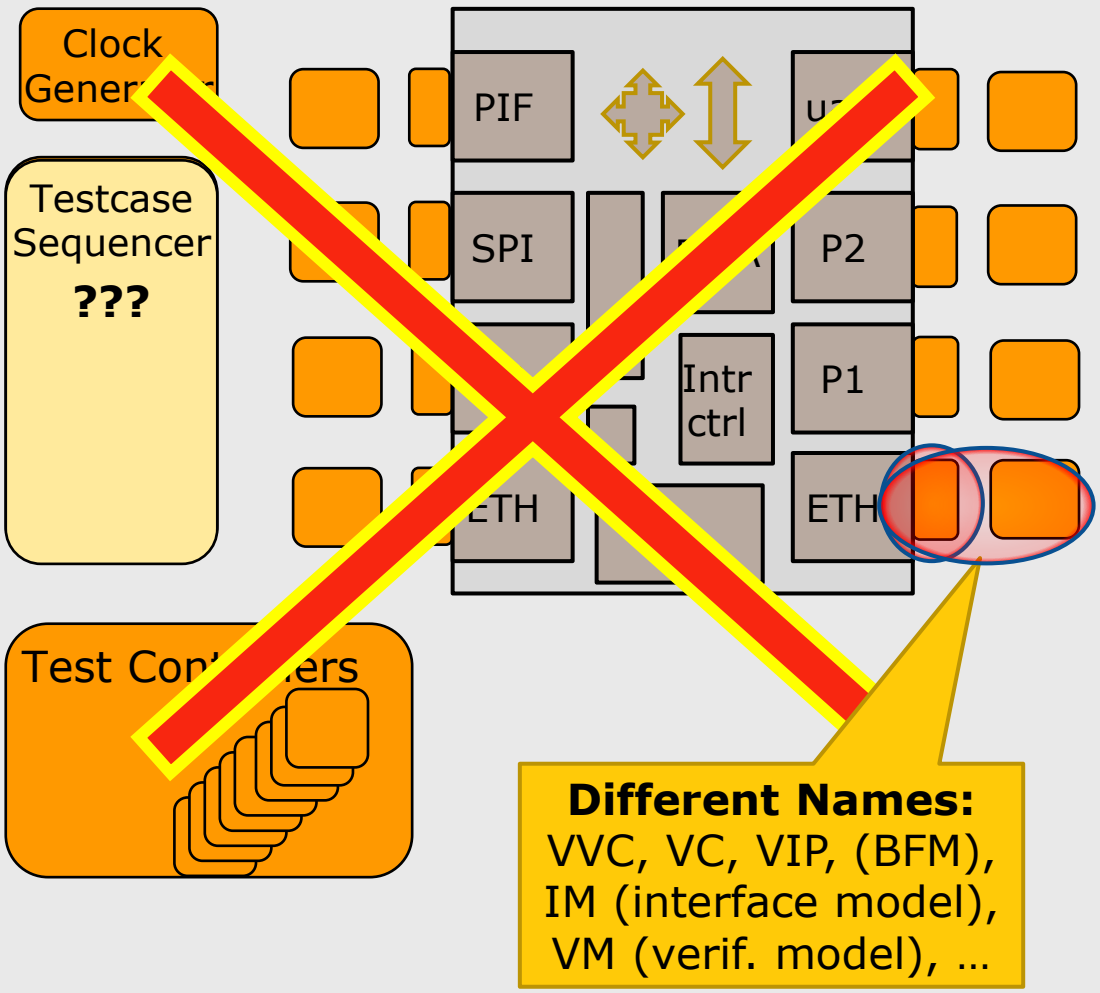
Parallel operation

➔ Multi-end cycle related corners

➔ Bugs are hard to find

EmLogic

Typical testbench approaches



How do designers handle
Cycle related corner cases?

Adding threads | Lab ? | CCL | ?

Ad hoc "structure"

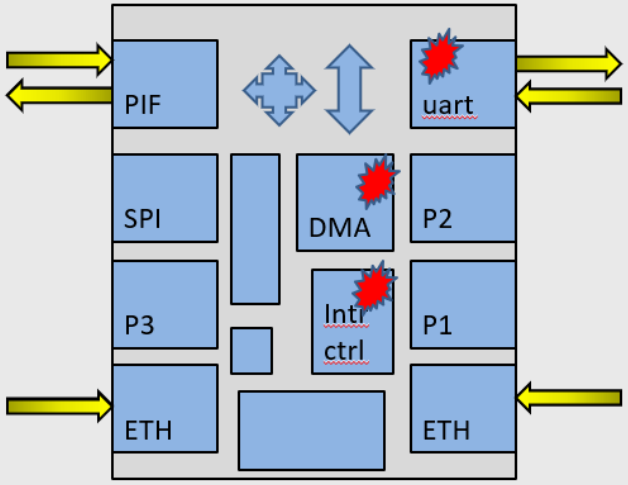
- Fixed structure
- Overview???
- Reuse???
- Bad synchronisation

Good architecture, overview, reuse & sync.

EmLogic

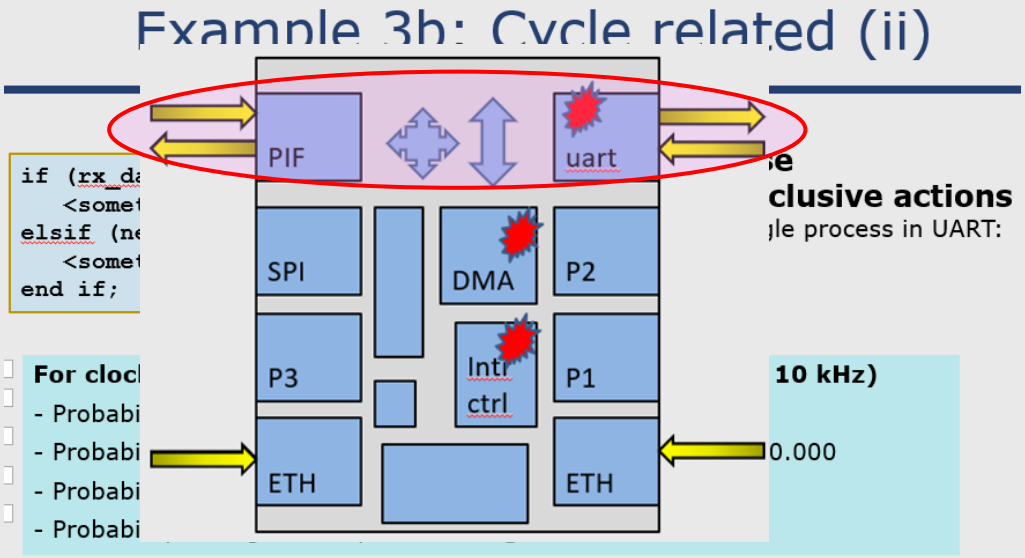
Testbench requirements
- for detection of Cycle related Corner cases

- MUST be able to
  - control all interfaces simultaneously
  - skew one interface wrt. another
  - know when an interface goes inactive
- For readability, maintenance and reuse
  - should avoid inter VC communication or communcation between local test controllers
    - E.g. Test controller A waiting for Test controller B
  - should allow all VCs to be controlled from one single process
    - Thus allowing all VCs to be controlled sequentially in one single sequencer
  - should allow this sequencer to see all activity and completion



EmLogic

Let's simplify the problem



- Let's consider a "dead simple" UART only
- We need to reach the corner cases

➔ Must be able to control SBI RX_DATA_REG read-out vs RX input

➔ and handle all other TB requirement given on the previous slide

Next - how UVVM can reach these corner cases in a structured way

But first a very brief introduction to UVVM

UVVM Utility Library

Testbench infrastructure library

- log(), alert(), report_alert_counters()
- check_value(), await_value()
- check_stable(), await_stable()
- clock_generator(), adjustable_clock_generator()
- random(), randomize()
- gen_pulse()
- block_flag(), unblock_flag(), await_unblock_flag()
- await_barrier()
- enable_log_msg(), disable_log_msg()
- to_string(), fill_string(), to_upper(), replace(), etc…
- normalize_and_check()
- set_log_file_name(), set_alert_file_name()
- wait_until_given_time_after_rising_edge()
- etc…

EmLogic

Lot's of free UVVM BFMs and VVCs

- AXI4-lite
- AXI4 Full
- AXI-Stream Transmit and Receive
- UART Transmit and Receive
- SBI
- SPI Transmit and Receive
- I2C Transmit and Receive
- GPIO
- Avalon MM
- Avalon Stream Transmit and Receive
- RGMII Transmit and Receive
- GMII Transmit and Receive
- Ethernet Transmit and Receive
- Wishbone
- Clock Generator
- Error Injector

All:

- Free
- Open Source
- Well documented
- Example Testbenches

The largest collection of VHDL Interface Models

VVC: VHDL Verif. Comps.

- Includes the corresponding BFM

Allows:

- Simultaneous interface handling
- Synchronization of interfaces
- Skewing between interfaces
- Additional protocol checkers
- Local sequencers
- Activity detection
- Simple reuse between projects

The newer stuff

- ESA Extensions in ESA-UVVM-1
  - **Scoreboarding**
  - **Monitors**
  - Controlling randomisation and functional coverage
  - Error injection (Brute force and Protocol aware)
  - Local sequencers
  - Controlling property checkers
  - **Watchdog** (Simple and Activity based)
  - Transaction info
  - Hierarchical VVCs - And Scoreboards for these
  - **Specification Coverage** (Requirement/test coverage)

ESA is helping VHDL designers speed up FPGA and ASIC development and improve their product quality!

EmLogic

The brand new stuff – October 2021

- Enhanced Randomisation

```
addr <= my_addr.rand(0, 18, ADD,(30,31), EXCL,(7));
```

```
addr <= my_addr.rand_range_weight((0,18,4),(19,31,1));
```

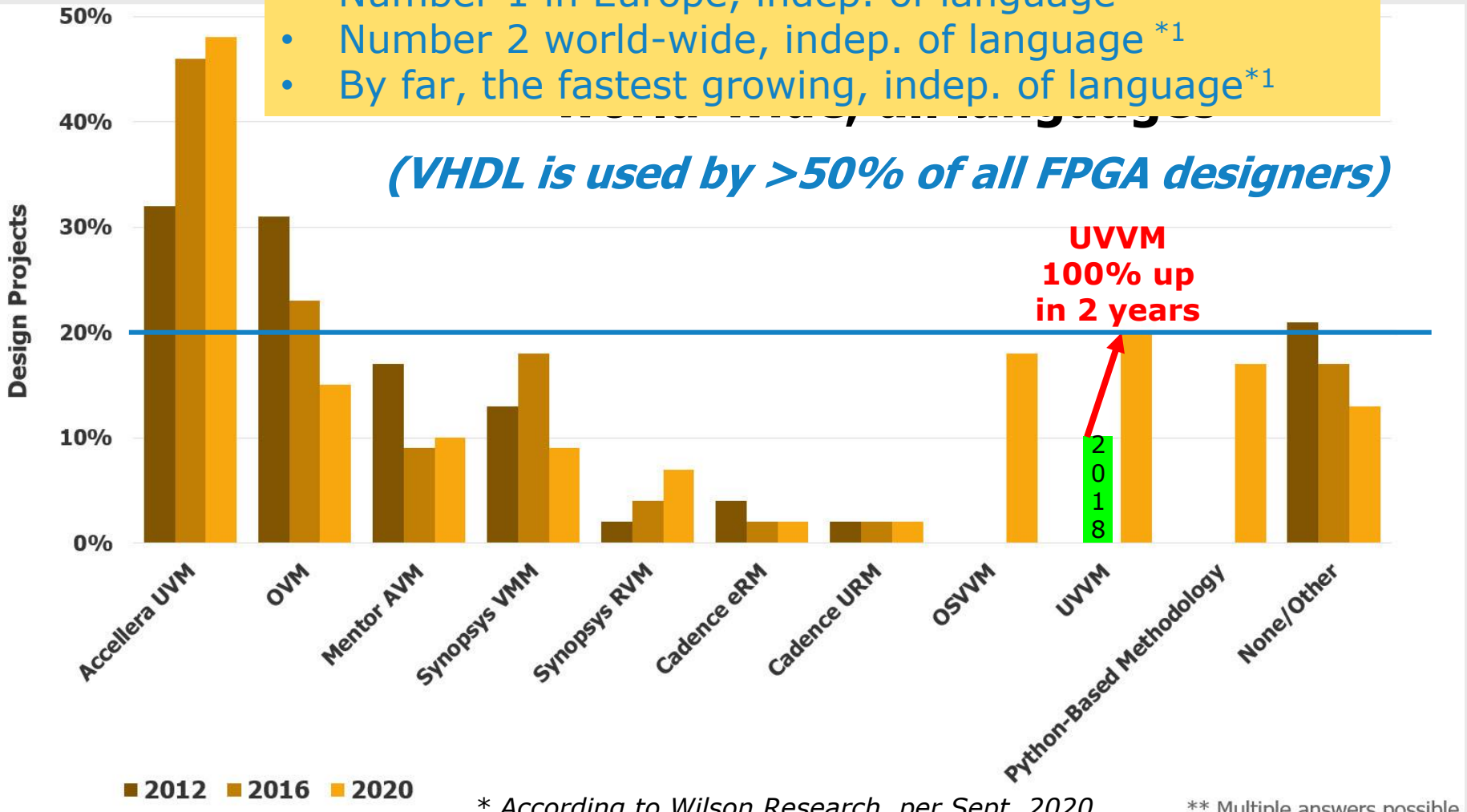
- Optimised Randomisation
  - Randomisation without replacement
  - Weighted according to target distribution AND previous events

  → the lowest number of randomisations for a given target
- Functional Coverage
  - Based on functional coverage in SV
    - But in VHDL, and without all the complexity of SV and UVM
  - Fully integrated with UVVM, but may be used stand-alone
- Will be released next week

UVVM – World-wide

- Number 1 world-wide for VHDL verification *1
- Number 1 in Europe, indep. of language *1
- Number 2 world-wide, indep. of language *1
- By far, the fastest growing, indep. of language*1

(VHDL is used by >50% of all FPGA designers)



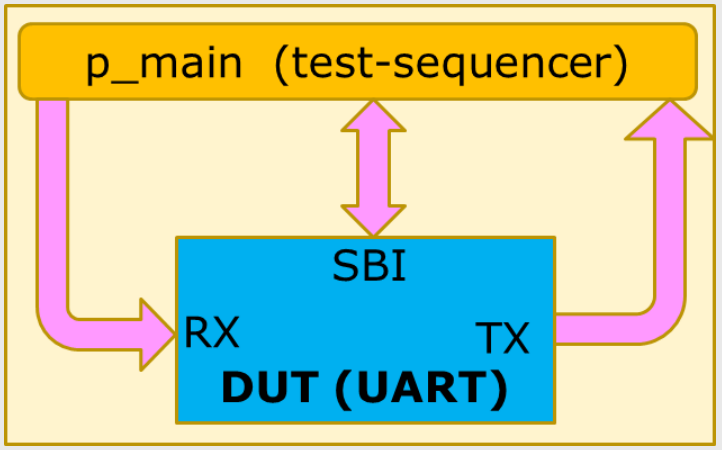
* *According to Wilson Research, per Sept. 2020*

** Multiple answers possible

BFM procedures are not sufficient

BFM: Defined here as a procedure only

- BFMs are great for simple testbenches
  - Dedicated procedures in a simple package
  - Just reference and call from a process
- BUT
  - A process can only do one thing at a time
    - Either execute that BFM
    - **Or** execute another BFM
    - **Or** do something else
- To do more than one thing:
  → Need an entity (or component)
  (VC = Verification Component)



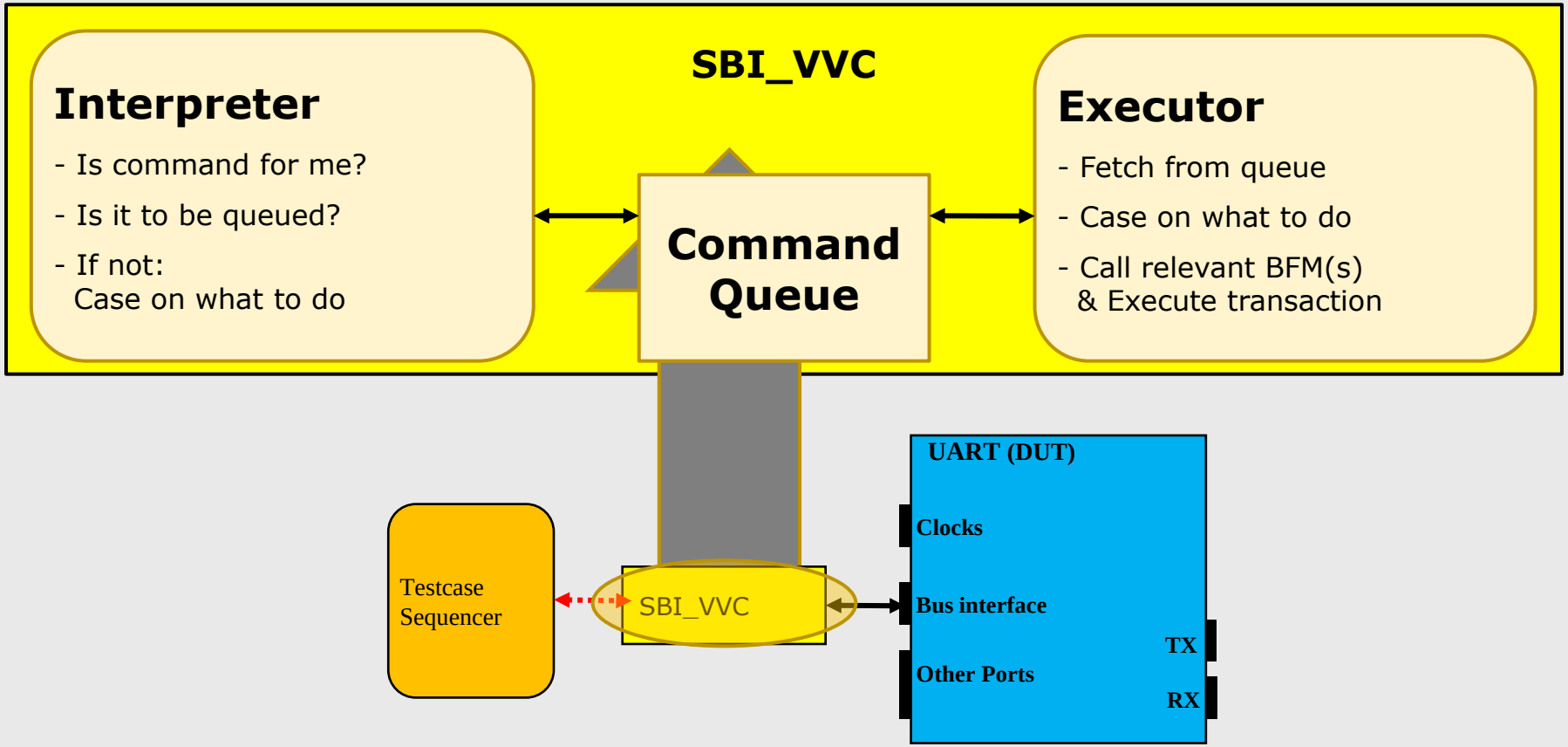
sbi_write(C_TX, x"B3")

uart_expect(x"B3")

VVC: VHDL Verification Component (UVVM VC with extended functionality)

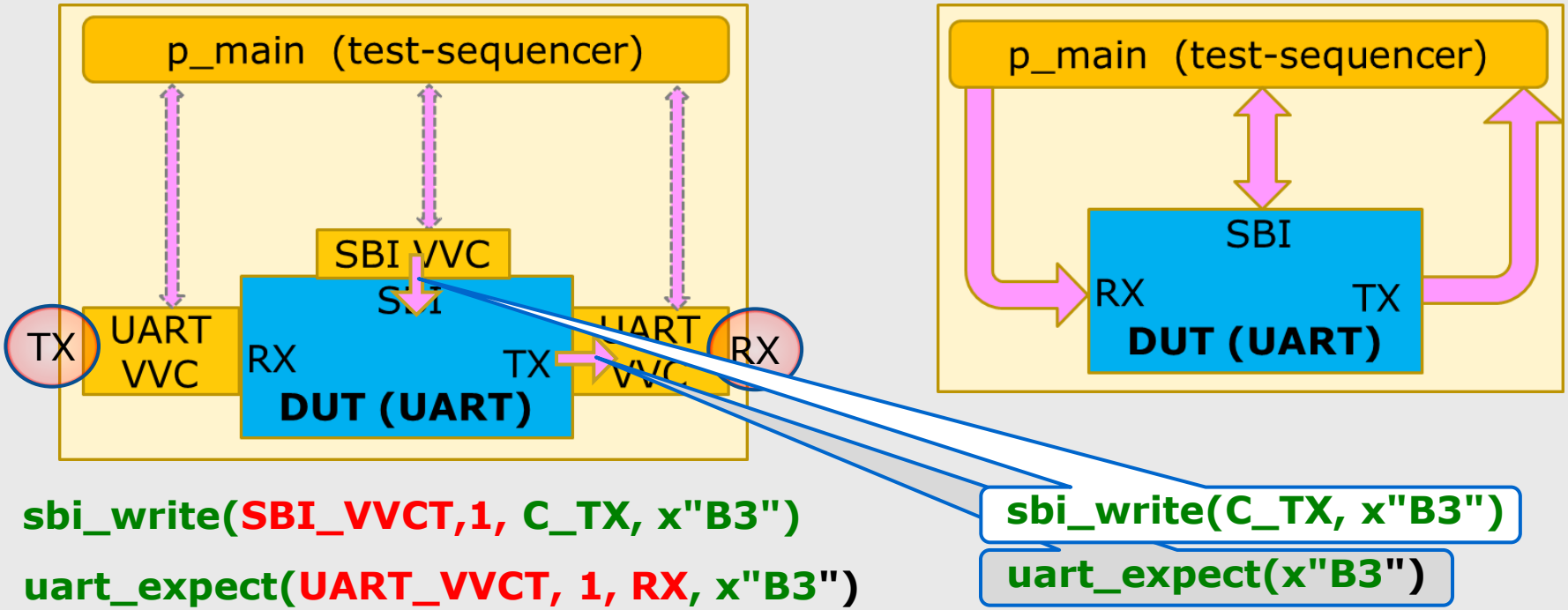
EmLogic

VVC: VHDL Verification Component



EmLogic

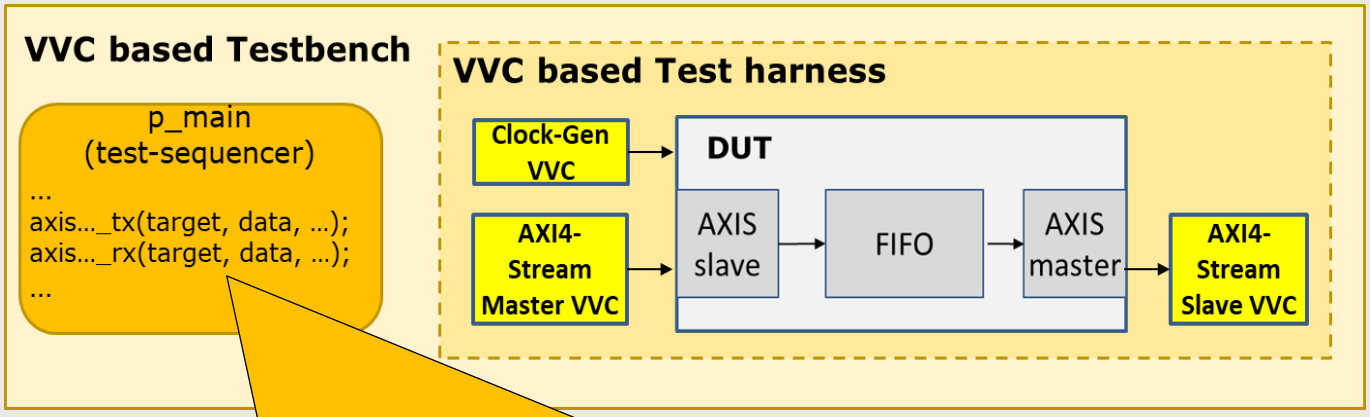
BFM to VVC: How?



UVVM VVCs also include: **Why?**

Delay-insertion, command queuing, completion detection, activity registration, multicast & broadcast, termination, set-up, data fetch, multi-channel support, interface checkers, scoreboards, transaction info, local sequencers, etc …

Simplicity where needed the most



TB with harness

VVCs

Test sequencer

```
axistream_transmit(AXISTREAM_VVCT,0, v_data_array, msg);

axistream_expect(AXISTREAM_VVCT,1, v_exp_array, "Expecting **** ");
```

Totalt Workload for complete verification

Project A

Project B

Project C

Project D

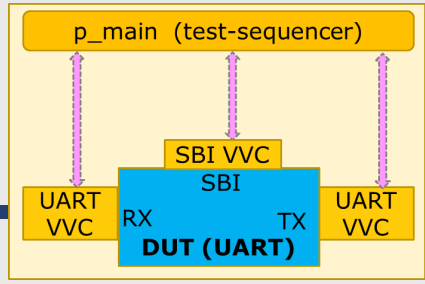
Test sequencer is by far the most time consuming
➔ The most important to simplify
➔ Especially for medium to complex DUTs

?

!

EmLogic

Verification plan - In plain English (1)

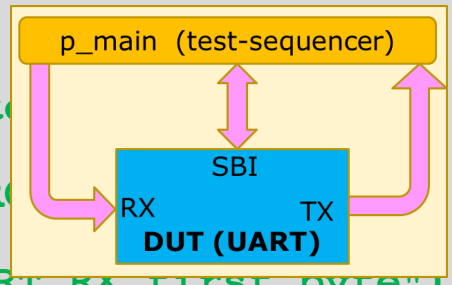


1. **First** Apply data to UART RX, **then** Wait for RX interrupt
   then Read reg RX_DATA. (to show standard serial sequencer)

Pure sequential - Using Sequential BFMs only

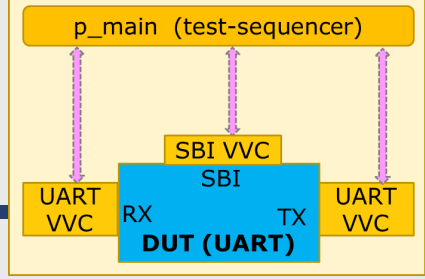
```
uart_transmit(x"A1", "First byte on UART RX");
await_value(rx_empty, '0', 0, 12*bit_period, ERROR, message);
sbi_check(C_ADDR_RX_DATA, x"A1", "UART RX first byte");
```

Pure sequential - using UVVM VVCs




```
uart_transmit(UART_VVCT,1,TX,  x"A1", "First byt
await_value(rx_empty, '0', 0, 12*bit_period, ERR
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "UAR
[ await_completion(SBI_VVCT,1, C_TIMEOUT_SBI_CHECK);   ]
```

Verification plan - In plain English (2)



2. Apply data to UART RX and read reg RX_DATA **at the same time**

   (i.e. effectively reading the previously received RX_DATA)

Parallel operation - using UVVM VVCs

```
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1","UART RX first byte");
uart_transmit(UART_VVCT,1,TX,  x"A2", "Second byte on UART RX");
```

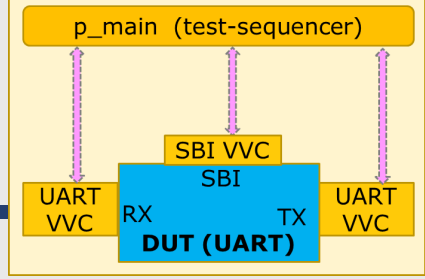
Orden doesn't matter

Parallel operation - using UVVM VVCs

Functionally 100% equal to the above

```
uart_transmit(UART_VVCT,1,TX,  x"A2", "Second byte on UART RX");
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "UART RX first byte");
```

Verification plan - In plain English (3)



3. Read the previous RX data

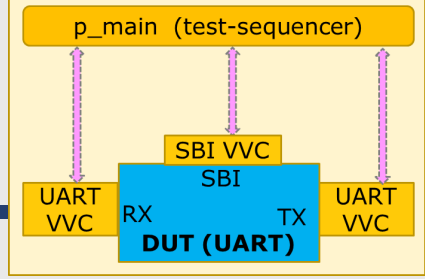
   when the new RX data has just been received

Parallel operation - using UVVM VVCs

```
uart_transmit(UART_VVCT,1,TX  x"A2", "Second byte on UART RX");
insert_delay(SBI_VVCT,1, C_FRAME_TIME + N * C_CLK_PERIOD);
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "UART RX first byte");
```

But – How can we be sure this is the exact problem area?

Verification plan - In plain English (4)



4. Extend the range in **3.** to include all possible cycle corner cases

Parallel operation - using UVVM VVCs

```
for i in -C_CYCLES_BEFORE_CORNER to C_CYCLES_AFTER_CORNER loop
  uart_transmit(UART_VVCT,1,TX, data(i), "New byte on UART RX");
  insert_delay(SBI_VVCT,1, C_FRAME_TIME + i * C_CLK_PERIOD);
  sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, data(i-1), message);
  await_completion(UART_VVCT,1,TX, "Finish before next transmit");
end loop;
```

Similar solution applies if corners between more than two interfaces.

"Normal VCs" are not sufficient

- Also requires
  - Activity control including delay insertion for skewing
  - Completion detection for skewing and alignment
- All controlled from one file
  - Imagine controlling this from two separate processes
  - And then what when the CPU interface need to be skewed wrt. other interfaces as well…

Parallel operation - using UVVM VVC Framework + VVCs

```
for i in -C_CYCLES_BEFORE_CORNER to C_CYCLES_AFTER_CORNER loop
  uart_transmit(UART_VVCT,1,TX data(i), "New byte on UART RX");
  insert_delay(SBI_VVCT,1, C_FRAME_TIME + i * C_CLK_PERIOD);
  sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, data(i-1), message);
  await_completion(UART_VVCT,1,TX, "Finish before next transmit");
end loop;
```

→ Multiple semaphores between multiple test controllers
→ Extremely difficult to understand
→ Near impossible for anyone but the implementor
→ Cannot be reused – anywhere

➔ **UVVM: Alignment, skewing, delay insertion, activity detection, …**

➔ **UVVM: Control of complete TB from one single test sequencer**

UVVM is much more…
But today was focus on Corner Cases

- For an introduction to current functionality like:
  - Utility Library with log, check, await, clock_gen, pulse_gen, etc…
  - BFMs : Functionality and usage
  - VVC and their benefits
  - The new functionality over the last 2 years
  - Why UVVM is the leading VHDL Verification Methodology

Check out my previous Presentations and Aldec Webinars

The latest presentations being:

MODERN VHDL TESTBENCHES
AN AXI-STREAM EXAMPLE, FIRST dead simple, - THEN advanced - Both as simple as possible
At FPGA Conference Europe, 6 July 2021. (Get in touch if you can't get it there)

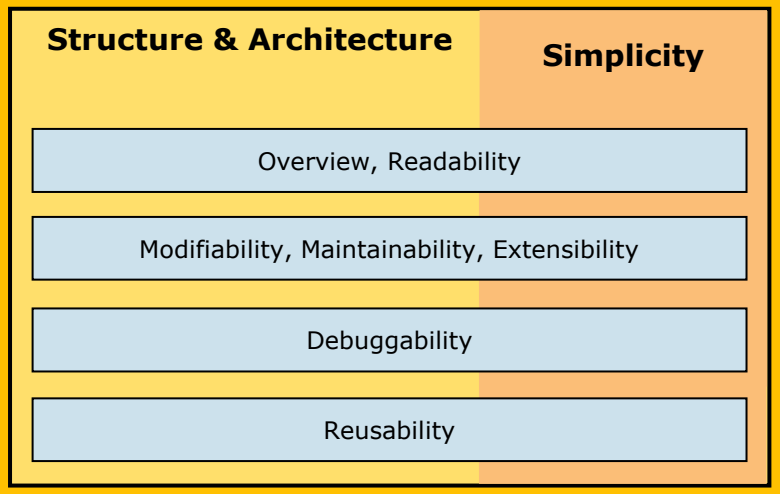
The two latest of many Aldec UVVM webinars

- *2 April 2020:* *UVVM steps up a gear*
- *25 April 2019:* *VHDL testbenches using models, scoreboards and transactions*

(See my linkedIn post for these links)

UVVM in a nutshell

- Huge improvement potential for more structured FPGA verification



UVVM (incl. all) is Open Source

UVVM targets all of this

➔ UVVM is a game changer for efficiency and quality

New ESA project is extending UVVM
- Will release new functionality in Q4

UVVM may save 200-2000 hours on a medium complex project

And at the same time improve TTM, MTBF & LCC

Usage is exploding

- World-wide number 1 for VHDL
- Fastest growing – of all

EmLogic

Cycle related Corner Case Mitigation

- Reduce specification complexity if possible
- Do not introduce implementation corners

➔ Architecture, structure, architecture, structure, architecture, ...

  - Allow proper upfront design structuring
  - All the way down to micro architecture

- Run proper and thorough Reviews
- Use structured testbenches with a good architecture
  - Overview, Readability, Maintainability
  - Detailed and simple alignment and skew control between DUT interfaces
  - Single test sequencer is required for good overview and readability
  - Simple reuse from one testbench to another

Riviera-Pro includes UVVM.

CR & FC
to be included soon

EmLogic

EmLogic

Thanks for your attention

Aldec Webinar, 14 October 2021

et@emlogic.no