



# ***bitvis***

A COMPANY IN THE ACANDO GROUP

## **UVVM**

**- The fastest growing FPGA  
verification methodology world-wide!**

*Workshop on Open Source Design Automation (OSDA) 2019*

*Please also see related conference paper:  
<https://osda.gitlab.io/19/tallaksen.pdf>*

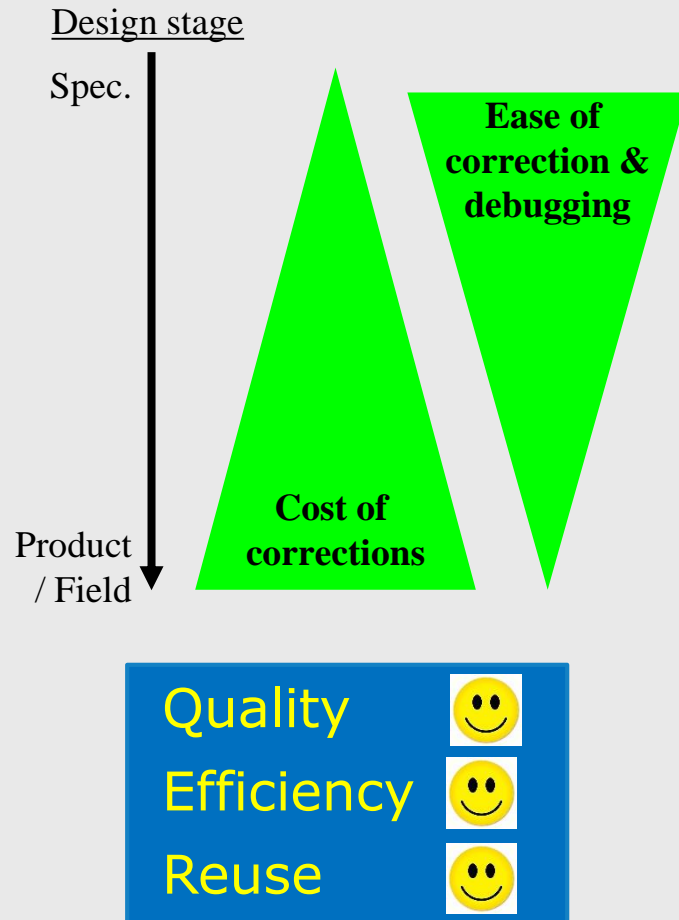
**www.bitvis.no**

**Your partner for Embedded SW and FPGA**

# Handout version

- Some slides were skipped during the presentation in order to keep to the schedule.  
These are now included (and marked as such)
- The presentation had a lot of animation to ease the understanding. This is not available in this PDF.  
If you would like to have a copy of the animated presentation (as a powerpoint-show-file), please send a request to [espen.tallaksen@bitvis.no](mailto:espen.tallaksen@bitvis.no) , and I will send it to you.
- You may download the complete UVVM from [www.github.com/UVVM](http://www.github.com/UVVM)

# Why Testbenches and Simulation?

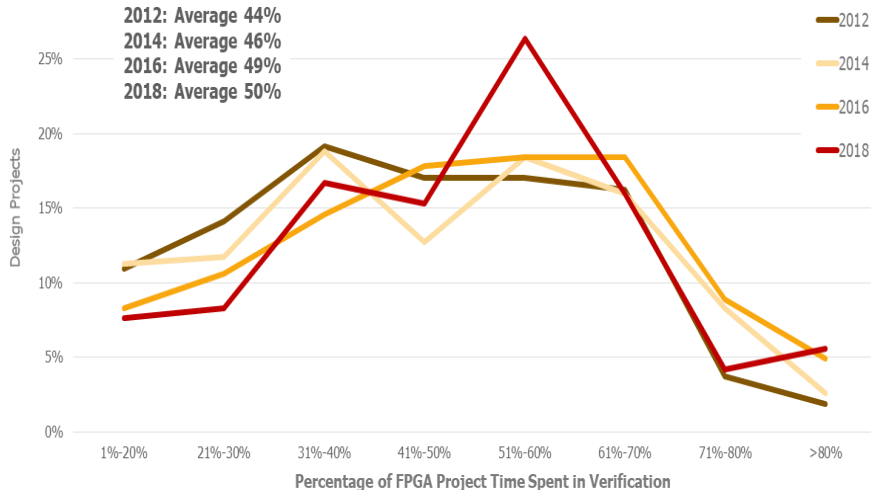


- Far more control and observability
  - ✓ Variables and intermediate signals can be viewed.
  - ✓ Environment and testdriver can also be viewed.
  - ✓ Must often coordinate I/O and internal state to verify corner cases.
  - ✓ Single stepping through code and signals is possible
  - ✓ "Embedded analysers" often sample on clock edges. Simulators show detailed signal sequences.
- Far faster iterations
  - ✓ even more important for time consuming P&R
- May have a structured bottom-up verification.
- Detect bugs that cannot or most probably will not be detected in a lab-test
  - ✓ Detect bugs in modules for functionality outside currently known scope.
  - ✓ Detect bugs that occur in abnormal situations
  - ✓ Detect bugs that are hard to provoke with current HW, SW or Test system
- Most bugs can be found with short simulations.

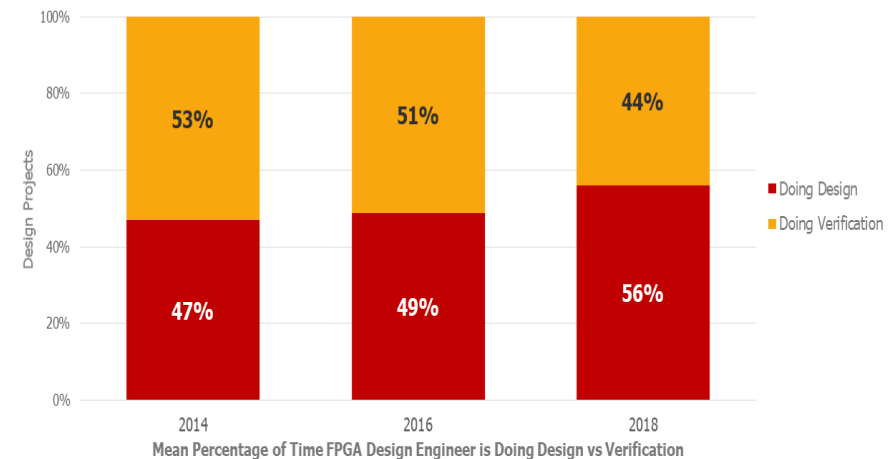
# The 2018 Wilson Research Group Functional Verification Study (1)

## Half the project time is spent in verification

### FPGA: Percentage of Project Time Spent in Verification



### FPGA: Mean % Time Design Engineer is Doing Design vs Verification

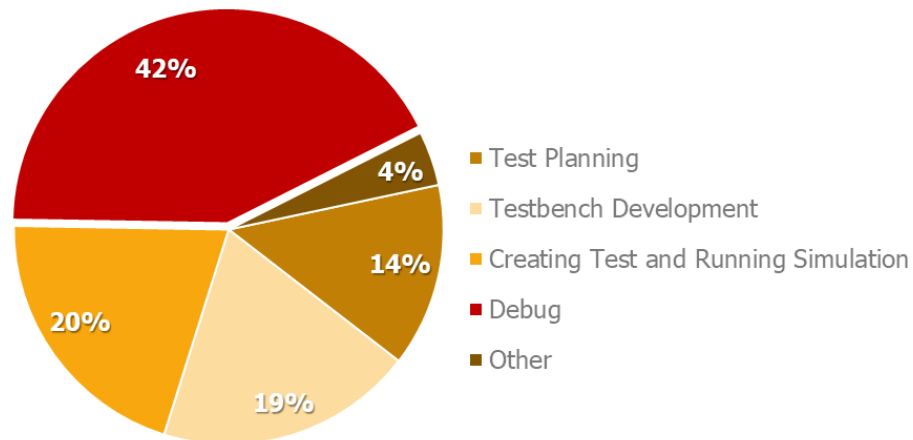


Could we be more efficient? - structured?

# The 2018 Wilson Research Group Functional Verification Study (2)

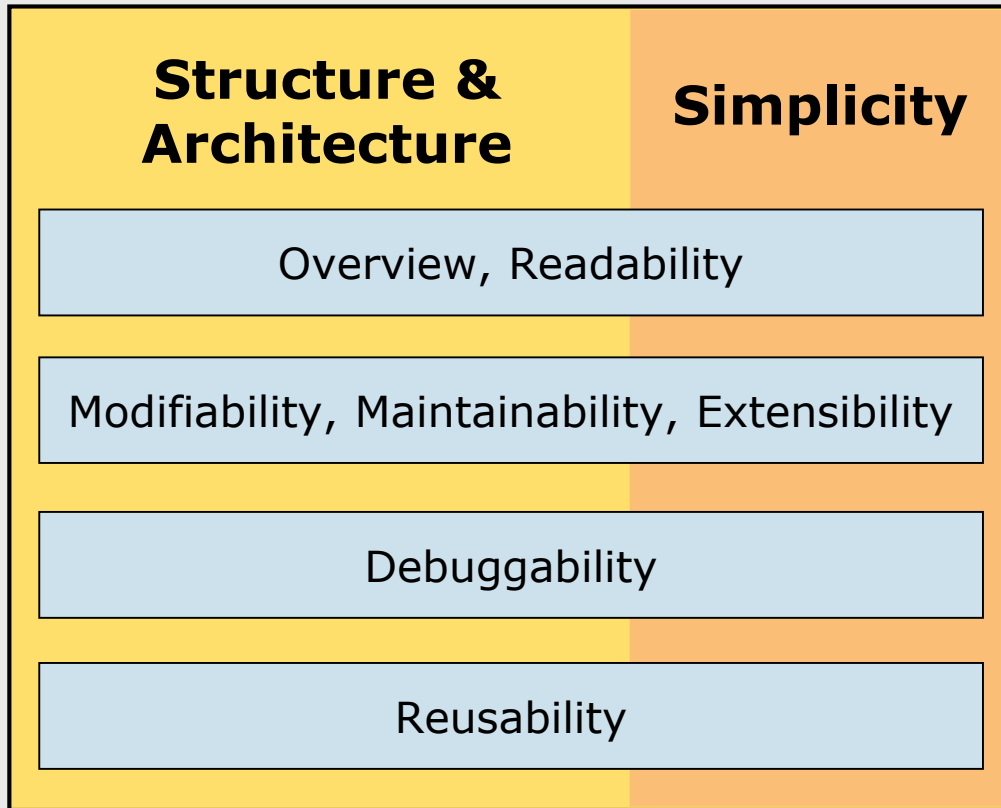
Half the verification time is spent on debugging

## FPGA: Where Verification Engineers Spend Their Time



We can definitely be more efficient! - structured!

# Quality and Efficiency enablers



Significantly affects:

- Man hours / Cost
- Schedule & TTM
- Quality & MTTF
- Product LCC
- ... Next project

Easily save 100-500 hours  
Sometimes 1000-2000 hours

Insufficient simulation will  
often cause late problems

# Why VHDL Verification?

- The most popular FPGA development language world-wide \*1
- 60% of all FPGA designer world-wide use VHDL \*1

## **For VHDL designers:**

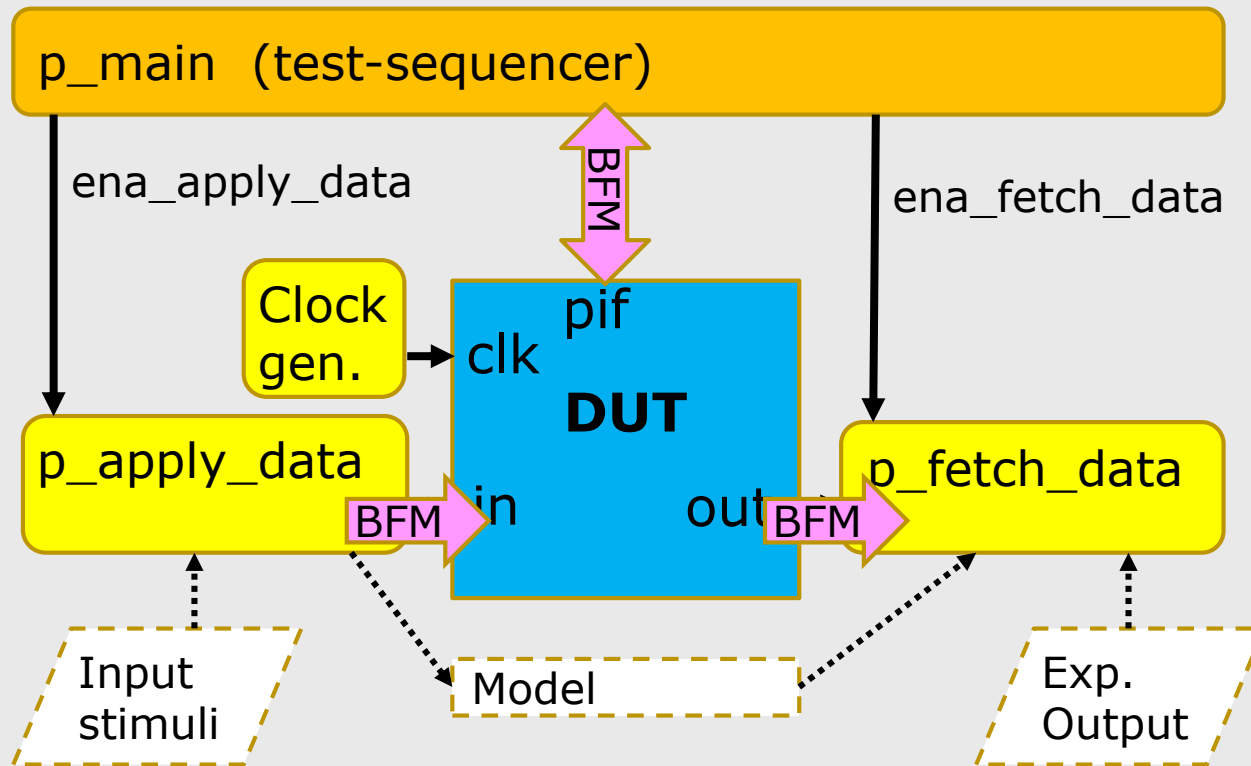
VHDL is by far the best language for verification

- ◆ The most efficient
- ◆ The least expensive

### *Note 1:*

- Numbers taken from Wilson Research 2018 (bi-annual)
- Numbers do actually go more in favour of VHDL (due to surveyee limitations)

# Simple testbench scenario



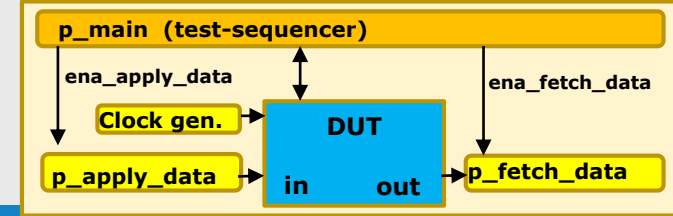
Typically applies to  
Data-path oriented  
design

Could also apply to  
Protocol oriented  
design

Control oriented  
design typically more  
complex to verify



# Achieving flexibility, readability, extendibility, ...



## ■ What is **always** required for a **any** good testbench?

- Logging - with good messages
- Alert handling - with good messages
- Checking values and time aspects
- Waiting for something to happen
- Randomisation (in many cases)

Why?

Why?

Why?

Why?

Why?



## ■ Required for both simple and advanced testbenches

- Advanced TB architectures need additional advanced structures,  
- but these are also building on the basic infrastructure.

# Using the log method

```
log(msg)  -- Simplest version of all
```

- Where? → Anywhere!

```
-- In test sequencer as a normal progress msg  
log("Checking Registers in UART");
```

```
BV: 160 ns    uart_tb    Checking Registers in UART
```

```
-- In test sequencer as a section header  
log(ID_LOG_HDR, "Check defaults for all registers");
```

```
BV:  60 ns    uart_tb    Check defaults for all registers  
BV:-----
```

Pluss lots of other log variants

# check\_value()

```
check_value(val, exp, severity, msg, [scope]) -- Simple version
```

- Checks value against expected (or boolean)
  - Triggers an **alert** if fail – and reports mismatch + message
- Overloads for sl, slv, u, s, int, bool, time
- With or without a return value (boolean OK)

```
-- E.g. inside the test sequencer
check_value(dout, x"00", ERROR, "dout must be default inactive");
```

```
BV: 60 ns  irqc_tb  check_value(slv x00)=> OK.
                        dout must be default inactive
```

```
BV:=====
BV: ERROR:
BV:      192 ns. irqc_tb
BV:           value was: 'xFF'.  expected 'x00'.
BV:           dout must be default inactive
BV:=====
```

# await\_value()

```
await_value(irq, '1', 0 ns, 2* C_CLK_PERIOD,  
            ERROR, "Interrupt expected immediately");
```

- expects (and waits for) a given value on the signal
  - inside the given time window
  - otherwise timeout - with an **alert**
  - accepts value if already present and min = 0ns

A variant on this is await\_change()

# Alerts and severities

- Severities
  - note, warning, error, failure
  - tb\_note, tb\_warning, tb\_error, tb\_failure
  - manual\_check
- All alert levels (severity levels) are counted separately
- May set\_alert\_stop\_limit(alert\_level,  $N \geq 0$ )
- May set\_alert\_attention(alert\_level, IGNORE|REGARD)
- May increment\_expected\_alerts(alert\_level, N)
- May report\_alert\_counters(VOID)

# Report summaries

report\_alert\_counters(VOID);

```
=====
BV:  ***  SUMMARY OF ALL ALERTS  ***
BV:  =====
BV:                                     REGARDED   EXPECTED   IGNORED   Comment?
BV:      NOTE           :           0           0           0         ok
BV:      TB_NOTE        :           0           0           0         ok
BV:      WARNING        :           0           0           0         ok
BV:      TB_WARNING     :           0           0           0         ok
BV:      MANUAL_CHECK   :           0           0           0         ok
BV:      ERROR          :           0           0           0         ok
BV:      TB_ERROR       :           0           0           0         ok
BV:      FAILURE        :           0           0           0         ok
BV:      TB_FAILURE     :           0           0           0         ok
BV:  =====
BV:  >> No mismatch between counted and expected serious alerts
BV:  =====
```

# More in UVVM Utility Library

- `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...

# Well Documented

## UVVM Utility Library – Quick Reference

### Checks and awaits

```
[v_bool :=] check_value(value, [exp], alert_level, msg, [...])  
[v_bool :=] check_value_in_range(value, min_value, max_value, alert_level, msg, [...])  
check_stable(target, stable_req, alert_level, msg, [...])  
await_change(target, min_time, max_time, alert_level, msg, [...])  
await_value(target, exp, min_time, max_time, alert_level, msg, [...])  
await_stable(target, stable_req, stable_req_from, timeout, timeout_from, alert_level, msg, [...])
```

### String handling

```
v_string := to_string(val)  
v_string := justify(val, justify)  
v_string := fill_string(val, fill)  
v_string := to_upper(val)  
v_character := ascii_to_char(val)  
v_int := char_to_ascii(val)  
v_natural := pos_of_left(val, search)  
v_natural := pos_of_right(val, search)
```

### 1.1 Checks and awaits

Name	Parameters and examples	Description
[v_bool :=] check_value()	<pre>val(bool), [exp(bool)], alert_level, msg, [scope, [msg_id, [msg_id_panel]]] val(slv), exp(slv), [match_strictness], alert_level, msg, [scope, [msg_id, [msg_id_panel]]] val(slv), exp(slv), [match_strictness], alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] val(u), exp(u), alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] val(s), exp(s), alert_level, msg, [scope, [radix, [format, [msg_id, [msg_id_panel]]]]] val(int), exp(int), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] val(real), exp(real), alert_level, msg, [scope, [msg_id, [msg_id_panel]]] val(time), exp(time), alert_level, msg, [scope, [msg_id, [msg_id_panel]]]</pre> <p><b>Examples</b></p> <pre>check_value(v_int_a, 42, WARNING, "Checking the integer"); v_check := check_value(v_slv5_a, "11100", MATCH_EXACT, ERROR, "Checking the SLV", "My Scope",                         HEX, AS_IS, ID_SEQUENCER, shared_msg_id_panel);</pre>	<p>Checks if <i>val</i> equals <i>exp</i>, and if values do not match. The result of the check is <i>v_bool</i>, called as a function.</p> <p>If <i>val</i> is of type <i>slv</i>, <i>unsigned</i> arguments:</p> <ul style="list-style-type: none"><li>- <i>match_strictness</i>: Specifies the match strictness, e.g. MATCH_EXACT, MATCH_OR, MATCH_AND.</li><li>- <i>radix</i>: for the vector representation, e.g. HEX, BIN, DEC, OCT, ASCII, ID_SEQUENCER, AS_IS, ID_SEQUENCER, shared_msg_id_panel.</li><li>- <i>format</i>: for the vector representation, e.g. HEX, BIN, DEC, OCT, ASCII, ID_SEQUENCER, AS_IS, ID_SEQUENCER, shared_msg_id_panel.</li></ul> <p>randomize(seed1, seed2)</p>
[tb_]error(msg, [scope])		
[tb_]failure(msg, [scope])		


### Signal generators



# How do you get started?

**A total of  
5 minutes**

## The exhaustive list of what to do:

1. Download from Github  
<https://github.com/UVVM/UVVM>  

2. Compile Utility Library
  - a) Inside your simulator go to `'uvvm_util/sim'`
  - b) execute: `'source ../script/compile_src.do'`
3. Include the library inside your testbench by adding the following lines before your testbench entity declaration:  

```
library uvvm_util;  
context uvvm_util.uvvm_util_context;
```
4. You may now enter any utility library command inside your testbench processes (or subprograms)  
e.g. `log("Hello world");`

# BFMs to handle interfaces

- Handle transactions at a higher level
  - ✓ E.g. Read, Write, Send packet, Config, etc

## BFM: **Bus Functional Model**

- A model or model set (or API) for handling transactions on a physical interface.
- Models the environment - e.g. a bus master

# BFMs to handle interfaces

- Handle transactions at a higher level
  - ✓ E.g. Read, Write, Send packet, Config, etc

Example: BFM procedure for a CPU access to a module's register

E.g. write 0xF0 into a register at address 0x22

- Models the environment - e.g. a bus master

```
cs      <= '1' ;  
we      <= '1' ;  
addr    <= x"22" ;  
data    <= x"F0" ;  
wait until rising_edge(clk) ;  
wait until falling_edge(clk) ;  
cs      <= '0' ;  
we      <= '0' ;
```

**Replaced by:**

```
write(x"22", x"F0") ;
```

**or:**

```
sbi_write(C_UART_TX, x"F0") ;
```

SBI: Simple Bus Interface

- Single cycle
- Optional ready
- Dead simple

# BFMs to handle interfaces

- Handle transactions at a higher level
  - ✓ E.g. Read, Write, Send packet, Config, etc
  - ✓ More understandable for anyone
  - ✓ Simpler code & Improved overview
  - ✓ Uniform style, method, sequence, result
  - ✓ Easy to add several very useful features

```
cs      <= '1' ;  
we      <= '1' ;  
addr    <= x"22" ;  
data    <= x"F0" ;  
wait until rising_edge(clk) ;  
wait until falling_edge(clk) ;  
cs      <= '0' ;  
we      <= '0' ;
```

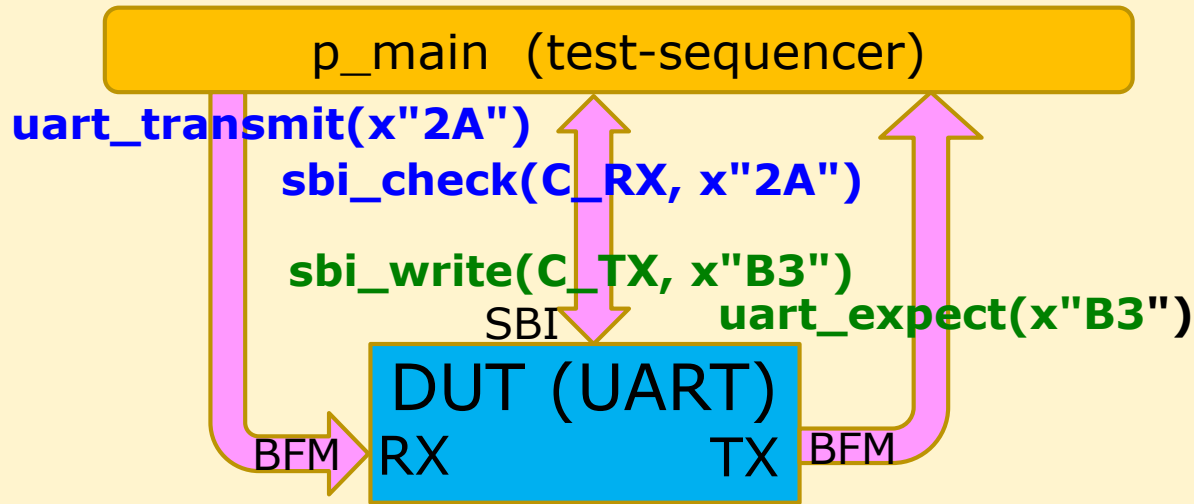
**Replaced by:**

```
write(x"22", x"F0") ;
```

**or:**

```
sbi_write(C_UART_TX, x"F0") ;
```

# Simple data communication



May use Utility Library  
and provided BFM

**Free, Open source BFM:**  
UART, AXI4-lite, SPI, I2C,  
Avalon MM, AXI4-stream,  
GPIO, SBI, ...

```
TB: 172 ns. uart_tb    uart_transmit(x2A) on UART RX
TB: 192 ns. uart_tb    sbi_check(x1, ==> x2A) completed. From UART RX
```

```
TB: 192 ns. uart_tb    sbi_write(x2, ==> xB3) completed. To UART TX
```

TB: ERROR:

```
TB:      192 ns. uart_tb
TB:          value was: 'xB2'.  expected 'xB3'.
TB:          (From uart_expect(xB3))
TB:=====
```

# Further testbench challenges

- Utility Library and BFM's are **great** for simple testbenches

- BUT

Additional challenges for more complex verification:

- Cycle related corner cases are almost never tested
- Difficult to get an overview for DUT with multiple interfaces
- Split transactions are cumbersome to control
- Difficult to synchronize stimuli/checks on multiple interfaces
- Several central sequencers often have to be coordinated
- The sequence of events is often difficult to follow
- Debugging is often terrible
- Functional coverage often too low
- Inefficient testbench reuse within a single project
- Inefficient testbench reuse from one project to another

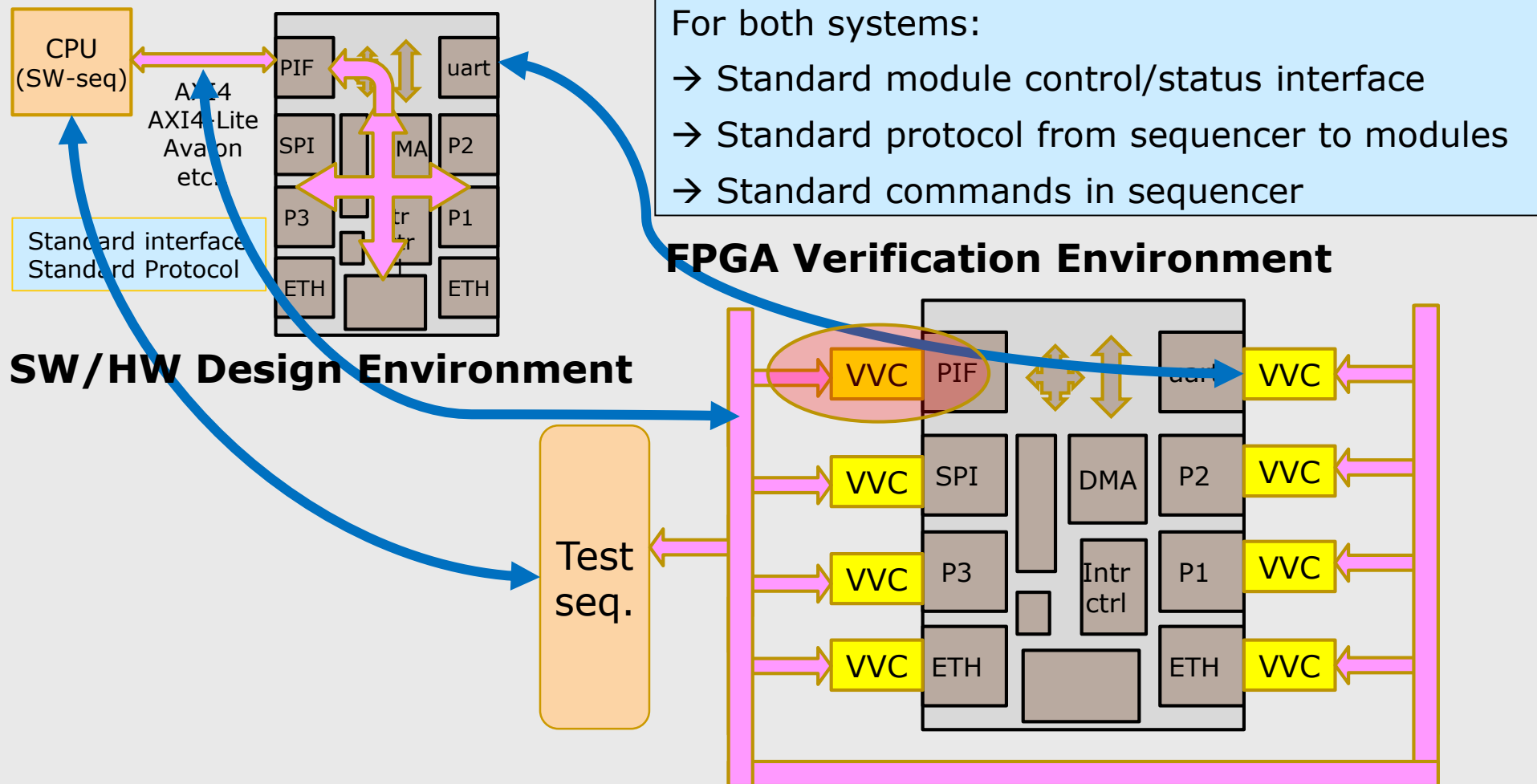
UVVM



- 
- SW command sequencer
- CPU
- AXI4  
AXI4-Lite  
Avalon  
etc..
- Standard interface  
Standard Protocol
- FPGA**
- PIF
- SPI
- P3
- ETH
- uart
- P2
- P1
- ETH

# Mirror the SW/HW interface

Added for  
handout version

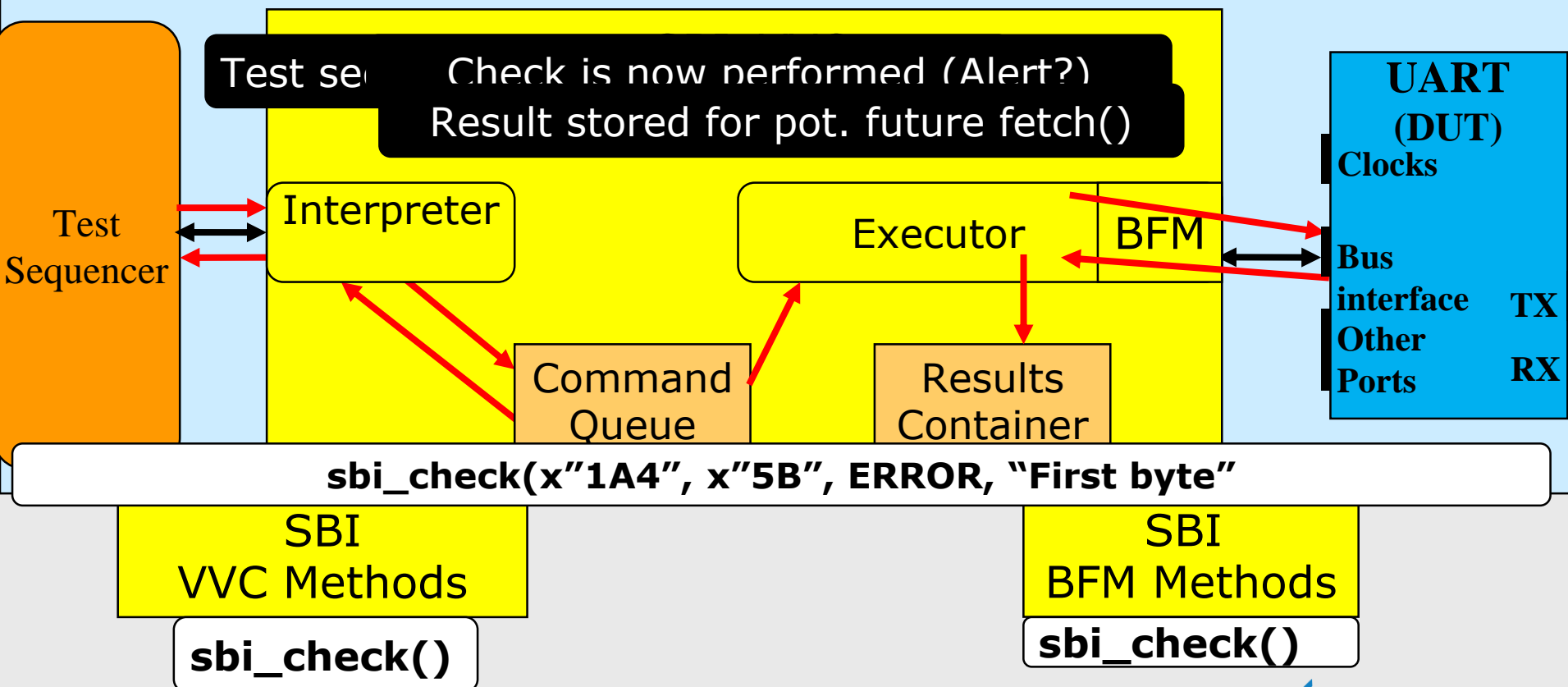




# Verification component

## Illustration of a simple check-command from sequencer

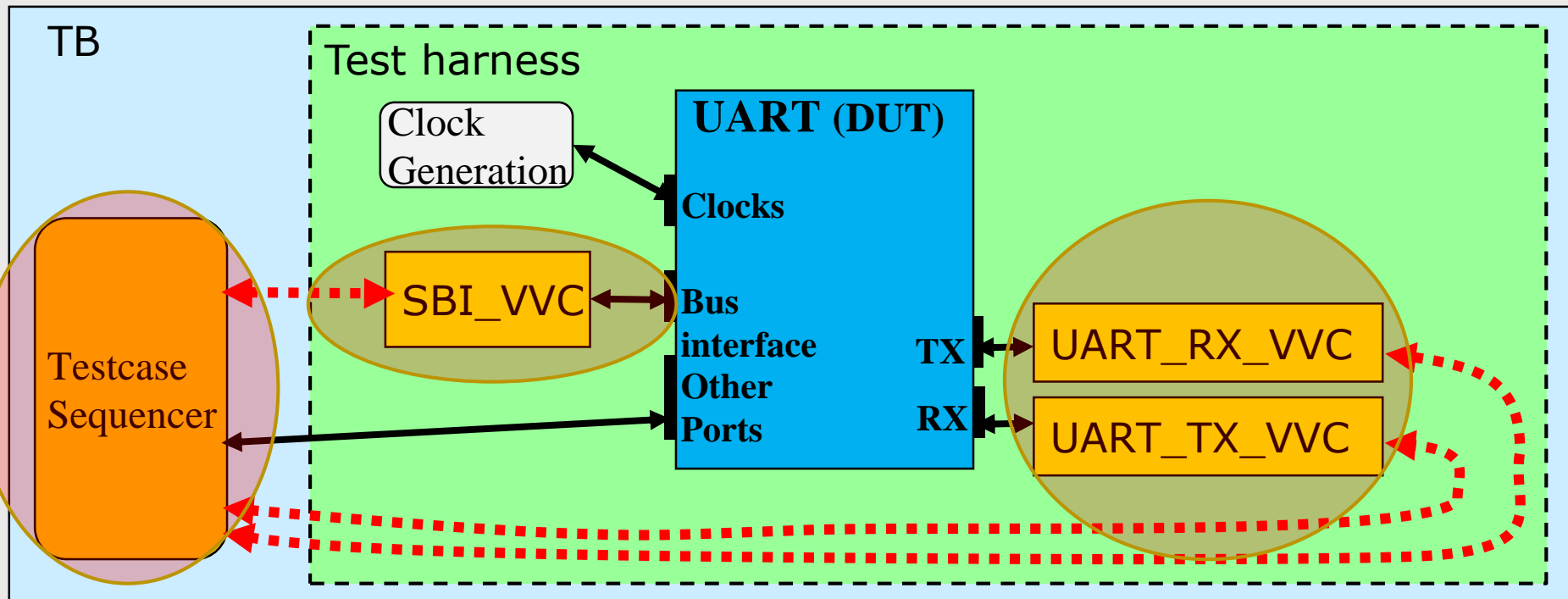
`sbi_check( SBI_VVCT, 1, x"1A4", x"5B", ERROR, "First byte")`



# Verification of more complex DUT:

## - Three main development areas

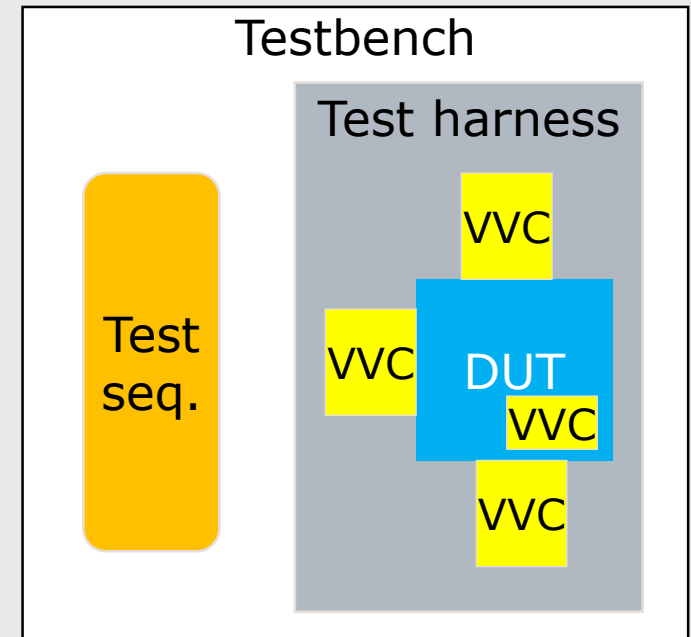
- 1: The complete Testbench with Test harness (optional hierarchy)
- 2: The Verification Components **Encapsulated BFM - plus more**
- 3: The Central Test Sequencer



# 1: The UVVM testbench/harness

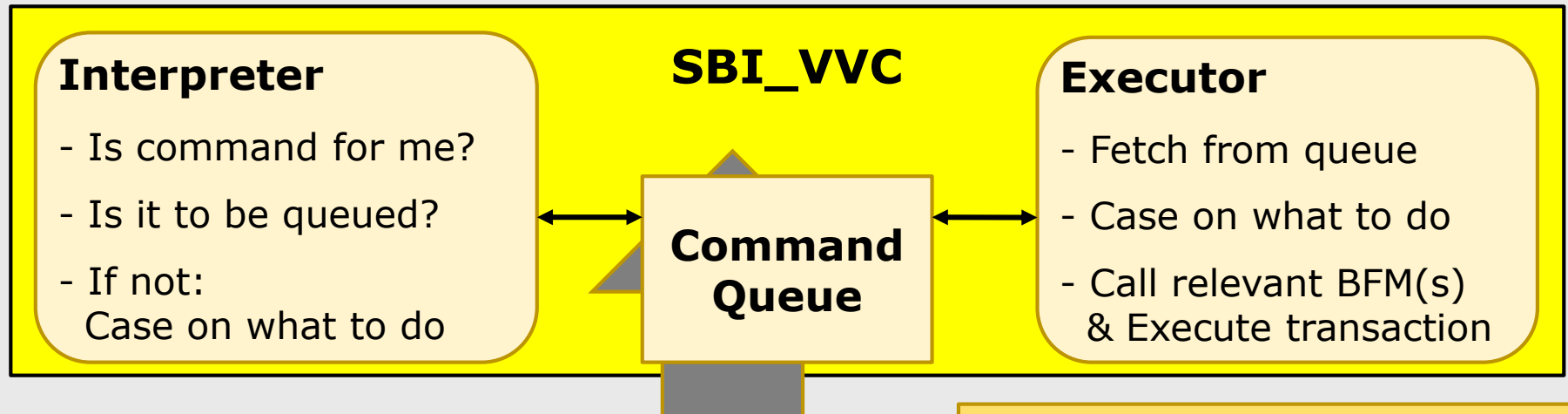
## UVVM is LEGO-like

- Build test harness
  - Instantiate DUT and VVCs
  - Connect VVCs to DUT
- Build TB with test sequencer
  - Instantiate test harness
  - Include VVC methods pkg
  - Connections included
  - No additional connections
  - VVCs could be inside DUT



- Standard global interface throughout test harness
- Standard protocol from test sequencer to VVC

## 2: VVC: VHDL Verification Component



### Same main architecture in every VVC

- >95% same code in Interpreters
- Same command queue
- 95% same code in Executors - apart from BFM calls

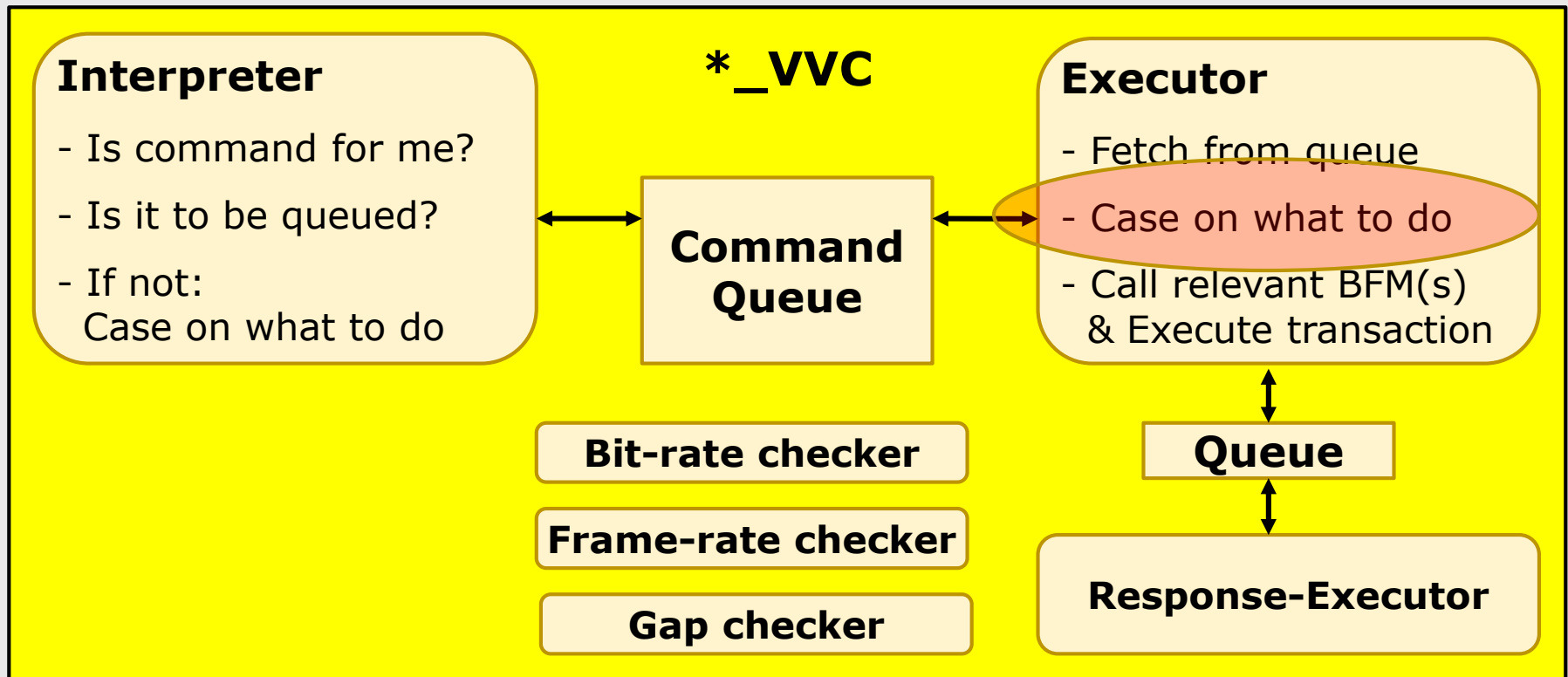
### VVC Generation

UART BFM to UART\_VVC:  
**less than 30 min**

→ Standard VVC internal architecture

## 2: VVC: VHDL Verification Component

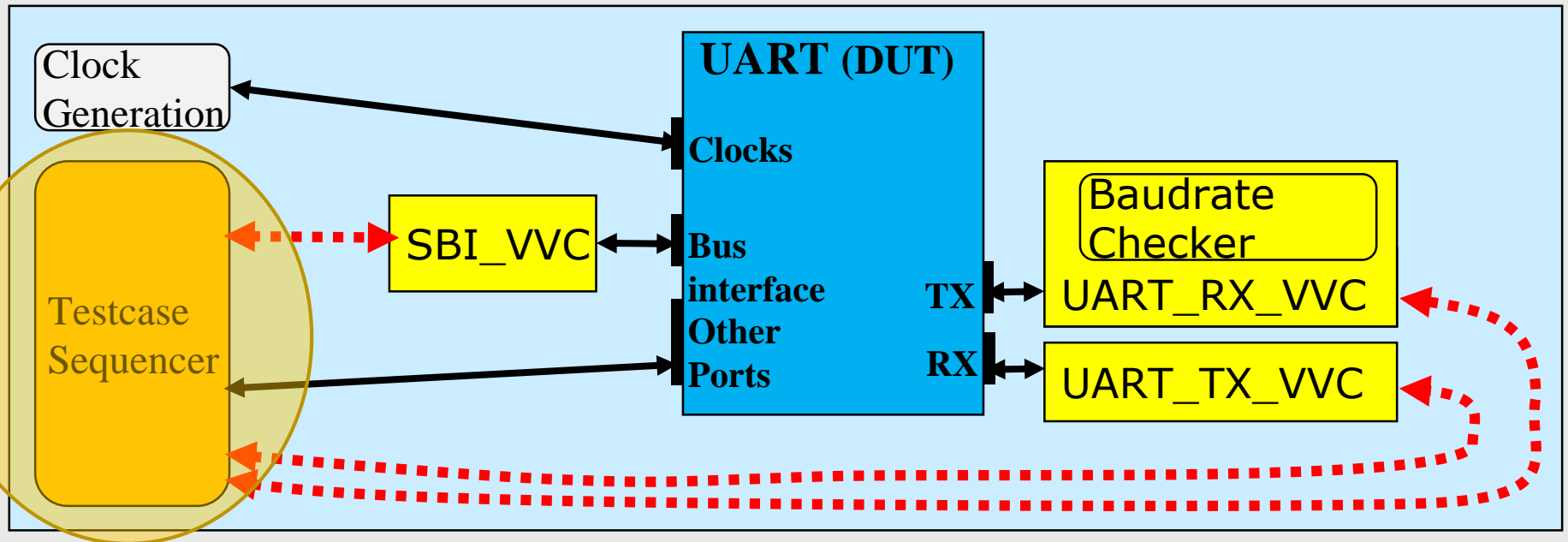
- Standard Queuing system
- Easy to handle split transactions
- Easy to add local sequencers
- Standard handling of multi-threaded interfaces
- Easy to add check of order of execution
- Standard control of parallel checkers



# 3: The test sequencer

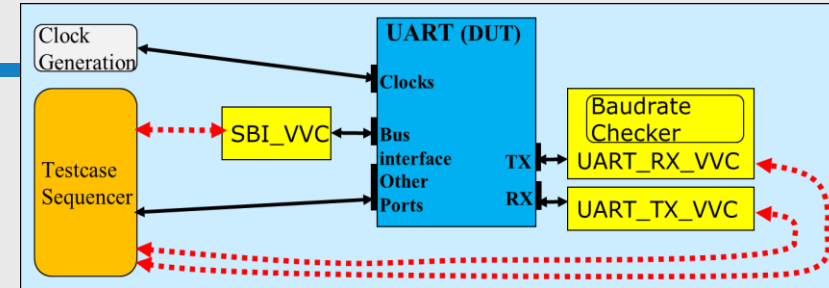
(Based on very structured TB and VVCs)

- The sequencer is the most important part of the Testbench
- Most man-hours will be (or should be) spent here
- MUST be easy to understand, modify, maintain, ....



# Command sequence - Transactions

Test sequencer issues commands  
1. Apply and check data:



```
sbi_write(SBI_VVCT,1, C_ADDR_TX_DATA, x"A0", "Send byte UART TX");  
uart_expect(UART_VVCT,1,RX  x"A0", "Check byte from UART TX");  
uart_transmit(UART_VVCT,1,TX  x"A1", "Apply byte on UART RX");  
wait for C_FRAME_PERIOD;  
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "Check UART RX byte");
```

> Standard command distribution syntax

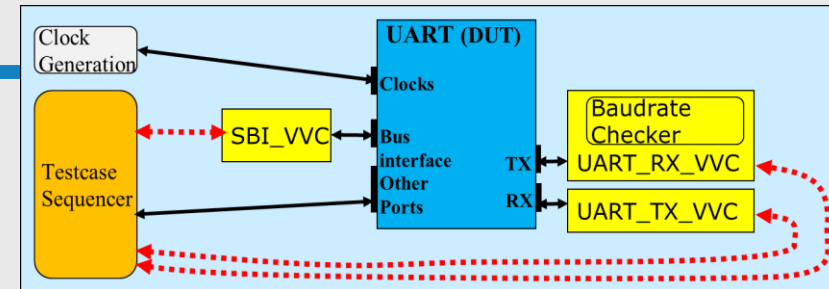
Several additional common commands for:

- Synchronization between VVCs
- Controlling the VVC behaviour and command flow to VVC

# Commands for synchronization

Included for  
handout version

Test sequencer issues commands



```
await_value(rx_empty, '0', 0, 12*bit_period, ERROR, message);
```

```
insert_delay(SBI_VVCT,1, 2 * C_CLK_PERIOD);
```

```
await_completion(UART_VVCT,1,RX, 1 us, "Finish before .....");
```

```
await_unblock_flag("my_flag", 100 ns, "waiting for my_flag")
```

```
await_barrier(global_barrier, 100 us, "waiting for global barrier")
```

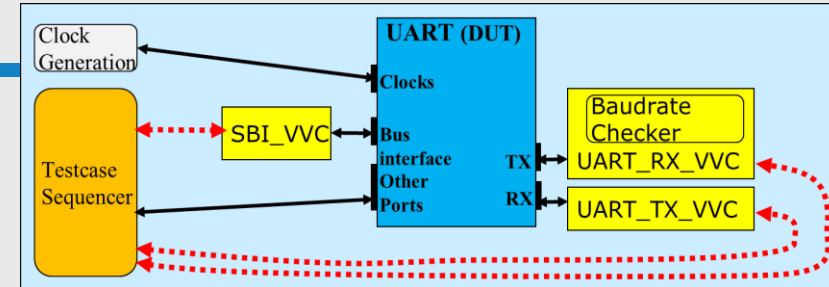
- Standard synchronization between any process or VVC
- Standard timeout and messaging



# Commands for VVC control

Included for  
handout version

Test sequencer issues commands



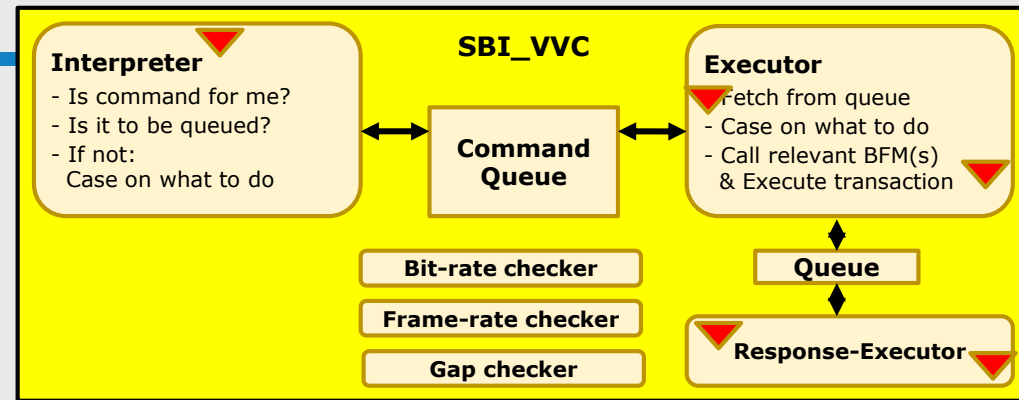
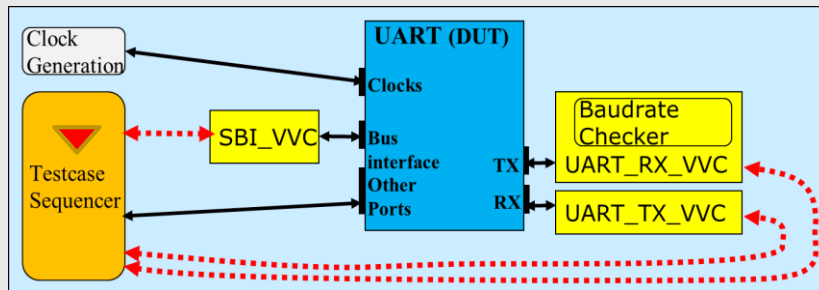
```
flush_command_queue(SBI_VVCT, 1, "Flushing command queue");  
  
fetch_result(SBI_VVCT, 1, v_idx, v_data, v_ok, "Fetching data");  
  
terminate_current_command(SBI_VVCT, 1, "Terminating command");  
  
get_last_received_cmd_idx(SBI_VVCT, 1);  
  
terminate_all_commands (VVC_BROADCAST, "Terminating all commands");
```

- Standard set of common commands for all VVCs
- Standard multicast and broadcast of common commands

# Debugging Commands and new VVCs

Included for handout version

Included for handout version



- Debugging TB is often more work than debugging the DUT...
- May follow the command through from test sequencer to execution
  - And automatically print out logs - just by enabling verbosity ▼

```
2045ns TB seq. (uvvm) ->uart_transmit(UART_VVC,1,TX, x"AA"): . [15]
2045ns UART_VVC,1,TX    uart_transmit(UART_VVC,1,TX, x"AA"). Command received [15]
2045ns UART_VVC,1,TX    uart_transmit(UART_VVC,1,TX, x"AA") Will be executed [15]
3805ns UART_VVC,1,TX    uart transmit(x"AA") completed. [15]
```

→ Standard debugging structure  
→ Standard debugging control

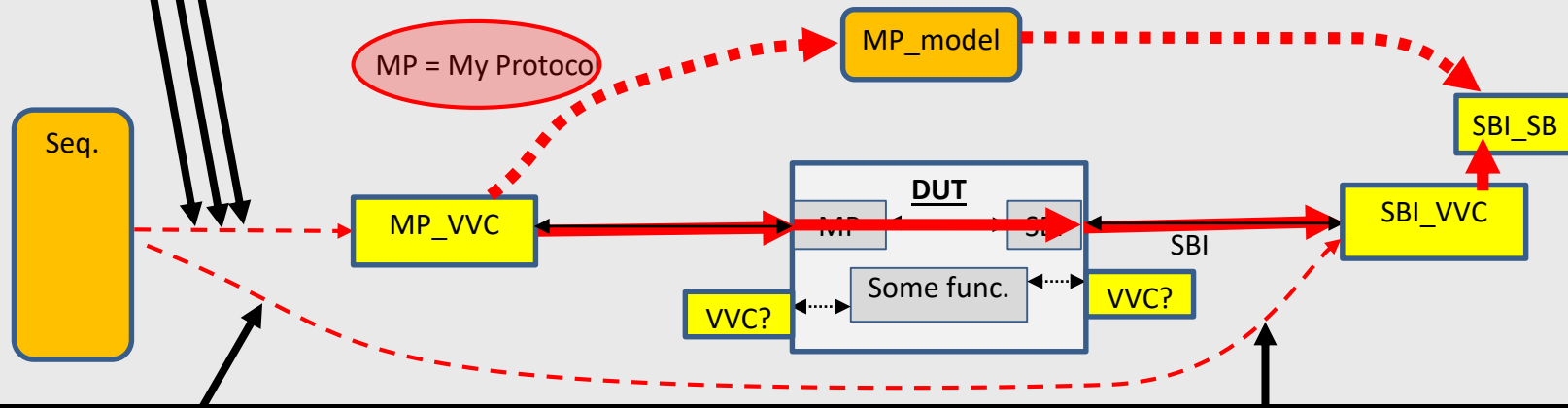
- ESA (European Space Agency) project on new UVVM extensions
- Intention: Improve FPGA quality and verification efficiency
- The extensions
  - Scoreboarding
  - Monitors
  - Controlling randomisation and functional coverage
  - Error injection
  - Local sequencer
  - Watchdog
  - Controlling property checkers
  - Req. vs Verif Matrix (Test coverage)

## - Using Scoreboards, VVCs and Models



```
uart_transmit(UART_VVCT,1,TX,  RANDOM_STIM, 256, "Transmit 256 rand bytes");
```

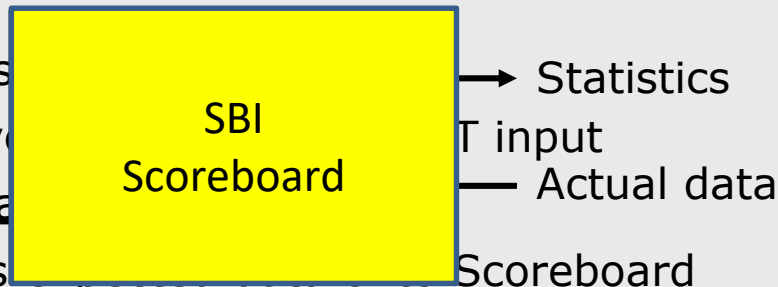
```
uart transmit(UART VVCT,1,TX,  x"42", "Transmit single byte");
```



```
sbi_read( SBI_VVCT, 1, C_UART_RX_REG, x"42", "Receive byte & send to Scoreboard");
```

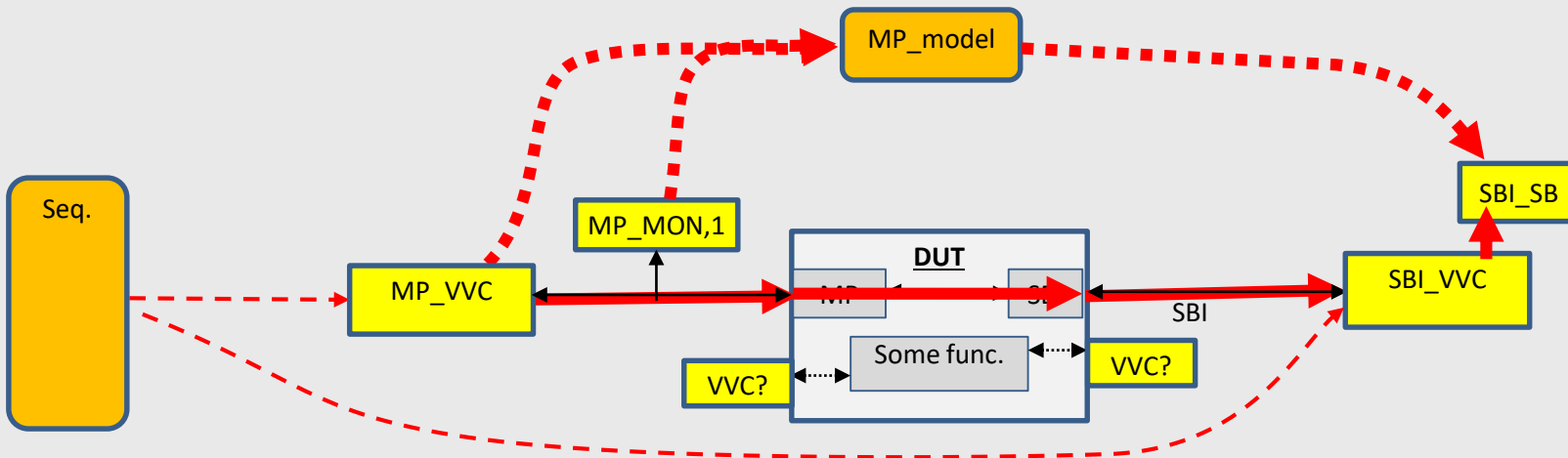
- **Model**

- Models
- Receive
- General
- Passes



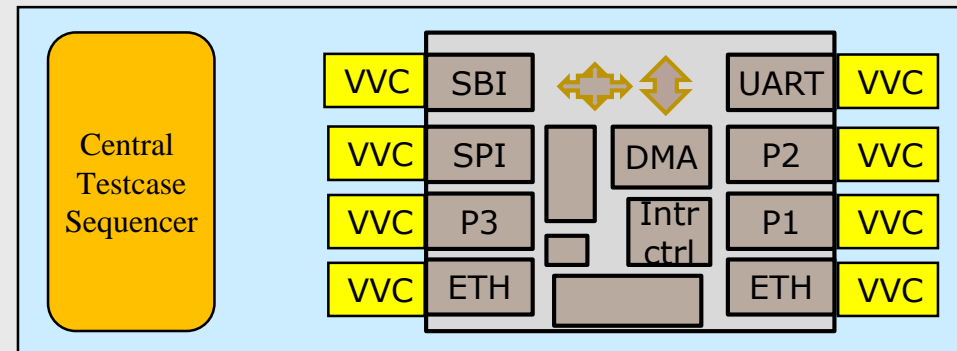
## Scoreboard

- ➔ Autonomous checks
- ➔ Pass vs Fail
- ➔ Error++ count
- ➔ Multiple statistics



# UVVM: Structure & Overview & Reuse

- Lego-like Test harness
- Reusable VVCs
- Reusable VVC structure
- Simple synchronisation
- handle any number of interfaces in a structured manner
- Clear sequence of event - almost like pseudo code
- Test cases are simple to understand
- simple debugging of TB and DUT



**Non UVVM BFM and VVCs may easily be wrapped to UVVM**

**UVVM BFM and VVCs may be used anywhere - exactly as is**

## UVVM

### Wouldn't it be nice if we could ...

- handle any number of interfaces in a structured manner?
- reuse major TB elements between module TBs?
- reuse major module TB elements in the FPGA TB?
- read the test sequencer almost as simple pseudo code?
- recognise the verification spec. in the test sequencer?
- understand the sequence of event
  - just from looking at the test sequencer
- allow simple debugging of TB and DUT



# Benefits of standardisation

- even more important for Open source..

- Same simple TB architecture independent of designer
- Same VVC architecture independent of designer
  - And almost independent of Interface
- Same commands from one VVC to another
  - Same behaviour and response from one VVC to another
  - Even simple for SW and HW designers to write and understand
- Easy to make new VVCs
  - And for others to use it - in all different ways
- Established debug-mechanisms and support
  - Allows much faster and better debugging
- Same synchronization mechanism between any VVC and TB
- Easy to reuse major TB parts from one TB to another
- Easy to share VVCs between **anyone**



# UVVM – Used world-wide

- UVVM is used by
  - ♦ 10% of all FPGA designers world-wide \*1
  - ♦ (VHDL used by >60% world-wide. 80-90% in Europe)
  - ♦ → UVVM: Used by approx 20% of all VHDL FPGA designers
- From almost zero 3 years ago...

**→ Fastest growing verification methodology in the world**

Recommended by Doullos for Testbench Architecture

ESA project to extend the UVVM functionality

\*1: According to Wilson Research, October 10, 2018 (Survey executed spring 2018)

# Summary

UVVM is Open Source

UVVM runs on GHDL (open source)

- Huge improvement potential for more structured FPGA verification  
→ UVVM is unlocking this improvement potential
- Massive improvement potential for testbench reuse  
→ UVVM is a game changer for efficient reuse
- Most testbenches are difficult to understand  
→ UVVM: Easily understandable, maintainable, extensible
- There has been no standardisation for VHDL testbenches  
→ UVVM standardises Test harness, VVCs and Commands  
→ UVVM standardisation does not result in any lock-in

Testbench standardisations allow cooperation and compatibility

ESA project is extending UVVM

Community VVCs soon?

UVVM may save 1000-2000 hours on a complex project

# 3-day course: **Advanced VHDL Verification – Made simple**

Included for  
handout version

**Achieve the key aspects for ANY good testbench:**

**Overview - Readability - Extensibility - Maintainability - Reuse**

- Using sub-programs and other important VHDL constructs for verification
- Making self-checking testbenches
- Using logging and alert handling
- Applying value and stability checkers and waiting with a timeout for events
- Making your own BFM – and adding features to speed up verification and debugging
- Making directed and constrained random tests – knowing where to use what - or a mix
- Learning to use OSVVM randomization and functional coverage
- Applying OSVVM coverage driven tests in a controlled manner
- Using verification components and advanced transactions (TLM) for complex scenarios
- Target data and cycle related corner cases and verifying them
- Learning to use UVVM to speed up testbench writing and the verification process

**Making an easily understandable and modifiable testbench even for really complex verification – and do this in a way that even SW and HW developers can understand them.**

More info under <https://bitvis.no/course-calendar/>