UVVM

The main benefits of the world's #1 VHDL Verification Methodology

By Espen Tallaksen, Director FPGA and Space, EmLogic

EmLogic
The Norwegian Embedded Systems & FPGA Design Centre

SIEMENS

Agenda

- Introduction
- The basics of UVVM for simple testbenches
- The basics of UVVM for advanced testbenches
- The main benefits of UVVM
- Why UVVM is better

What is UVVM?

- Number 1 world-wide for VHDL verification *1
- By far the fastest growing (indep. of lang.) *1

- Very structured infrastructure and architecture
  - Simplicity where it matters the most
  - → Significantly improves Verification Efficiency
  - → Assures a far better Design Quality
  - → Unique Reuse friendliness
- Extremely fast adoption by the world-wide VHDL community
- Recommended by Doulos for Testbench architecture
- Supported by more and more EDA vendors
- ESA projects to extend the functionality

Simple as default
Advanced when needed

*1: According to The Wilson Research Group Functional Verification Study from September 2020

The full overview…

… is not possible to give in this short presentation, but…

- You can find significantly more details in my previous free webinars for Mentor/Siemens and Trias:
  - *1. An introduction to efficient VHDL verification - using the open source UVVM*
    https://trias-mikro.de/webinars/an-introduction-to-efficient-vhdl-verification-using-the-open-source-uvvm/
  - *2. UVVM – Advanced VHDL Verification – Made simple*
    https://trias-mikro.de/webinars/uvvm-advanced-vhdl-verification-made-easy/
  - *3. Modern VHDL testbenches.*
    An AXI4-stream example, First dead simple, then advanced – as simple as possible
    https://trias-mikro.de/webinars/modern-vhdl-testbenches-an-axi-stream-example-first-dead-simple-then-advanced-as-simple-as-possible/
- I can send you a PDF of the presentations on request (espen.tallaksen@emlogic.no)
- Above webinar references is used in some of the following slides. See webinar ?

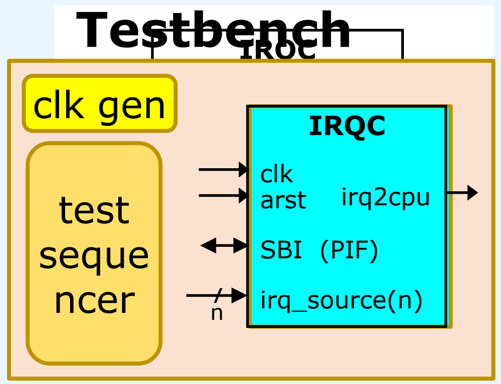
EmLogic SIEMENS

The basics of UVVM
- For simple testbenches

Typical simple verif. scenario
- a low complexity interrupt controller

```
clock_generator(clk, GC_CLK_PERIOD);
```

```
log(ID_LOG_HDR, "Started simulation of IRQC_TB");
...
check_value(irq2cpu, '0', "irq2cpu default inactive");
...
check_stable(irq2cpu, now – v_reset_time);
...
gen_pulse(irq_source(2), '1', clk_period, "Set source 2 for clock period");
...
await_value(irq2cpu, '1', 0 ns, 2* C_CLK_PERIOD,
            "Interrupt expected immediately");
...
-- Register writes and reads – via BFMs
sbi_write(C_ADDR_IER. x"1F“, “Enable all interrupts”);
...
report_alert_counters(FINAL);
```

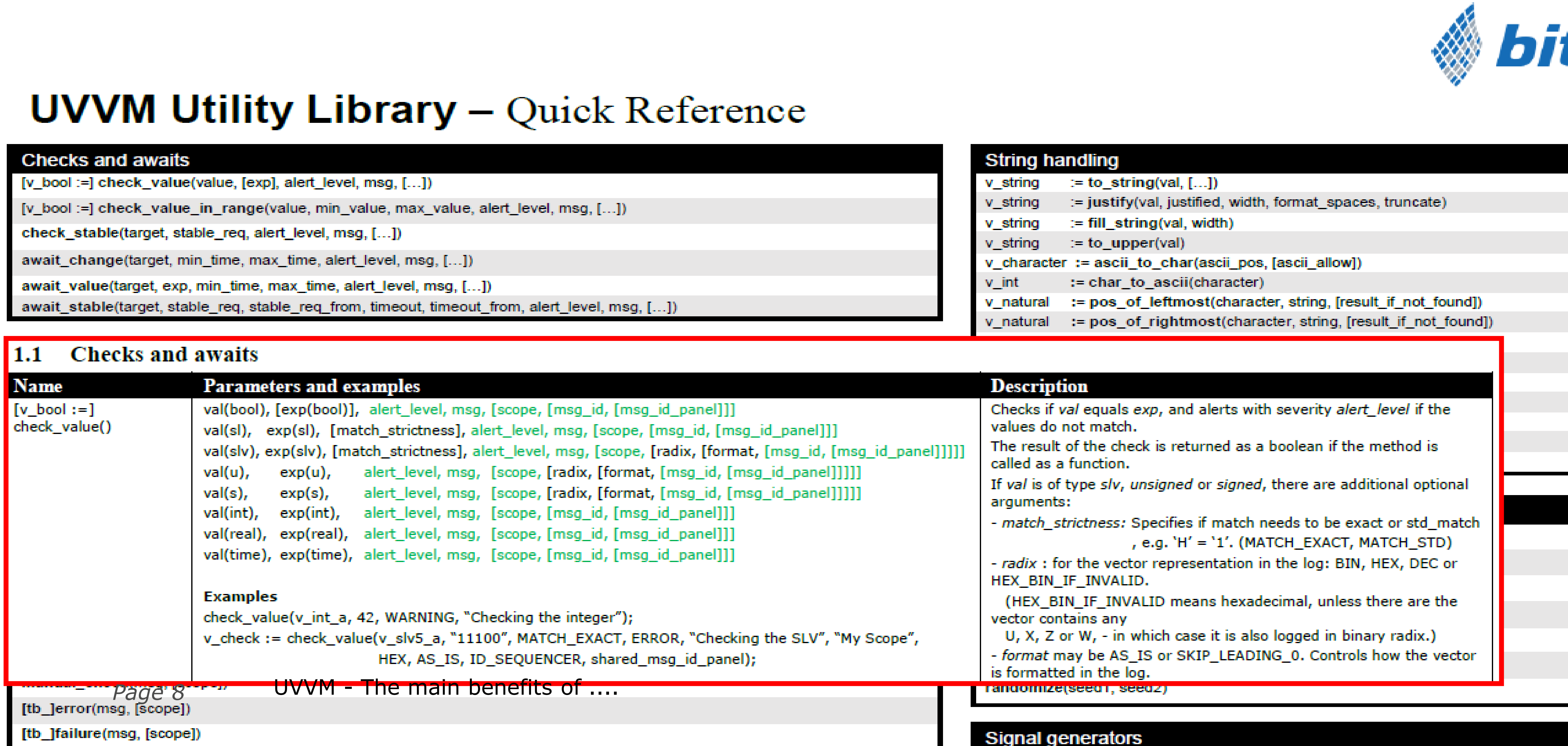


All procedures with:
- Positive acknowledge If wanted
- Alert message and mismatch report
- Alert count and ctrl

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, […])

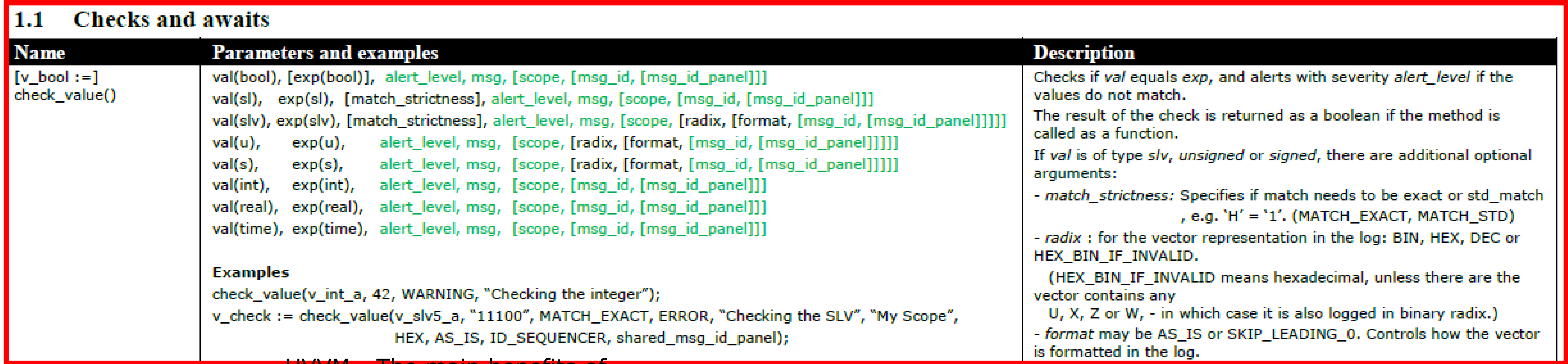
[tb_]**error**(msg, [scope])

[tb_]**failure**(msg, [scope])

String handling	
v_string	:= **to_string**(val, […])
v_string	:= **justify**(val, justified, width, format_spaces, truncate)
v_string	:= **fill_string**(val, width)
v_string	:= **to_upper**(val)
v_character	:= **ascii_to_char**(ascii_pos, [ascii_allow])
v_int	:= **char_to_ascii**(character)
v_natural	:= **pos_of_leftmost**(character, string, [result_if_not_found])
v_natural	:= **pos_of_rightmost**(character, string, [result_if_not_found])

randomize(seed1, seed2)

Signal generators

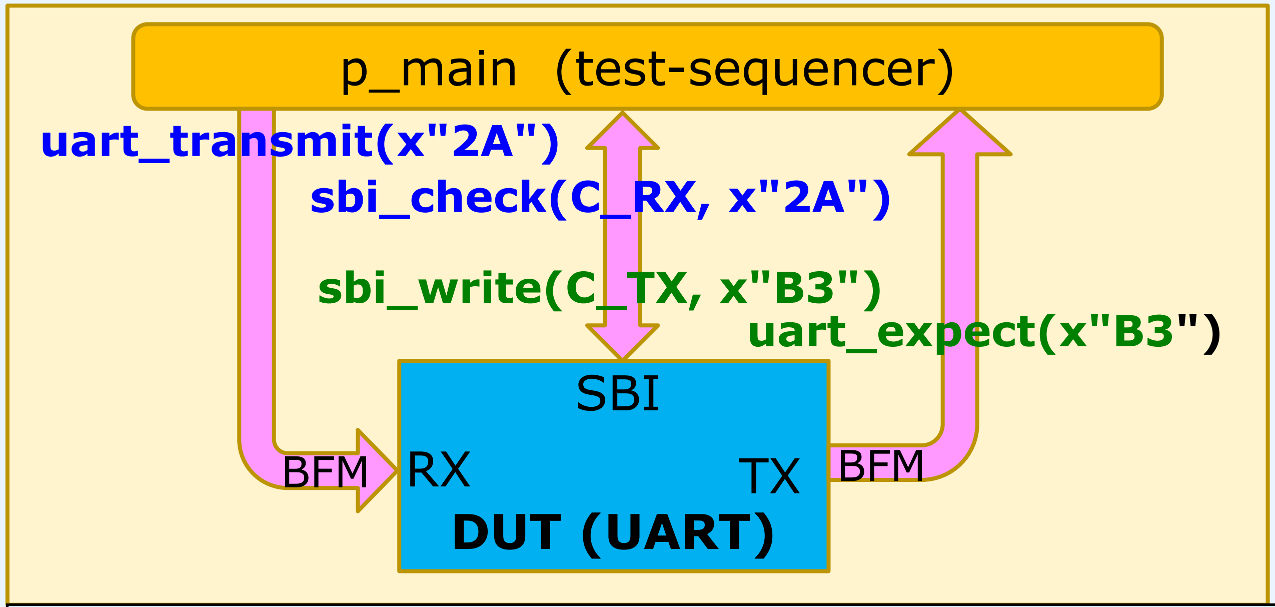


1.1 Checks and awaits

Name	Parameters and examples	Description

[v_bool :=] check_value()	val(bool), [exp(bool)], alert_level, msg, [scope, [msg_id, [msg_id_panel]]]
	val(sl), exp(sl), [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]]]
	
	Examples
	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);
		Checks if *val* equals *exp*, and alerts with severity *alert_level* if the values do not match.
		The result of the check is returned as a boolean if the method is called as a function.
		If *val* is of type *slv*, *unsigned* or *signed*, there are additional optional arguments:
		- *match_strictness:* Specifies if match needs to be exact or std_match , e.g. 'H' = '1'. (MATCH_EXACT, MATCH_STD)
		- *radix* : for the vector representation in the log: BIN, HEX, DEC or HEX_BIN_IF_INVALID.
		(HEX_BIN_IF_INVALID means hexadecimal, unless there are the vector contains any U, X, Z or W, - in which case it is also logged in binary radix.)
		- *format* may be AS_IS or SKIP_LEADING_0. Controls how the vector is formatted in the log.

Simple data communication



```
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:===============================================================
```

sbi_check() =
→ sbi_read()
→ compare

uart_expect() =
→ uart_receive()
→ compare
→ repeat until match
(choose max iterations)
(default 1. I.e. first byte)

May use Utility Library and provided BFMs

Free, Open source BFMs:
AXI4, AXI4-lite, SPI, I2C, Avalon MM, AXI4-stream, Avalon stream, UART, SBI, GPIO, GMII, RGMII, …

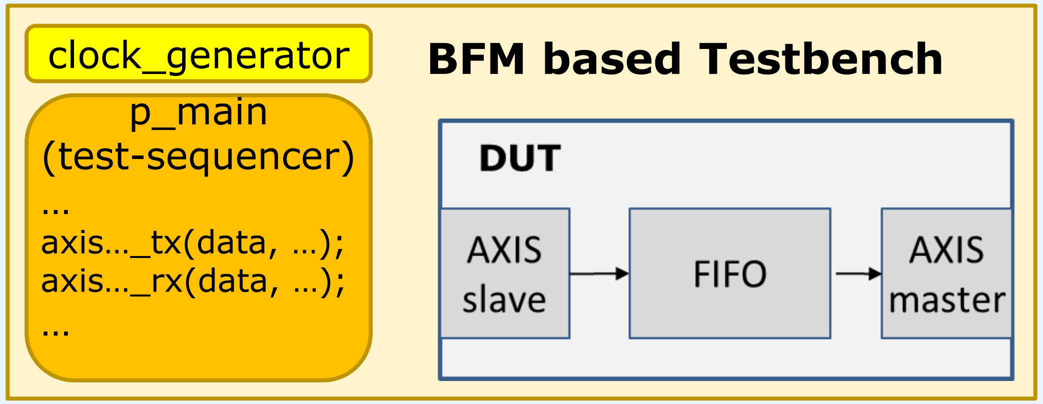
Quick References provided

EmLogic SIEMENS

AXI-stream BFM based TB

- as simple as possible

See webinar 3



UVVM_Light (from github)

uvvm_util (library)

```
log, check_value, await_value, etc…
clock_generator( )
axistream_transmit(data, ...)    (procedure)
axistream_receive(data, ...)     (procedure)
axistream_expect(data, ...)      (procedure)
etc…
```

```
axistream_transmit(v_byte_array, msg, clk, m_axis);
```

- No test harness (for simplicity)
- Sequencer has direct access to DUT signals
  - Thus BFMs from p_main can also see the DUT signals

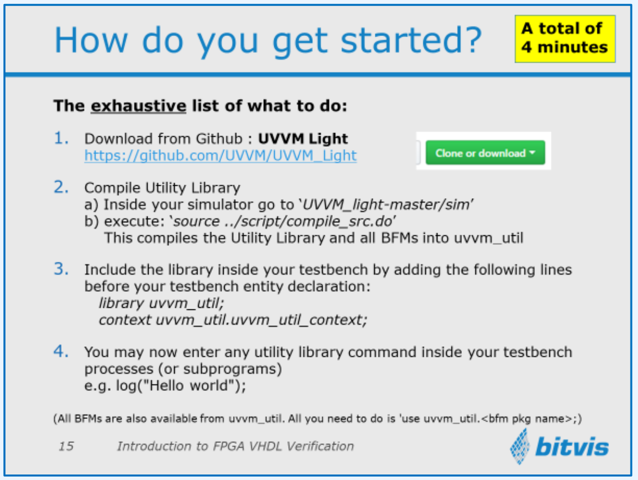
Only need to download from Github (clone or zip) and compile (total 5 min)

- Simplified UVVM
  - For simple usage
- Subset of UVVM
  No VVCs or VCC support
- All BFMs in the same directory and library

EmLogic SIEMENS

More details on Intro to UVVM

See webinars 1,2



How do you get started?

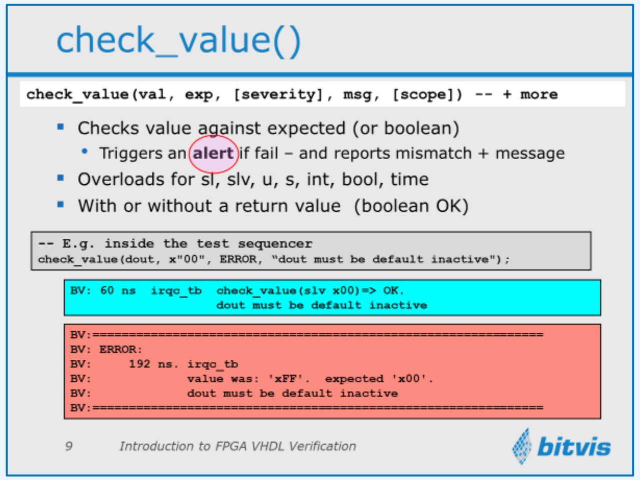
A total of 4 minutes

The <u>exhaustive</u> list of what to do:

1. Download from Github : **UVVM Light**
   https://github.com/UVVM/UVVM_Light
   Clone or download
2. Compile Utility Library
   a) Inside your simulator go to '*UVVM_light-master/sim*'
   b) execute: '*source ../script/compile_src.do*'
   This compiles the Utility Library and all BFMs into uvvm_util
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");

(All BFMs are also available from uvvm_util. All you need to do is 'use uvvm_util.<bfm pkg name>;)

15 *Introduction to FPGA VHDL Verification* bitvis



check_value()

```
check_value(val, exp, [severity], msg, [scope]) -- + more
```

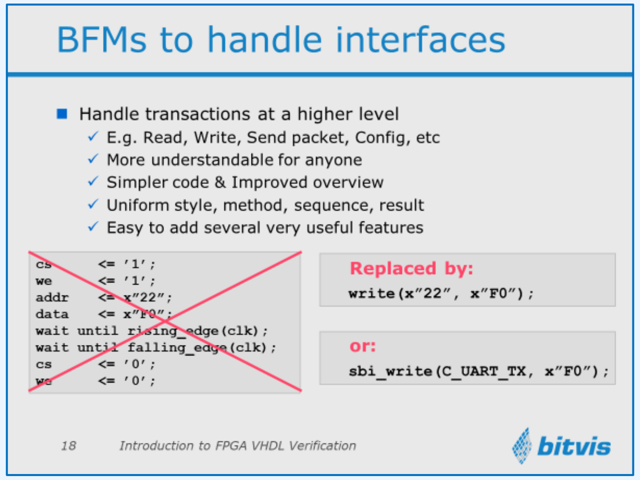
- Checks value against expected (or boolean)
  - Triggers an **alert** if fail – and reports mismatch + message
- Overloads for sl, slv, u, s, int, bool, time
- With or without a return value (boolean OK)

```
-- 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:============================================================
```

9 *Introduction to FPGA VHDL Verification* bitvis



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';
```

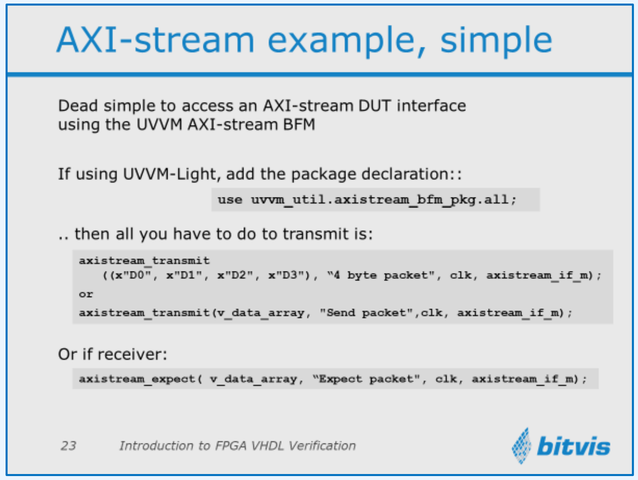
Replaced by:

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

or:

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

18 *Introduction to FPGA VHDL Verification* bitvis



AXI-stream example, simple

Dead simple to access an AXI-stream DUT interface using the UVVM AXI-stream BFM

If using UVVM-Light, add the package declaration::

```
use uvvm_util.axistream_bfm_pkg.all;
```

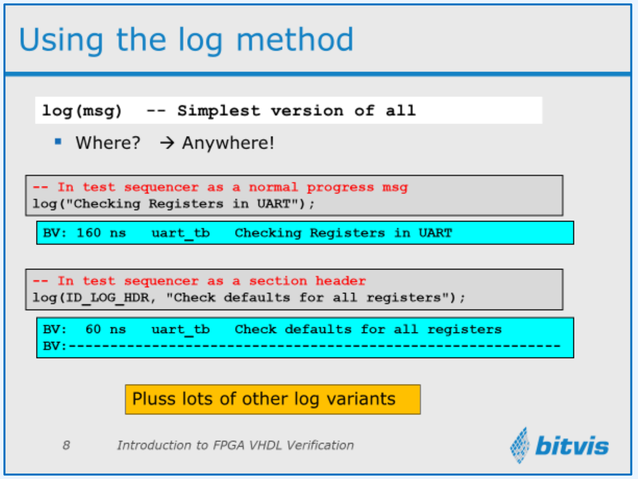
.. then all you have to do to transmit is:

```
axistream_transmit
   ((x"D0", x"D1", x"D2", x"D3"), "4 byte packet", clk, axistream_if_m);
or
axistream_transmit(v_data_array, "Send packet",clk, axistream_if_m);
```

Or if receiver:

```
axistream_expect( v_data_array, "Expect packet", clk, axistream_if_m);
```

23 *Introduction to FPGA VHDL Verification* bitvis



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");
```

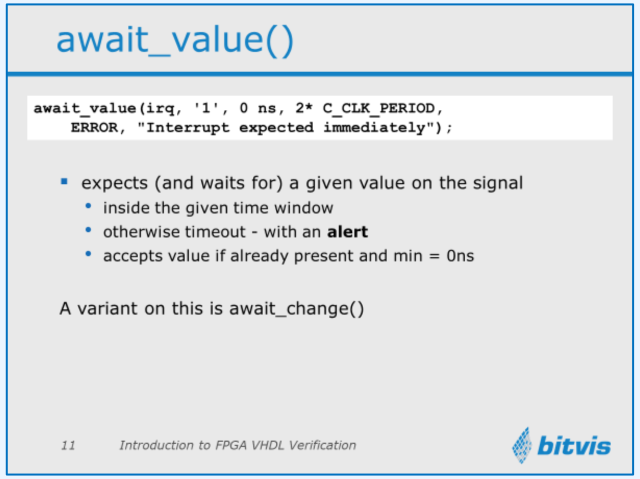
```
BV: 160 ns   uart_tb   Checking Registers in UART
```

```
-- 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

8 *Introduction to FPGA VHDL Verification* bitvis



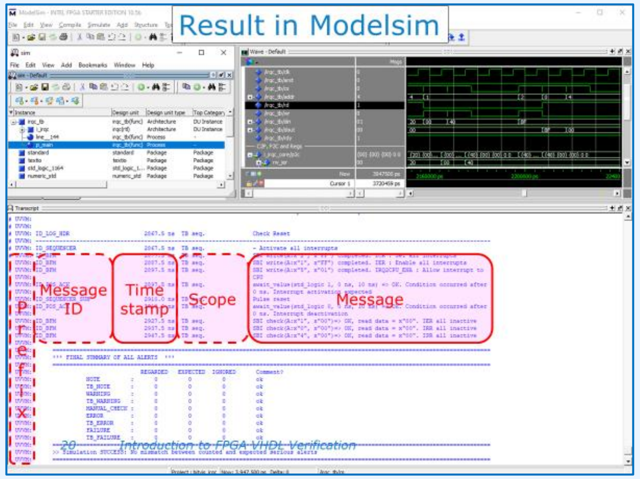
await_value()

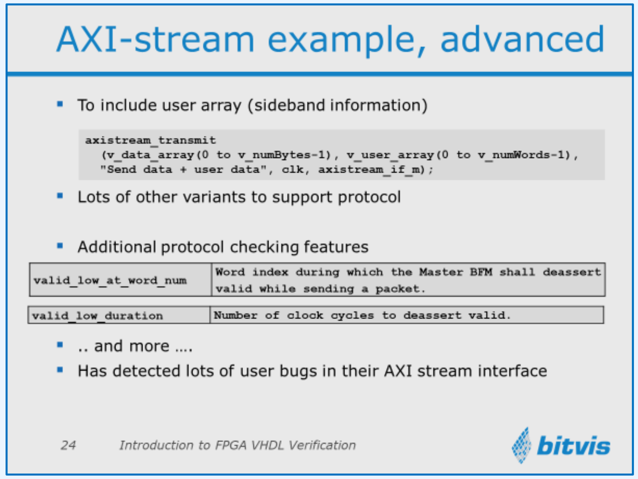
```
await_value(irq, '1', 0 ns, 2* C_CLK_PERIOD,
    ERROR, "Interrupt expected immediately");
```

- 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()

11 *Introduction to FPGA VHDL Verification* bitvis





AXI-stream example, advanced

- To include user array (sideband information)

```
axistream_transmit
  (v_data_array(0 to v_numBytes-1), v_user_array(0 to v_numWords-1),
  "Send data + user data", clk, axistream_if_m);
```

- Lots of other variants to support protocol
- Additional protocol checking features

valid_low_at_word_num	Word index during which the Master BFM shall deassert valid while sending a packet.
valid_low_duration	Number of clock cycles to deassert valid.

- .. and more ....
- Has detected lots of user bugs in their AXI stream interface

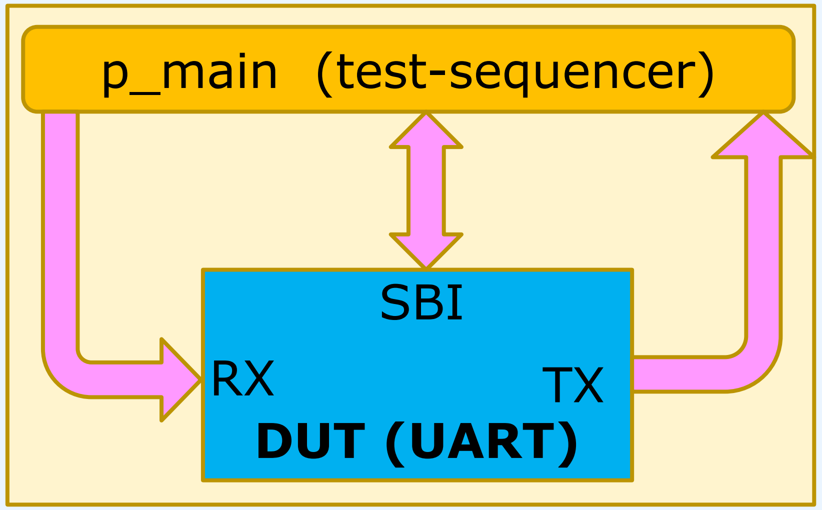
24 *Introduction to FPGA VHDL Verification* bitvis

EmLogic SIEMENS

The basics of UVVM
- For advanced verification

BFM to VVC: Why and how?

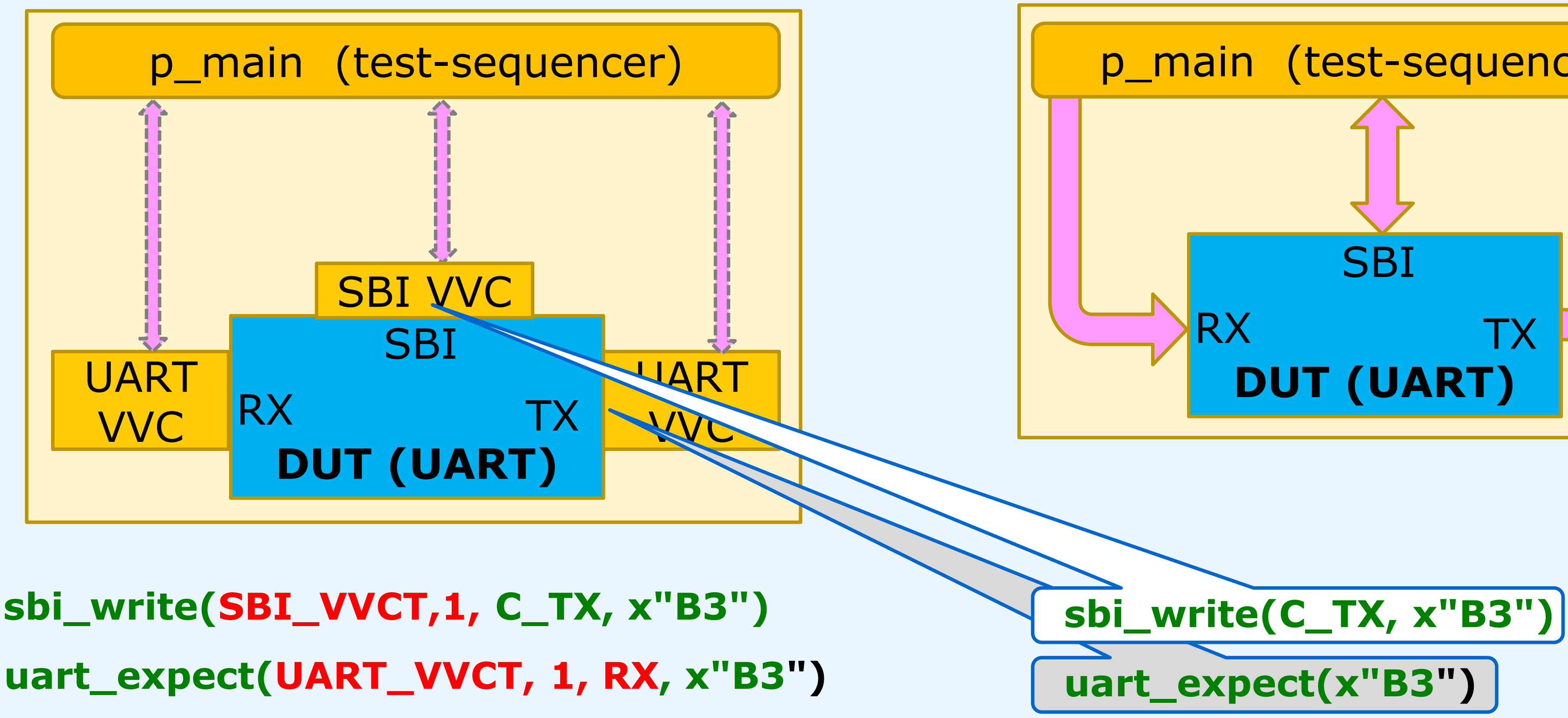
- BFMs are great for simple testbenches
  - Dedicated procedures in a simple package
  - Just reference and call from a process
- BUT
  - A process can only do one thing at a time
    - Either execute that BFM
    - **<u>Or</u>** execute another BFM
    - **<u>Or</u>** do something else
- To do more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



sbi_write(C_TX, x"B3")

uart_expect(x"B3")

BFM to VVC: Why and how?



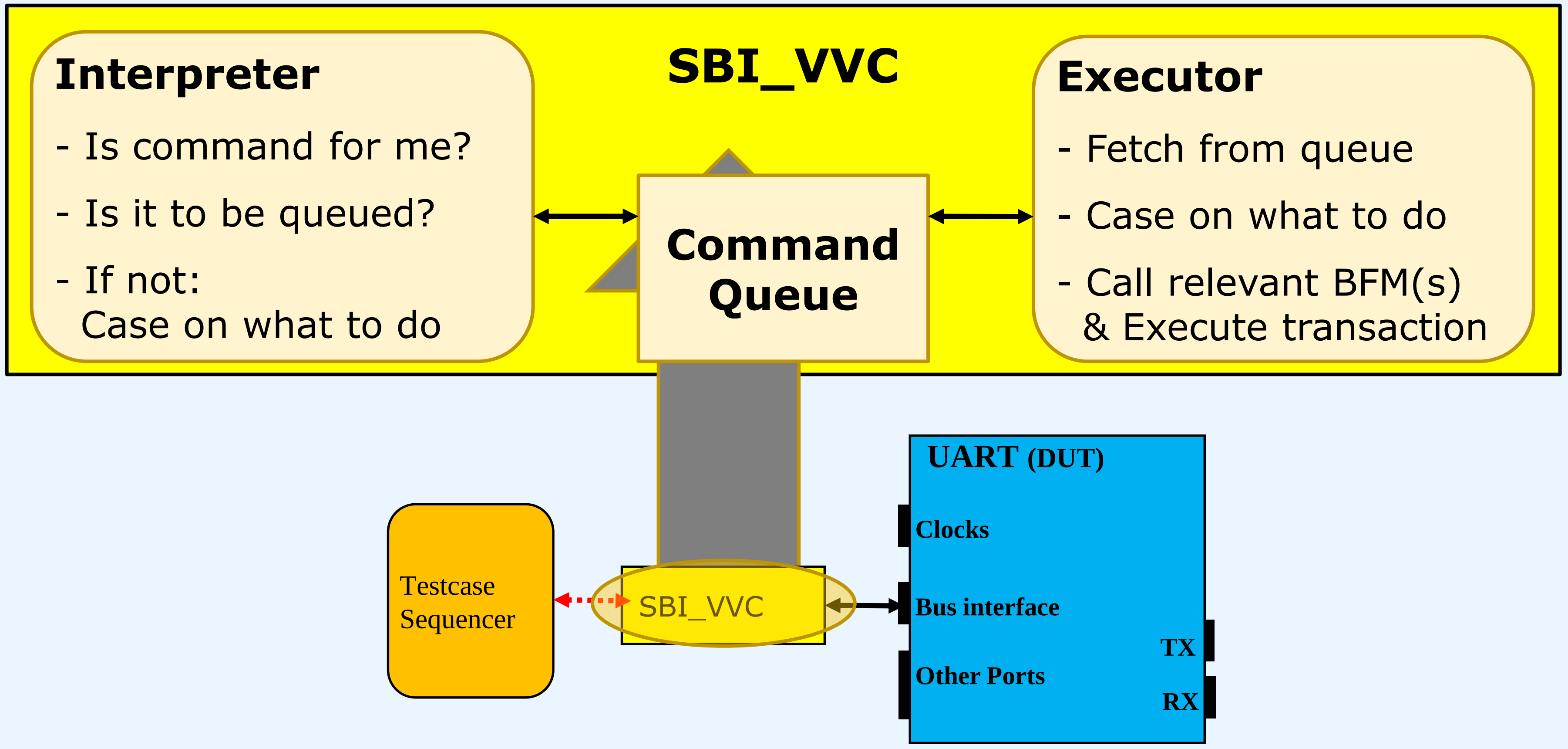
sbi_write(C_TX, x"B3")

uart_expect(x"B3")

sbi_write(SBI_VVCT,1, C_TX, x"B3")

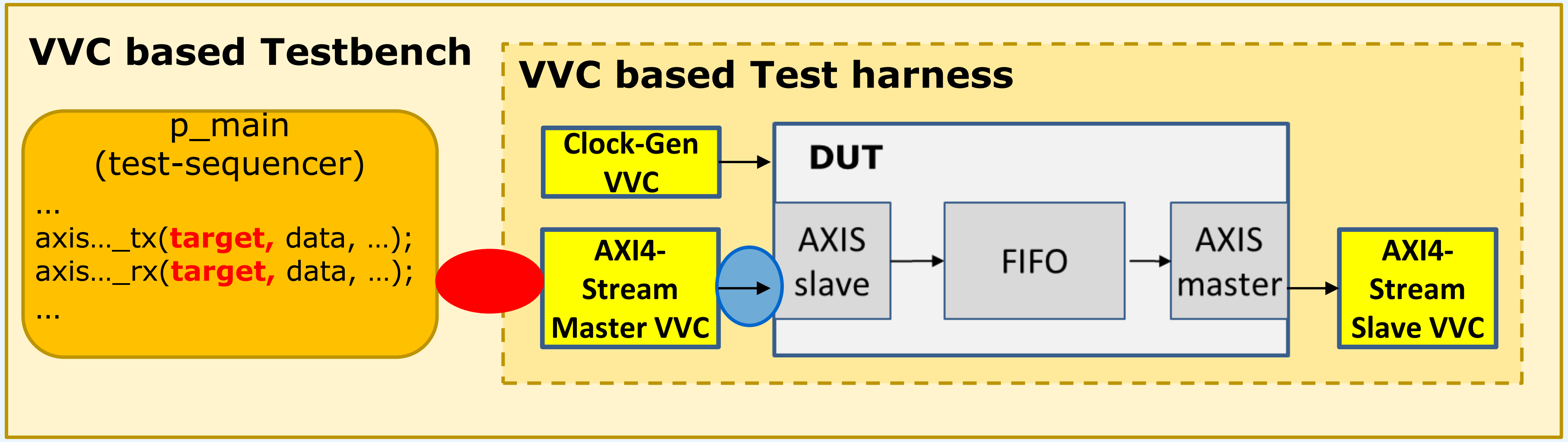
uart_expect(UART_VVCT, 1, RX, x"B3")

VVC: VHDL Verification Component

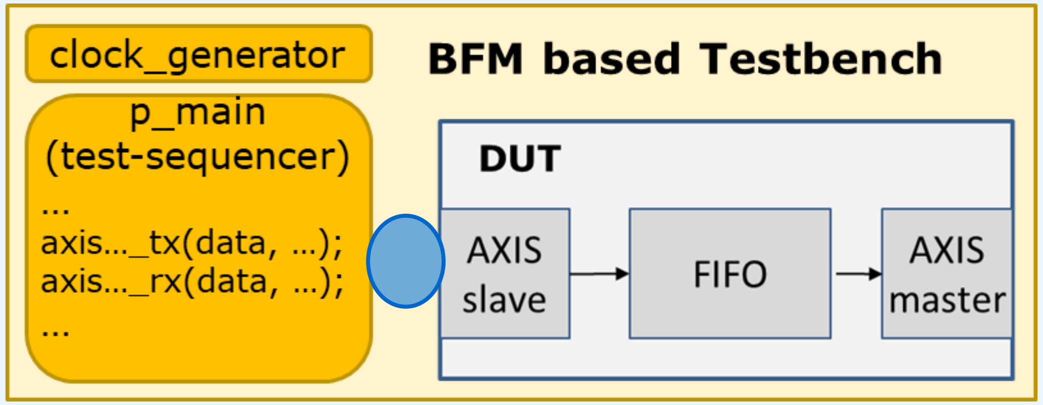


AXI-stream VVC based TB

See webinar 3



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



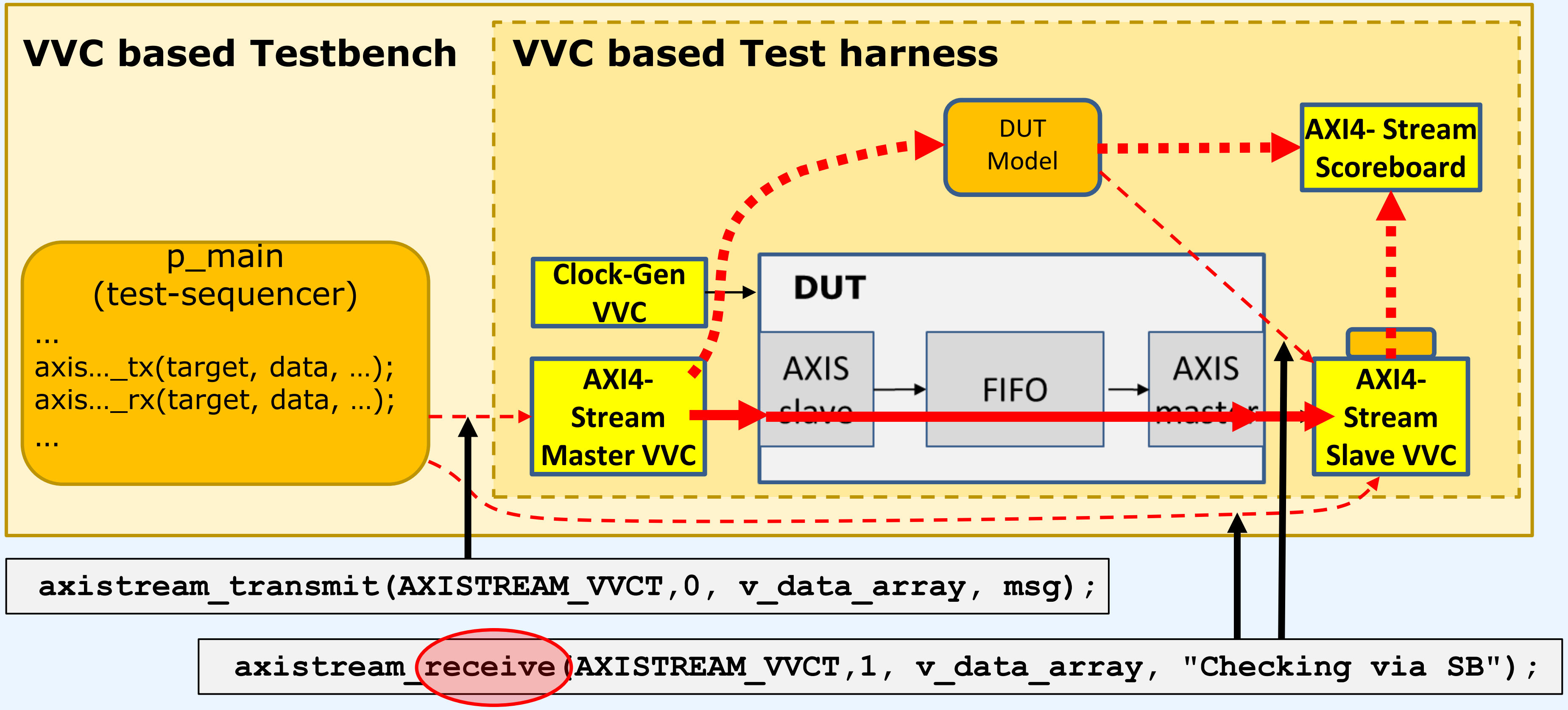
Only in **UVVM** VVCs:

- May insert delay between commands – from sequencer
  → The only system to target cycle related corner cases
- Simple handling of split transactions and out of order protocols
- Common commands to control VVC behaviour
- May use Broadcast and Multicast
- Really unique: Control all VVCs from a single sequencer!

```
axistream_transmit(v_byte_array, msg, clk, m_axis);
```

EmLogic SIEMENS

Advanced scoreboard-based TB



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

```
axistream_receive(AXISTREAM_VVCT,1, v_data_array, "Checking via SB");
```

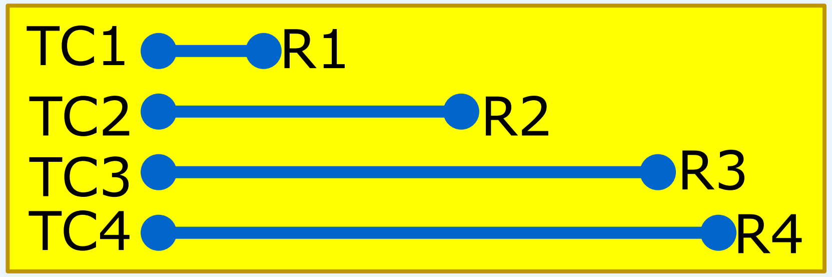
Specification Coverage

See webinar 2

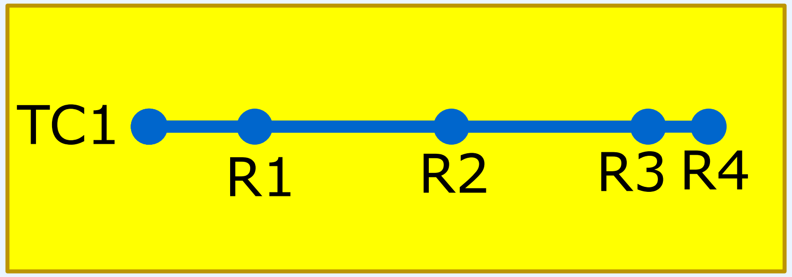
- Assure that all requirements have been verified
  1. Specify all requirements
  2. Report coverage from test sequencer (or other TB parts)
  3. Generate summary report

Requirement Label	**Description**
MOTOR_R1	The acceleration shall be ***
MOTOR_R2	The speed shall be given by ***
MOTOR_R3	The deceleration shall be ***
MOTOR_R4	The final position shall be ***

- Solutions exist to report that a testcase finished successfully
  - BUT - reporting that a testcase has finished is not sufficient
- What if multiple requirements are covered by the same testcase?
  - E.g. Moving/turning something to a to a given position
    R1: Acceleration R2: Speed R3: Deceleration 4: Position etc..

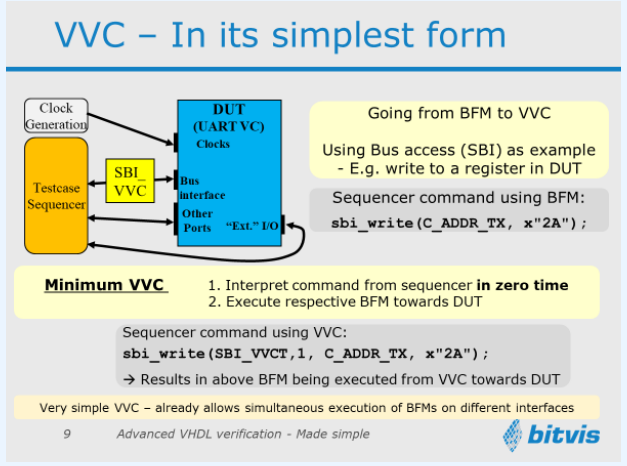


VS



More details on – Advanced TBs

See webinar 3



VVC – In its simplest form

Going from BFM to VVC

Using Bus access (SBI) as example
- E.g. write to a register in DUT

Sequencer command using BFM:

```
sbi_write(C_ADDR_TX, x"2A");
```

Minimum VVC

1. Interpret command from sequencer **in zero time**
2. Execute respective BFM towards DUT

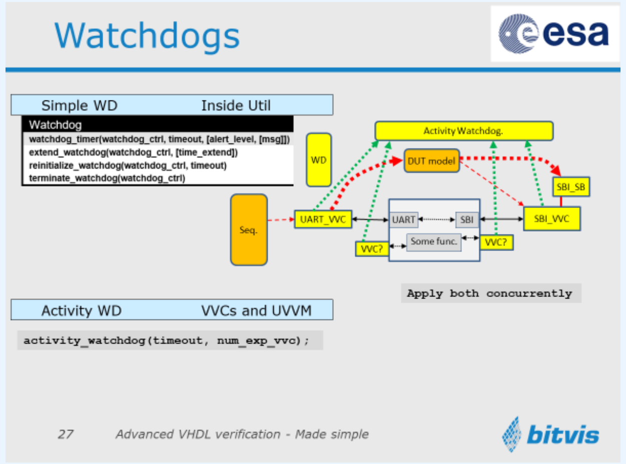
Sequencer command using VVC:

```
sbi_write(SBI_VVCT,1, C_ADDR_TX, x"2A");
```

→ Results in above BFM being executed from VVC towards DUT

Very simple VVC – already allows simultaneous execution of BFMs on different interfaces

9 *Advanced VHDL verification - Made simple* bitvis



Watchdogs

esa

Simple WD	Inside Util

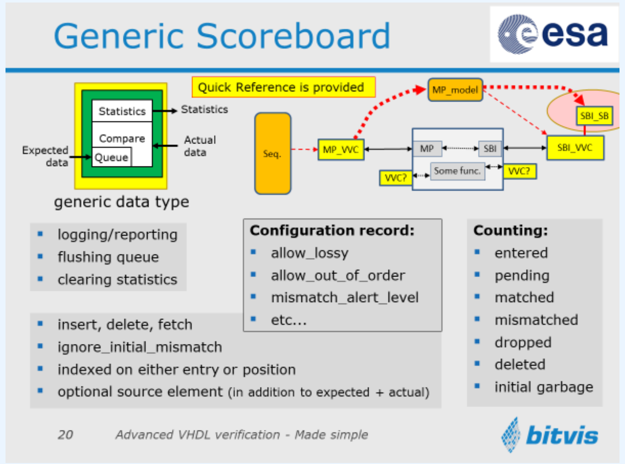
Watchdog
- watchdog_timer(watchdog_ctrl, timeout, [alert_level, [msg]])
- extend_watchdog(watchdog_ctrl, [time_extend])
- reinitialize_watchdog(watchdog_ctrl, timeout)
- terminate_watchdog(watchdog_ctrl)

`Apply both concurrently`

Activity WD	VVCs and UVVM

```
activity_watchdog(timeout, num_exp_vvc);
```

27 *Advanced VHDL verification - Made simple* bitvis



Generic Scoreboard

esa

generic data type

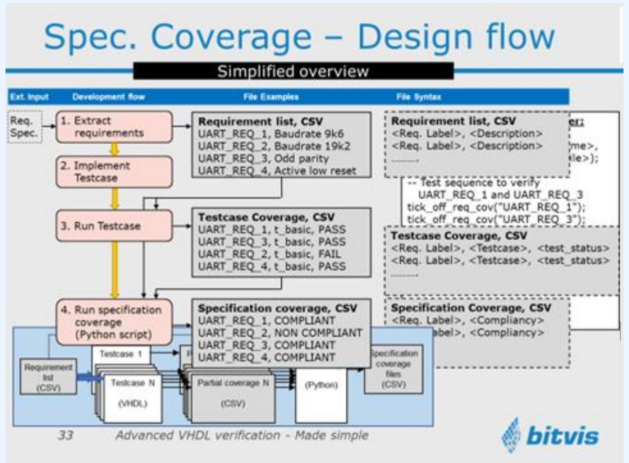
- logging/reporting
- flushing queue
- clearing statistics

- insert, delete, fetch
- ignore_initial_mismatch
- indexed on either entry or position
- optional source element (in addition to expected + actual)

Configuration record:
- allow_lossy
- allow_out_of_order
- mismatch_alert_level
- etc...

Counting:
- entered
- pending
- matched
- mismatched
- dropped
- deleted
- initial garbage

20 *Advanced VHDL verification - Made simple* bitvis



Spec. Coverage – Design flow

Simplified overview

Requirement list, CSV
UART_REQ_1, Baudrate 9k6
UART_REQ_2, Baudrate 19k2
UART_REQ_3, Odd parity
UART_REQ_4, Active low reset

Testcase Coverage, CSV
UART_REQ_1, t_basic, PASS
UART_REQ_3, t_basic, PASS
UART_REQ_2, t_basic, FAIL
UART_REQ_4, t_basic, PASS

Specification coverage, CSV
UART_REQ_1, COMPLIANT
UART_REQ_2, NON COMPLIANT
UART_REQ_3, COMPLIANT
UART_REQ_4, COMPLIANT

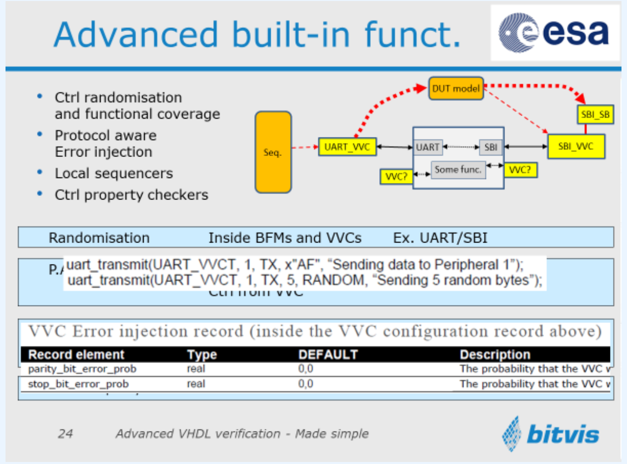
Requirement list, CSV
<Req. Label>, <Description>
<Req. Label>, <Description>
....

```
-- Test sequence to verify
   UART_REQ_1 and UART_REQ_3
tick_off_req_cov("UART_REQ_1");
tick_off_req_cov("UART_REQ_3");
```

Testcase Coverage, CSV
<Req. Label>, <Testcase>, <test_status>
<Req. Label>, <Testcase>, <test_status>
....

Specification Coverage, CSV
<Req. Label>, <Compliancy>
<Req. Label>, <Compliancy>

33 *Advanced VHDL verification - Made simple* bitvis



Advanced built-in funct.

esa

- Ctrl randomisation and functional coverage
- Protocol aware Error injection
- Local sequencers
- Ctrl property checkers

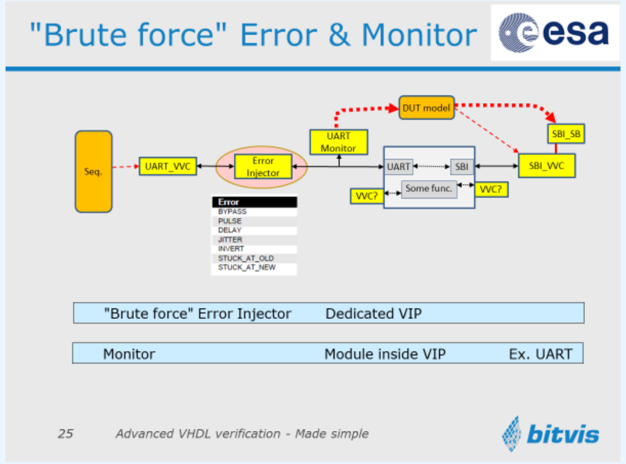
Randomisation	Inside BFMs and VVCs	Ex. UART/SBI

```
uart_transmit(UART_VVCT, 1, TX, x"AF", "Sending data to Peripheral 1");
uart_transmit(UART_VVCT, 1, TX, 5, RANDOM, "Sending 5 random bytes");
```

VVC Error injection record (inside the VVC configuration record above)

Record element	Type	DEFAULT	Description
parity_bit_error_prob	real	0,0	The probability that the VVC v
stop_bit_error_prob	real	0,0	The probability that the VVC v

24 *Advanced VHDL verification - Made simple* bitvis



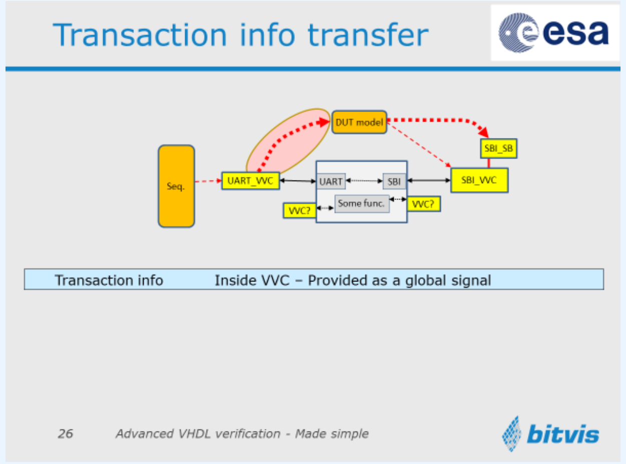
"Brute force" Error & Monitor

esa

Error
BYPASS
PULSE
DELAY
JITTER
INVERT
STUCK_AT_OLD
STUCK_AT_NEW

"Brute force" Error Injector	Dedicated VIP	
Monitor	Module inside VIP	Ex. UART

25 *Advanced VHDL verification - Made simple* bitvis

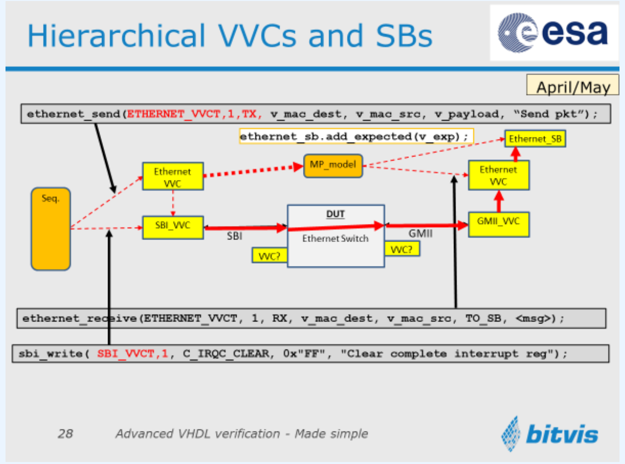


Transaction info transfer

esa

Transaction info	Inside VVC – Provided as a global signal

26 *Advanced VHDL verification - Made simple* bitvis



Hierarchical VVCs and SBs

esa

April/May

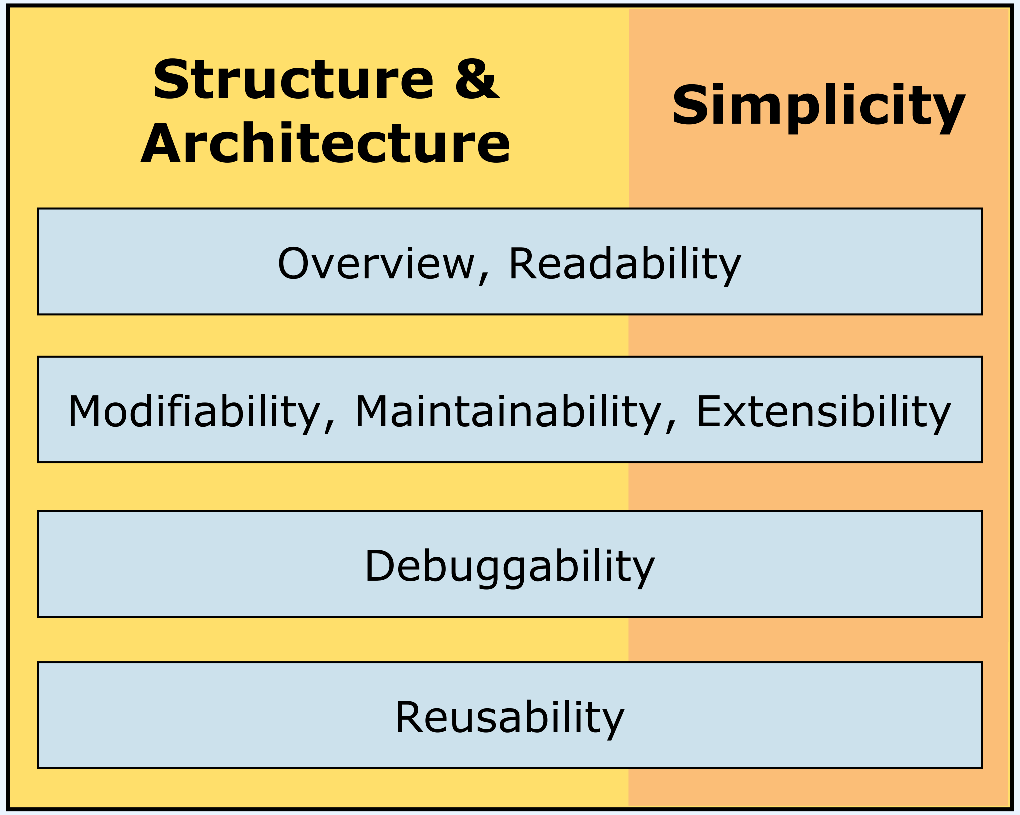
```
ethernet_send(ETHERNET_VVCT,1,TX, v_mac_dest, v_mac_src, v_payload, "Send pkt");
ethernet_sb.add_expected(v_exp);
ethernet_receive(ETHERNET_VVCT, 1, RX, v_mac_dest, v_mac_src, TO_SB, <msg>);
sbi_write( SBI_VVCT,1, C_IRQC_CLEAR, 0x"FF", "Clear complete interrupt reg");
```

28 *Advanced VHDL verification - Made simple* bitvis

EmLogic SIEMENS

Main Benefits of UVVM

Quality and Efficiency enablers



The «mandatory» target for any good Design and Testbench…

The three main development areas for adv. TBs
vs structure and efficiency evaluations

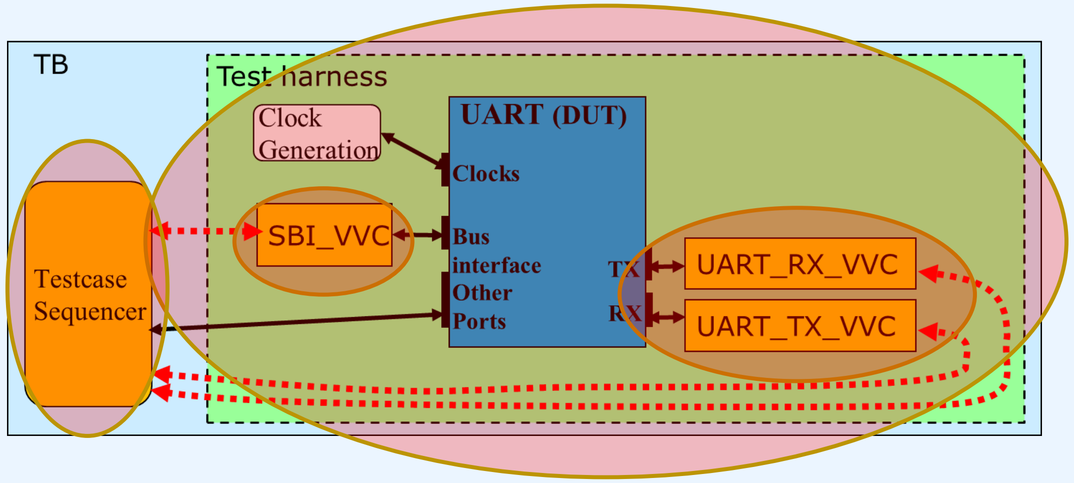
Next: Let's check out the reuse potential

The central sequencer

- Always by far the most time consuming
- Massively simplified (commands + sync)
- Even a SW designer can read **and** write it
- Any number of VVCs easily controlled
- **Huge time saving where it matters the most**

Test harness

- Dead simple
- Anyone can understand it
- Anyone can understand the interaction

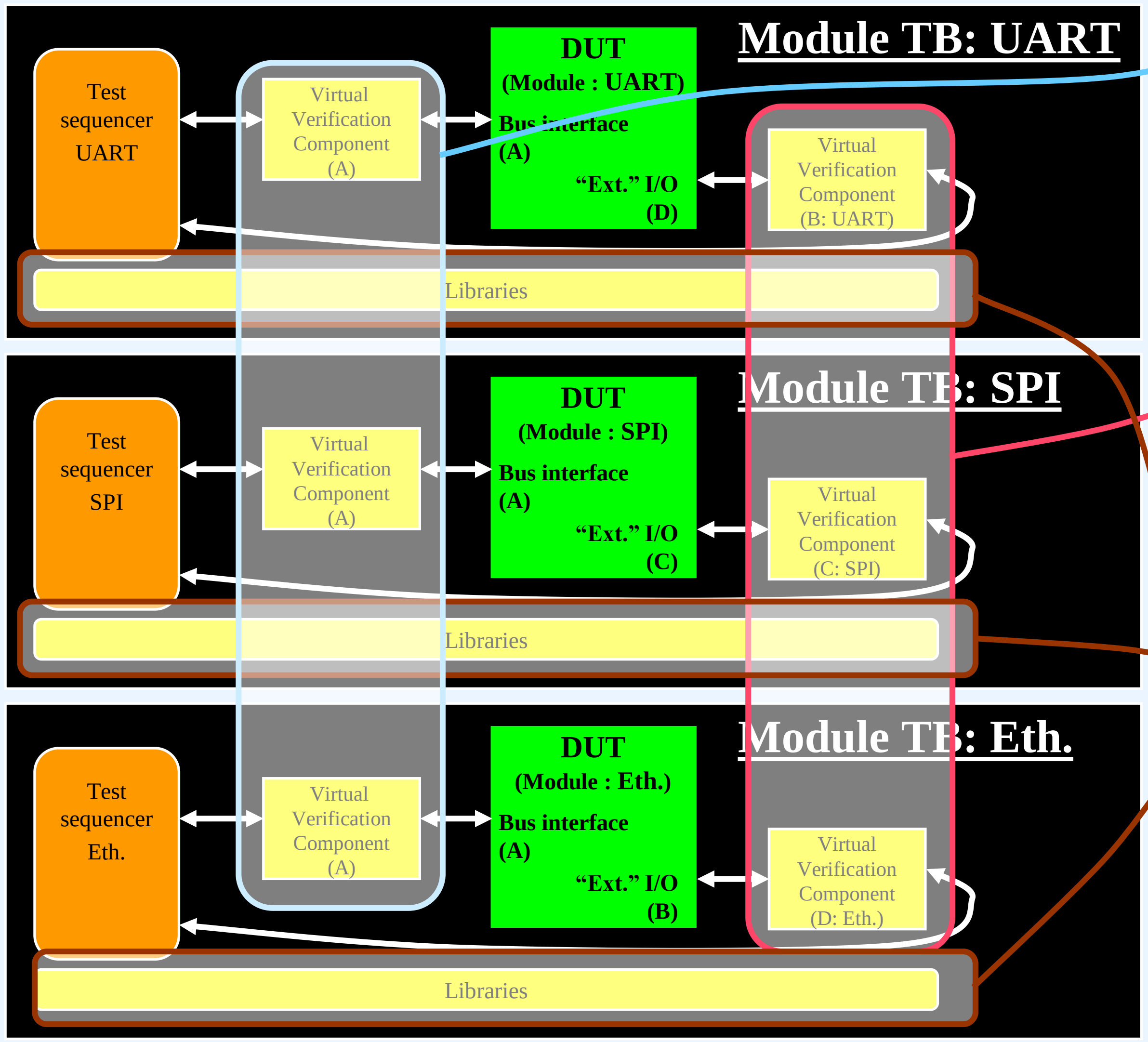


Verification Components

- Autonomous operation
- Encapsulated interface functionality
- Easy to extend with new functionality
- Easy to adapt to more complex protocols
- Plug and play for reuse
- Allows really simple test harness
- Yields a huge improvement for testcase writers

EmLogic SIEMENS

Reuse between Module Testbences



Direct reuse.

Same component.

Structured "copy, paste & modify".

Modification mainly required for actual BFMs and higher level protocol - that really have to be written anyway.

Direct reuse.

Common libraries

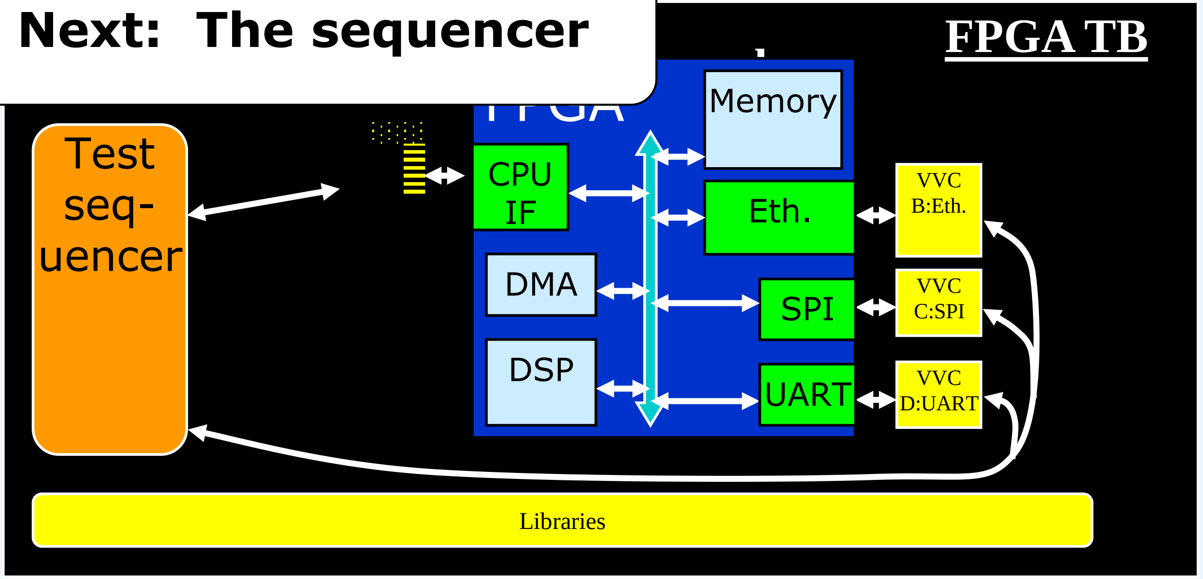
➔ Hence a major reuse between module testbenches.

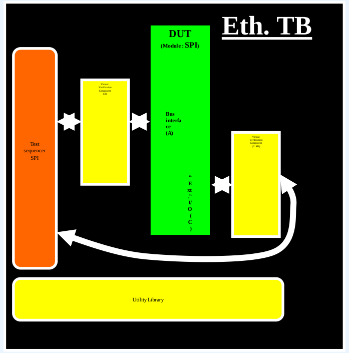
➔ Very efficient & Good overview.

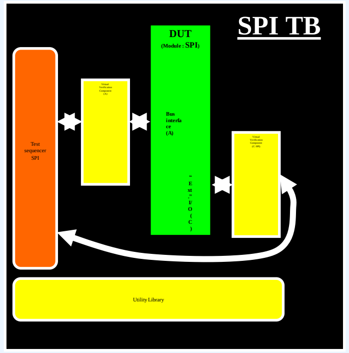
EmLogic SIEMENS

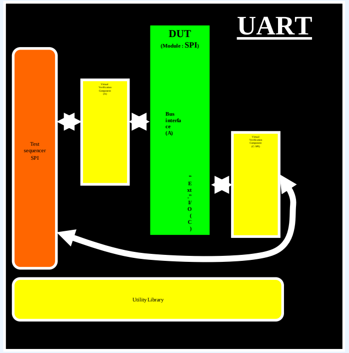
Reuse from module TBs to FPGA TB

Next: The sequencer









Work involved for FPGA TB:

- Consider Test harness.
  - Remove DUT
- - The libraries exist
- - I/O VVCs used as is
- CPU VVC ≈ BUS IF VVC
  - Same Command
  - Same arch./structure
  - Slightly different BFM

- ➔ An extreme reuse from module TBs to FPGA TB
- ➔ Hence FPGA testbench can be made very fast
- ➔ Very efficient & Good overview.

EmLogic SIEMENS

Testbench Sequencer

Simple example:

```
. . .
sbi_write(SBI_VVCT,1 , x"C2", x"58", "Uart TX");

uart_expect(UART_VVCT,1,RX, x"58“);
uart_transmit(UART_VVCT,1,TX, x“A1”);

insert_delay(UART_VVCT,1,TX, 2*C_BIT_PERIOD);
uart_transmit(UART_VVCT,1,TX, x“B2”);
await_completion(UART_VVCT,1,TX);

sbi_check(SBI_VVCT,1, x"C3", x“A1", "Uart RX");
sbi_check(SBI_VVCT,1, x"C3", x“B2", "Uart RX");
.........
report_simulation_summary;
```

➔ Anyone can understand it

➔ Anyone can write it

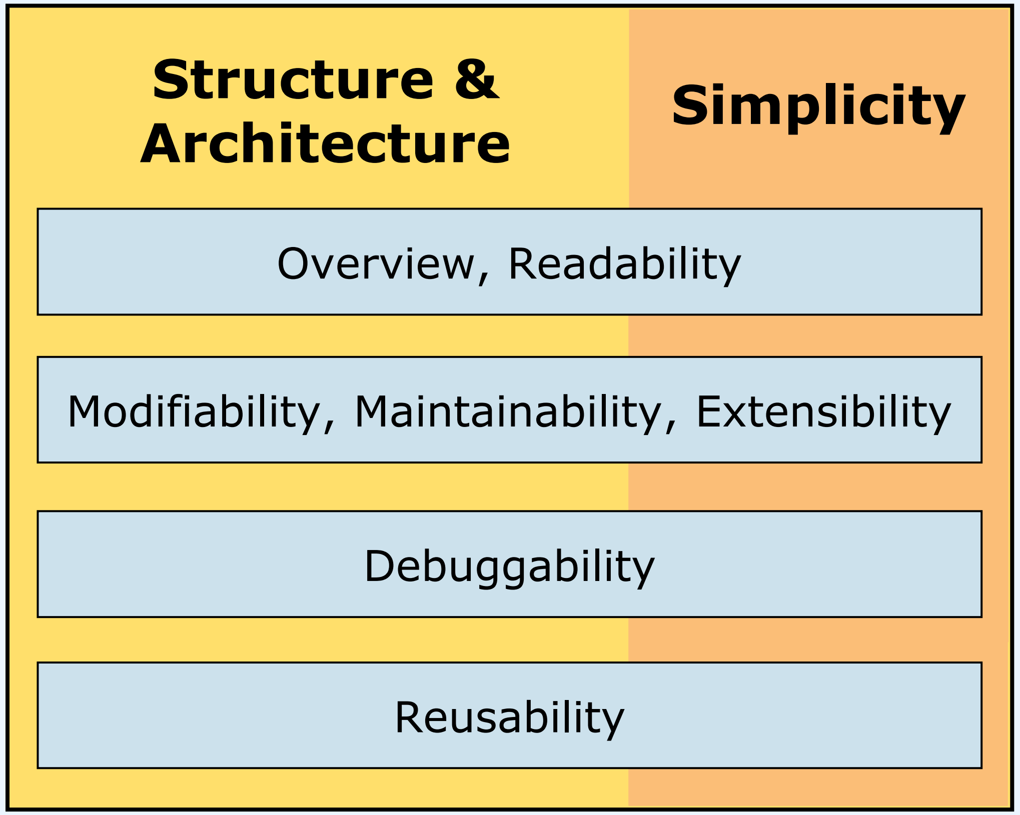
A huge majority of verification time is spent on:

- Writing test cases
- Debugging test cases
- Adapting a chaotic testbench
- Debugging inter process comm.

➔ **Significant speed-up**

EmLogic SIEMENS

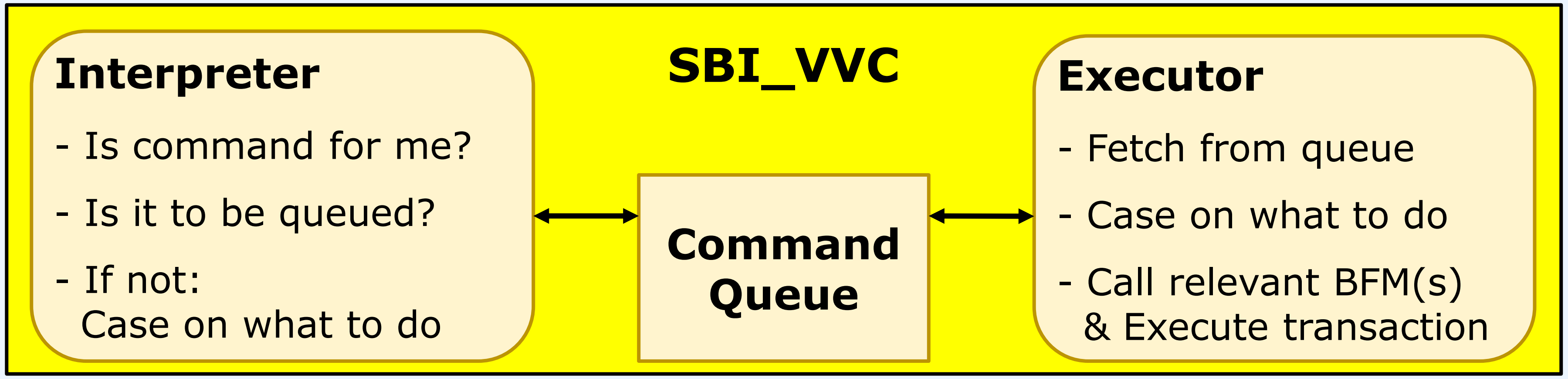
Quality and Efficiency enablers - revisited



UVVM provides this.
UVVM promotes and encourages this.
UVVM facilitates this for your TB
UVVM drives your TB towards this

Why UVVM is better

VVC: VHDL Verification Component



Same main architecture in every VVC

- >95% same code - apart from BFM calls

→ Standard VVC internal architecture

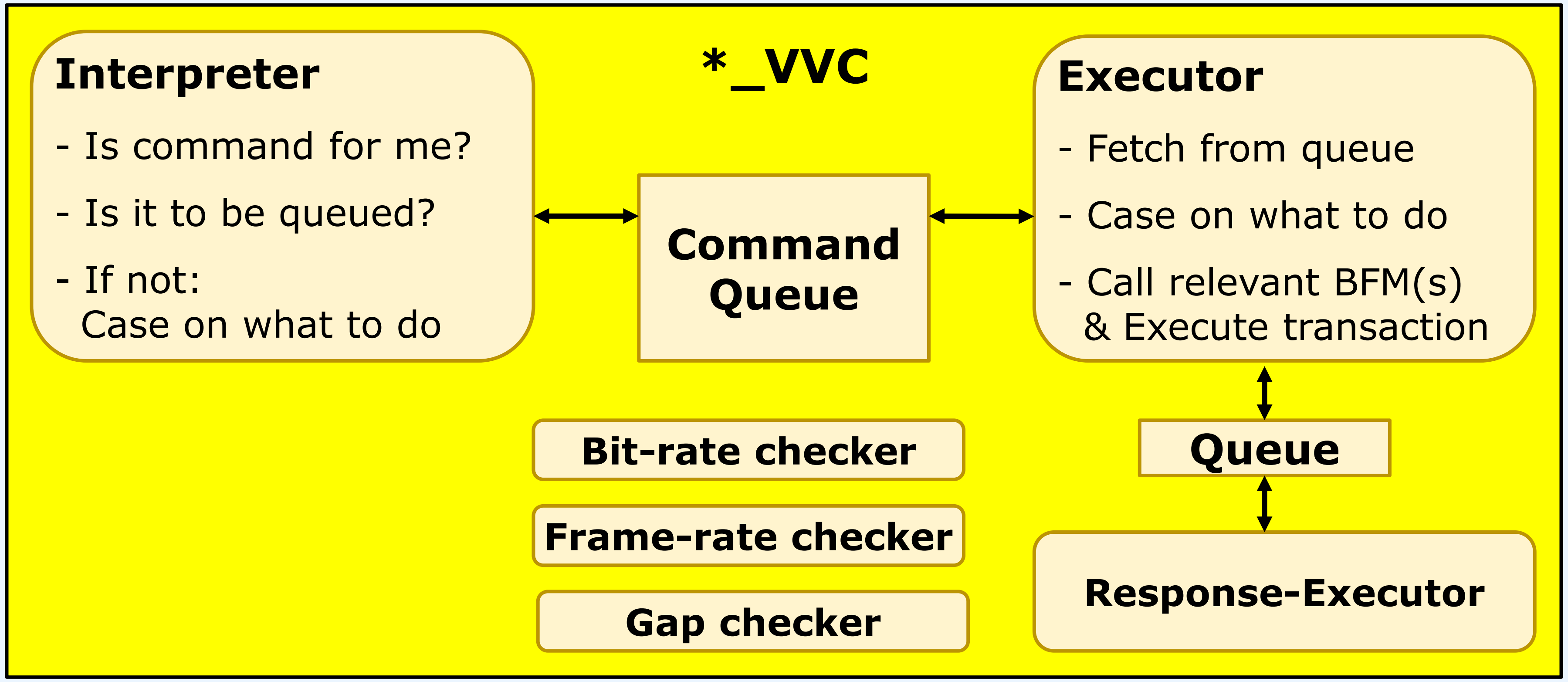
→ Standard VVC external interface

<u>VVC Generation</u>

UART BFM to UART_VVC:
less than 30 min
(using vvc_generator.py)

VVC: Easy to extend

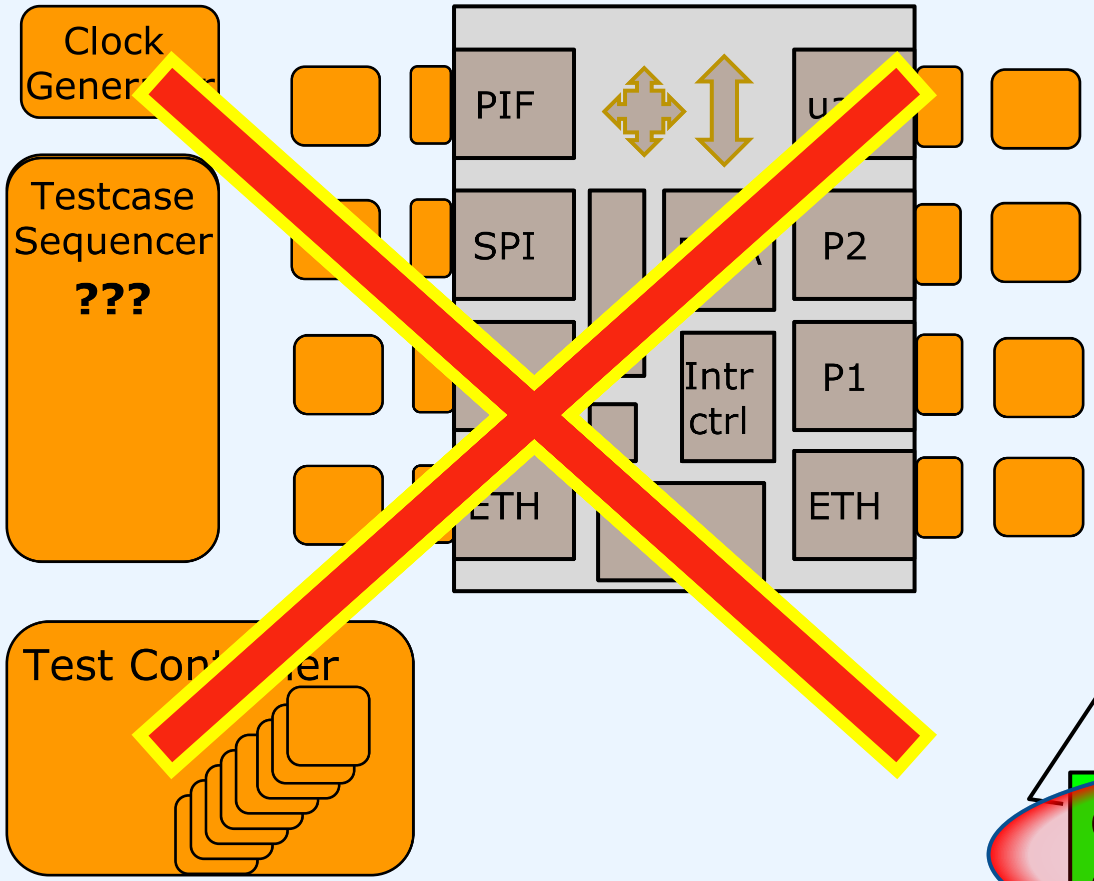
- **Easy to add local sequencers**
- **Easy to add checkers/monitors/etc**
- **Easy to handle split transactions**
- **Easy to handle out of order execution**

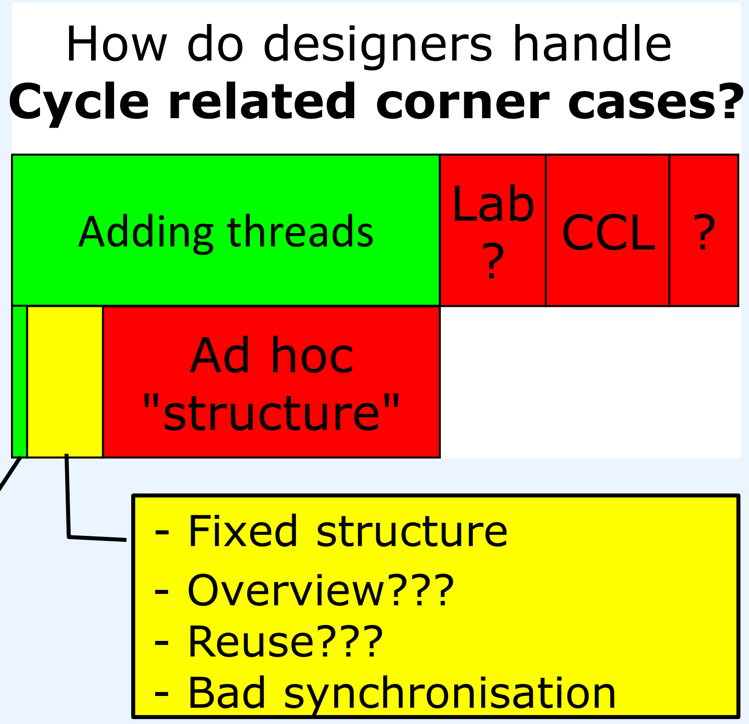


<u>Standardised</u>:
- Queuing system
- Handling of multithreaded interfaces
- Control of parallel checkers

EmLogic SIEMENS

Cycle related corner cases & Multiple interfaces





How do designers handle
Cycle related corner cases?

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

Good architecture, overview, reuse & sync.

Equally important for handling multiple interfaces in general...

EmLogic SIEMENS

More features unique to UVVM

- May simultaneously control all VVCs from a single sequencer – if you like
- Simple synchronization of interface actions – from that single sequencer
- May insert delay between commands – from sequencer
- Simple handling of split transactions and out of order protocols
- Common commands to control general VVC behaviour
- May use Broadcast and Multicast for common commands

The only system to also target cycle related corner cases

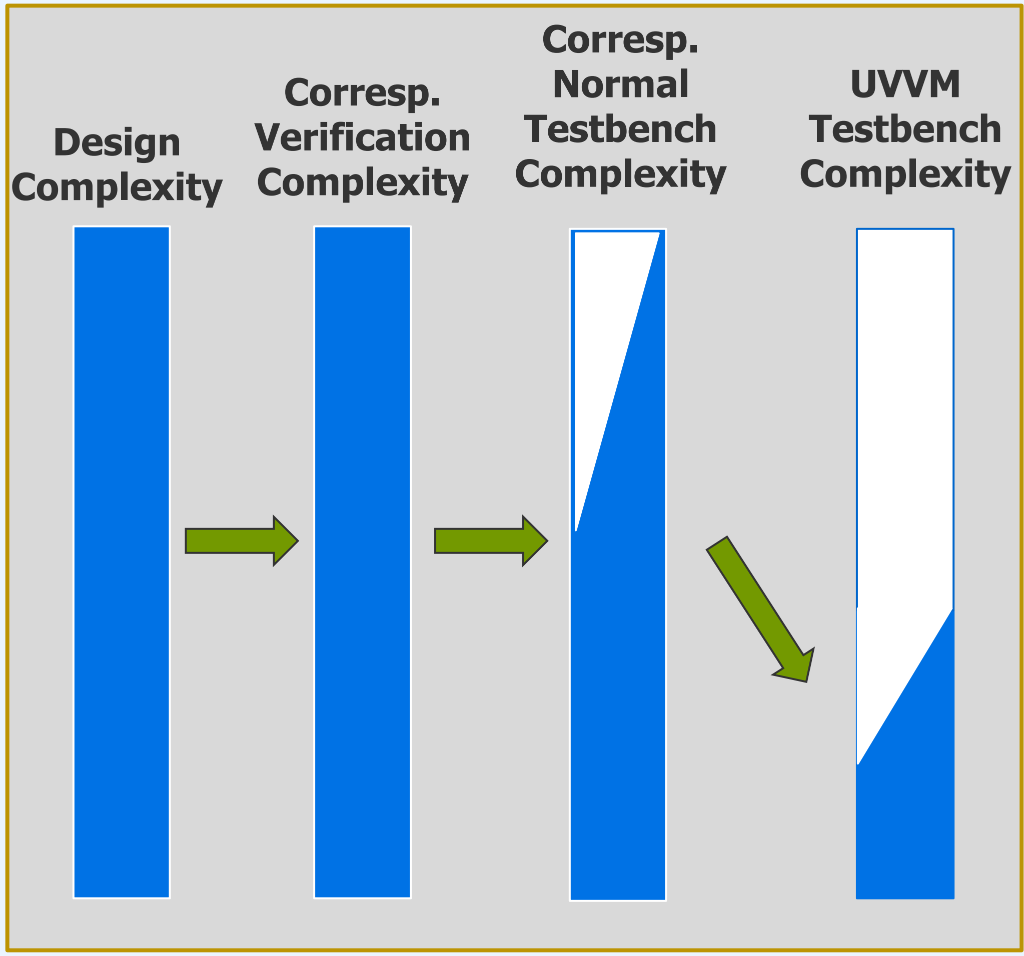
The only system to align all interface stimuli and checks from a single test sequencer process
(but still allows multiple test sequencers when that is needed)

EmLogic SIEMENS

Wishful thinking? - And the result of that

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


Design Complexity
Corresp. Verification Complexity
Corresp. Normal Testbench Complexity
UVVM Testbench Complexity

EmLogic SIEMENS

The largest collection of interface models

UVVM has by far the largest collection of open source VIP available

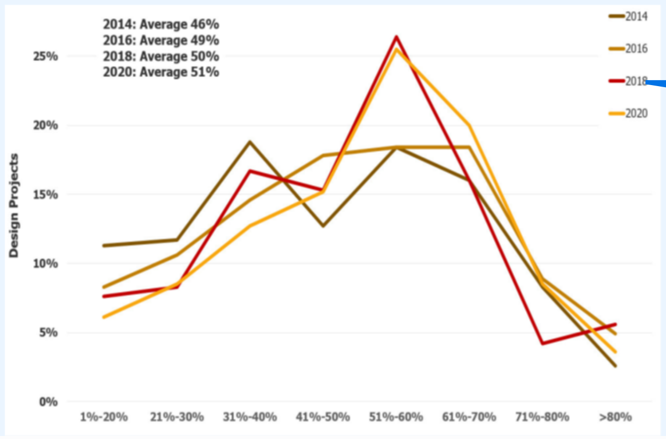
All available as both BFMs and VVCs – your choice

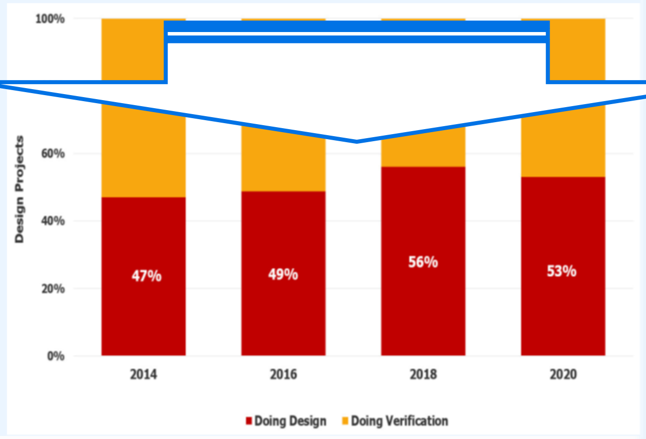
<u>Free, Open source BFMs and VVC (*:VVC-only):</u>

- **AXI4-lite**
- **AXI4-stream**
- **Full AXI4**
- **Avalon MM**
- **Avalon Stream**
- **SPI**
- **I2C**
- **UART**
- **GPIO**
- **SBI**
- **GMII**
- **RGMII**
- **Ethernet (*)**
- **Clock Generator**
- **Error Injector (*)**

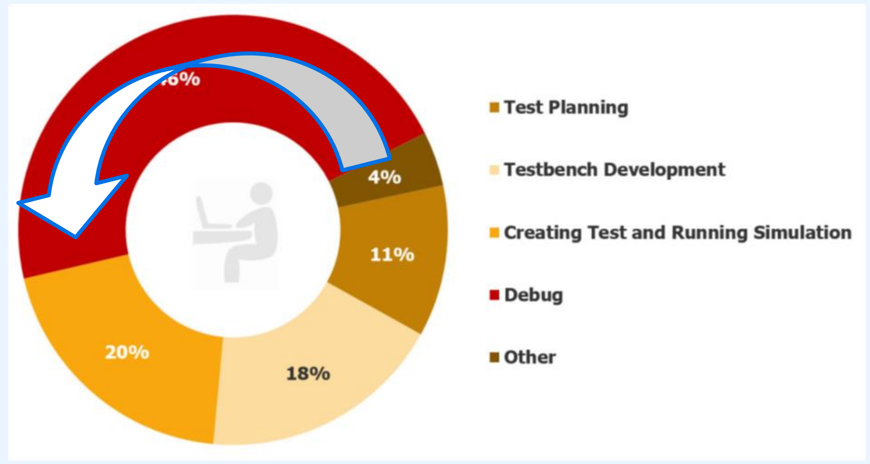
Summary

Half the project time is spent in verification





Half the verification time is spent on debugging



2020 WILSON RESEARCH GROUP, FUNCTIONAL VERIFICATION STUDY, FPGA FUNCTIONAL VERIFICATION TREND REPORT

Structure & Architecture | **Simplicity**

- Overview, Readability
- Modifiability, Maintainability, Extensibility
- Debuggability
- Reusability

Significantly affects:

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

Easily save 100-500 hours
Sometimes 1000-3000 hours

Reduce late project iterations

Faster SW development

More happy customers

EmLogic SIEMENS

UVVM
The main benefits of the world’s #1 VHDL Verification Methodology

By Espen Tallaksen, Director FPGA and Space, EmLogic

EmLogic
The Norwegian Embedded Systems & FPGA Design Centre

SIEMENS