bitvis

110001011010011110100111011011010011110011

UVVM
Setting a standard for VHDL testbenches

ESA - SEFUW 2018

www.bitvis.no

Your partner for SW and FPGA

Added for handout version

Handout version

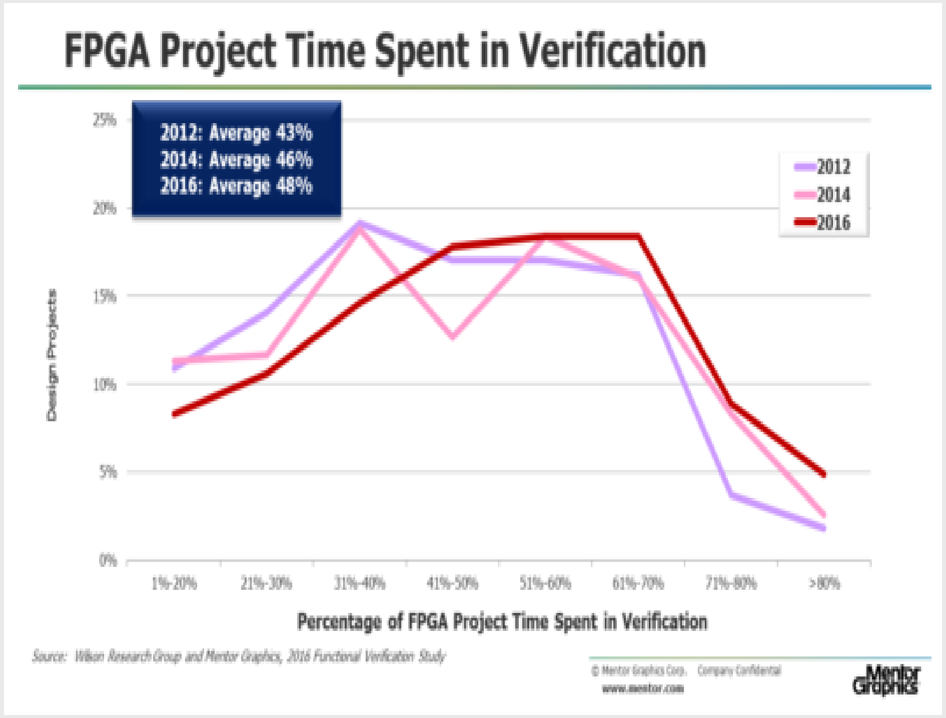
- Some slides were skipped during the presentation in order to keep to the schedule.
  These are now included (and marked as such)
- The presentation had a lot of animation to ease the understanding. This is not available in this PDF.
  If you would like to have a copy of the animated presentation (as a powerpoint-show-file), please send a request to espen.tallaksen@bitvis.no , and I will send it to you.
- You may download the complete UVVM from www.github.com/UVVM
- Note that the UVVM project on Github is being reorganised in week 15 in order to allow community contributions.
  Improved info will be published during this period.

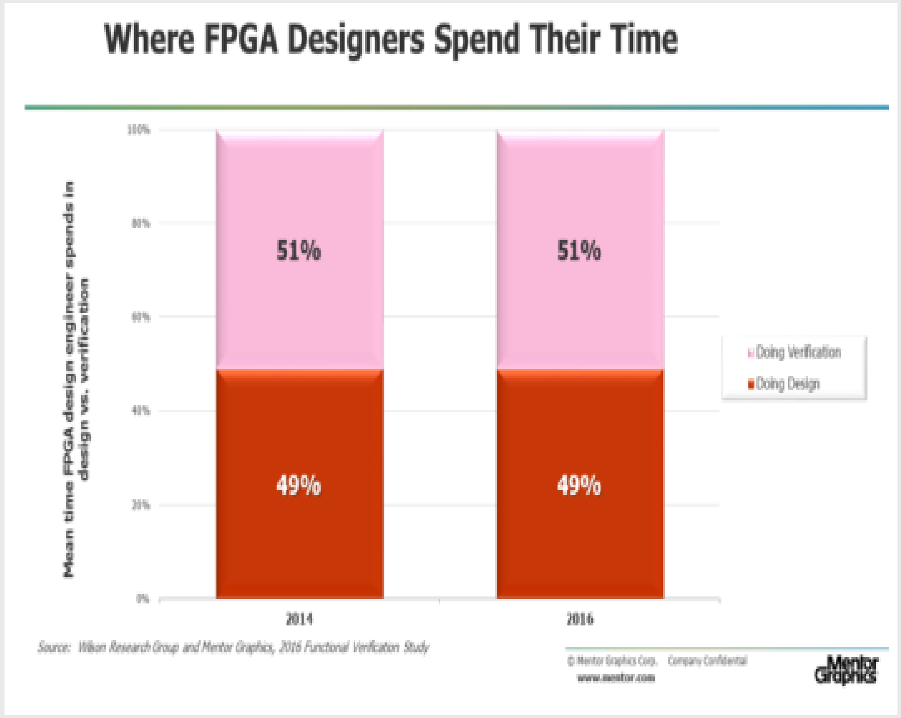
Added for handout version

VHDL

- A huge majority of European FPGA designers are using VHDL for both design and verification
  - Some published surveys have strange numbers for VHDL usage. These numbers are not even close to correct, and the survey summaries often fail to mention the strong position that VHDL has in Europe in particular.
    - The incorrectness is most probably caused by the fact that the survey is sent to an email list that is dominated by non-VHDL users - for various reasons.
  - Questions to participants at FPGA Kongress in Munich (probably the largest FPGA conference in Europe) indicated that maybe as many as 9 out of 10 was using VHDL...

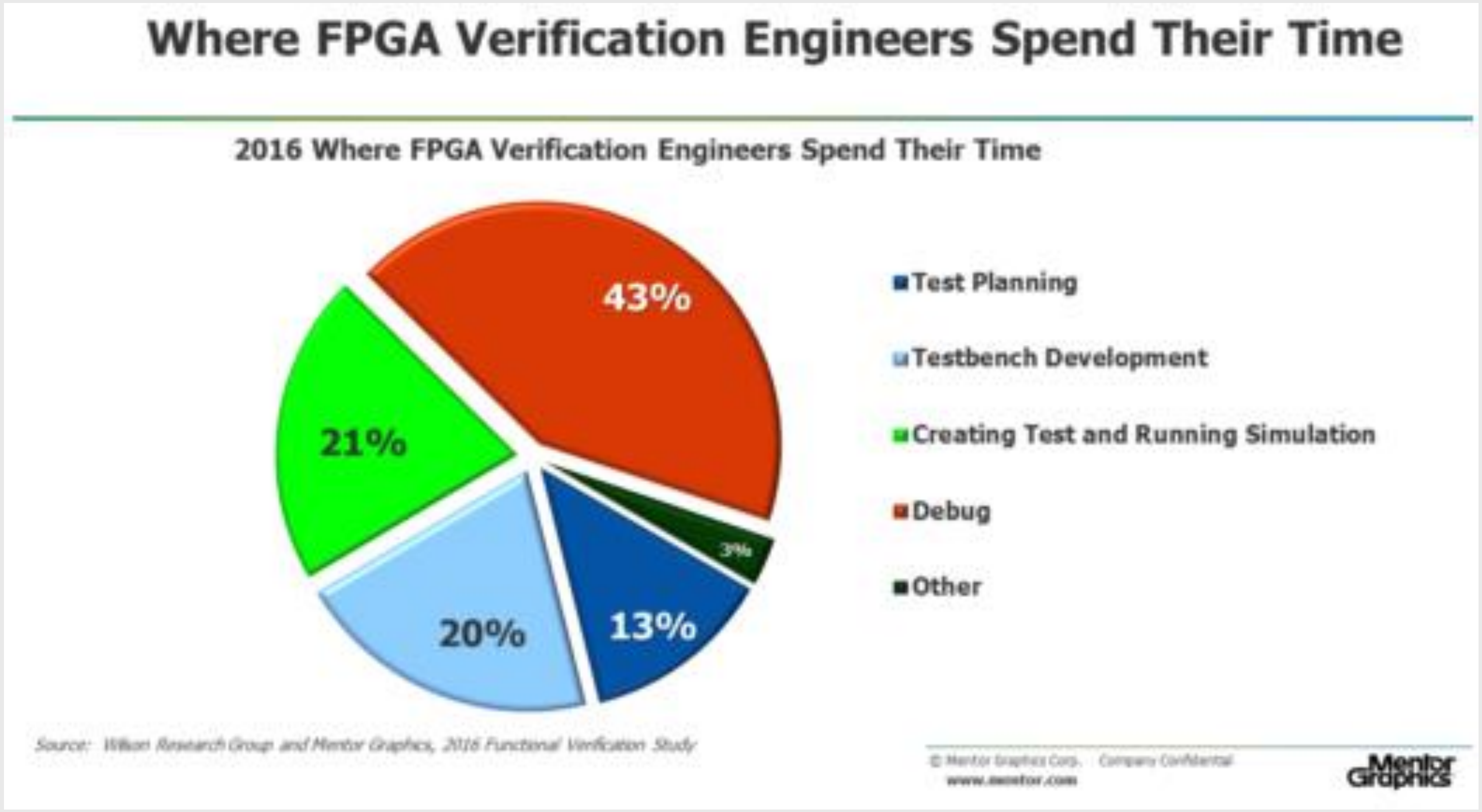
The 2016 Wilson Research Group Functional Verification Study (1)





bitvis

The 2016 Wilson Research Group Functional Verification Study (2)



Main Testbench scope

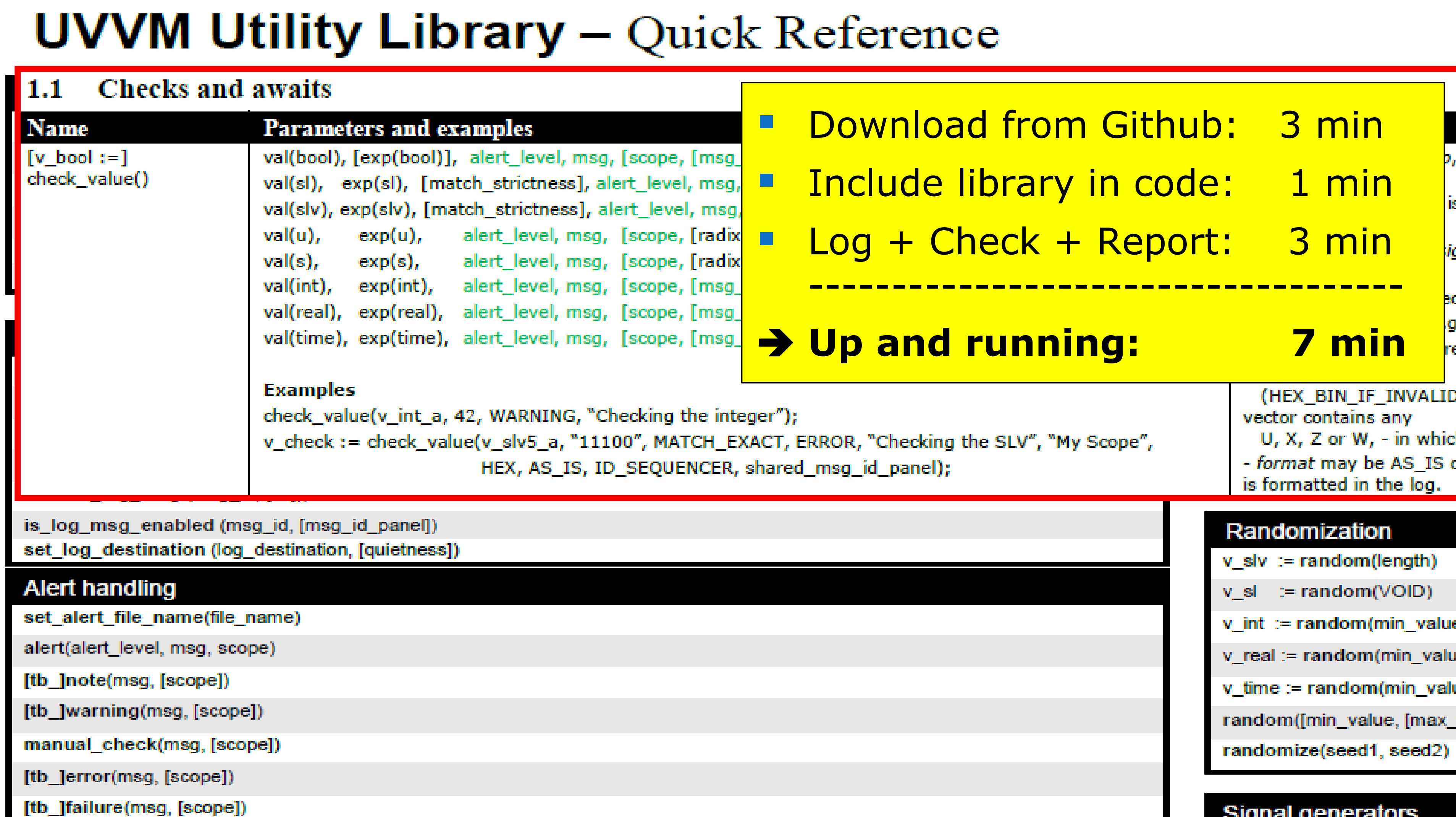
- Purpose:
  To verify DUT requirements

- Challenge:
  Sufficient functional coverage with a minimum effort

- **Testbench Requirements to meet challenge:**
  - ➔ Simple to write
  - ➔ **Simple to understand and modify - by anyone**
  - ➔ **Simple to execute, debug and understand reports & results**

Applies to **any** Testbench of **any** complexity

Entry level : UVVM Utility Library



UVVM Utility Library – Quick Reference

1.1 Checks and awaits

Name	Parameters and examples
[v_bool :=] check_value()	val(bool), [exp(bool)], alert_level, msg, [scope, [msg_
	val(sl), exp(sl), [match_strictness], alert_level, msg,
	val(slv), exp(slv), [match_strictness], alert_level, msg,
	val(u), exp(u), alert_level, msg, [scope, [radix
	val(s), exp(s), alert_level, msg, [scope, [radix
	val(int), exp(int), alert_level, msg, [scope, [msg_
	val(real), exp(real), alert_level, msg, [scope, [msg_
	val(time), exp(time), alert_level, msg, [scope, [msg_

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);

(HEX_BIN_IF_INVALID
vector contains any
U, X, Z or W, - in whic
- *format* may be AS_IS o
is formatted in the log.

is_log_msg_enabled (msg_id, [msg_id_panel])
set_log_destination (log_destination, [quietness])

Alert handling

set_alert_file_name(file_name)
alert(alert_level, msg, scope)
[tb_]**note**(msg, [scope])
[tb_]**warning**(msg, [scope])
manual_check(msg, [scope])
[tb_]**error**(msg, [scope])
[tb_]**failure**(msg, [scope])

Randomization

v_slv := **random**(length)
v_sl := **random**(VOID)
v_int := **random**(min_value
v_real := **random**(min_valu
v_time := **random**(min_valu
random([min_value, [max_
randomize(seed1, seed2)

Signal generators

- Download from Github: 3 min
- Include library in code: 1 min
- Log + Check + Report: 3 min

➔ **Up and running: 7 min**

Simple data path TB

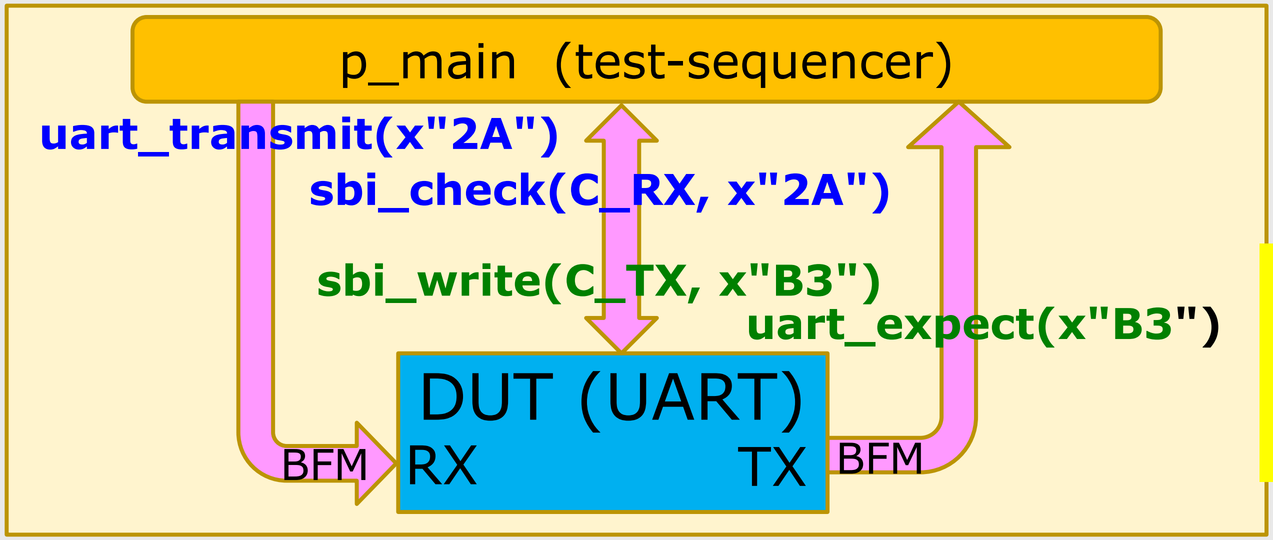
```
check_value(out_val, exp_val, ERROR, "Byte #" & to_string(cnt));
```

```
BV:===============================================
BV: ERROR:
BV:     192 ns. filter_tb
```

May use Utility Library

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

Simple data communication



May use Utility Library and provided BFMs

Free, Open source BFMs:

UART, AXI4-lite, SPI, I2C, Avalon MM, AXI4-stream, GPIO, SBI, …

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

```
BV:   192 ns. uart_tb    sbi_write(x2, ==> xB3) completed. To UART TX
BV: ERROR:
BV:      192 ns. uart_tb
BV:                value was: 'xB2'.  expected 'xB3'.
BV:                (From uart_expect(xB3))
BV:==============================================================
```

bitvis

Quality and Efficiency enablers

Structure & Architecture

Overview, Readability, Simplicity

Modifiability, Maintainability, Extendibility

Debuggability

Reusability

TB approach for complex DUTs

- What is required to verify/test any complex DUT?

  → Provide stimuli on interfaces

  → Check outputs on interfaces

  → Try to reach corner cases (Value **and** Cycle related)

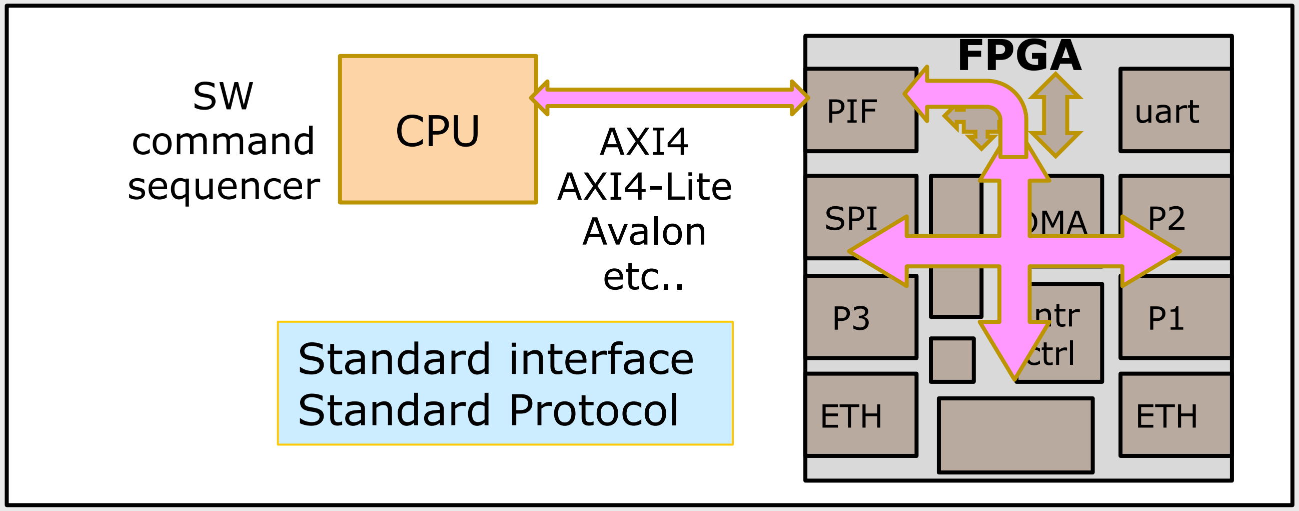
  → Must be able to control multiple interfaces simultaneously

  But how? - for a simple DUT? - for a complex DUT (UART...?)

- Why not learn from SW (controlling HW)?
  - A far more mature methodology!
  - Issues commands at a high abstraction level
  - Distributes tasks to the HW modules - then "forgets" them
    - Controls HW outputs, Reads HW inputs and Configures the HW behaviour
    - Then tasks are handled autonomously inside each HW module
    - HW modules "scream" if something is wrong - or attention is needed
  - Standardised interfaces, protocols, layers, registers, etc...

The SW/HW interface

- Inherently a lot of parallel activity and huge complexity
  - SW/User cannot possibly control all the details inside each module at all times
  - SW/user thus issues pre-defined commands (register setup)
- SW and Design Harness (HW) are totally separated
  - Enables separate and independent work
  - SW is often a magnitude more work than HW
    → Important to allow SW development to be as simple as possible
  - Thus often an abstraction layer in between to allow higher level programming



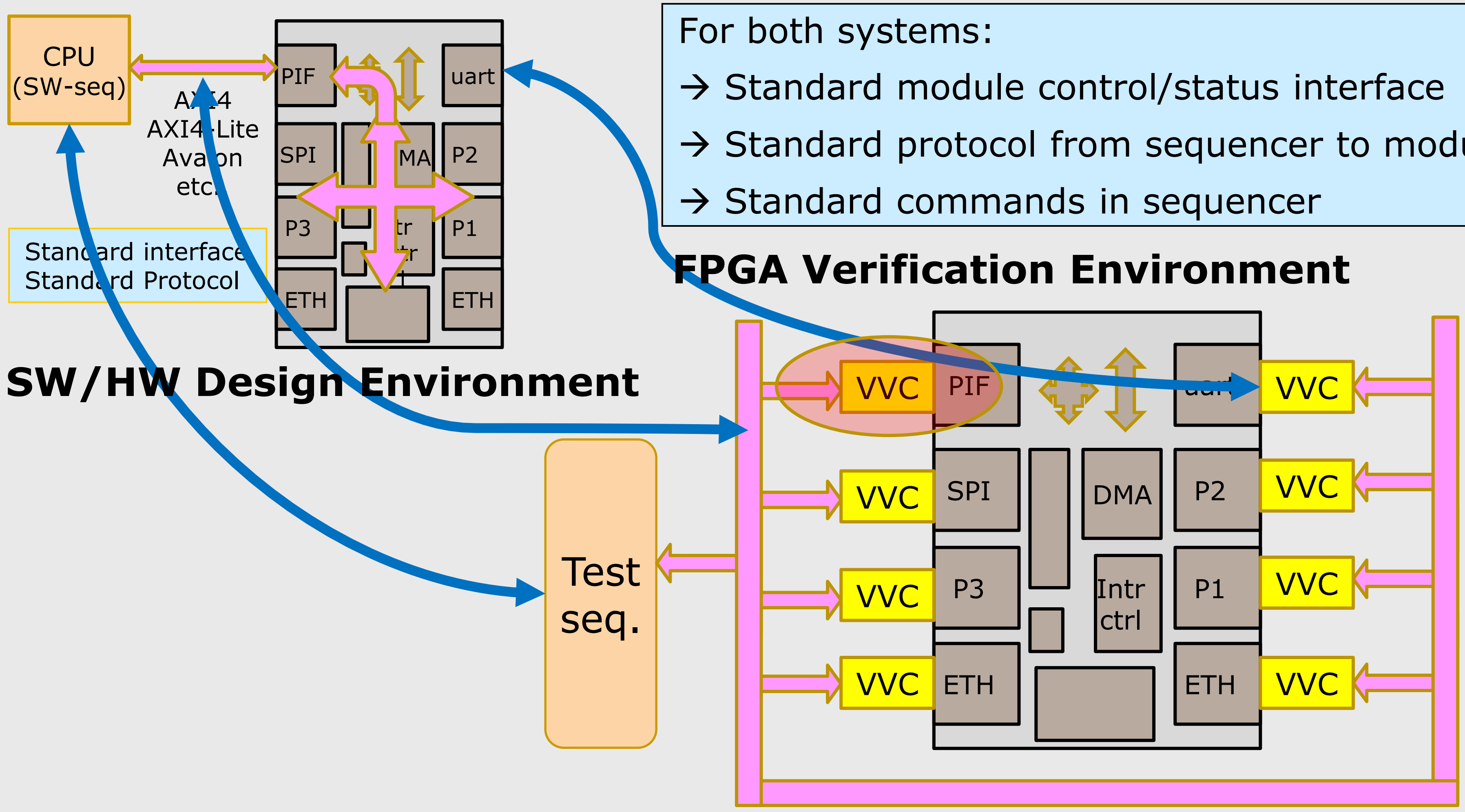
Mirror the SW/HW interface

For both systems:

→ Standard module control/status interface

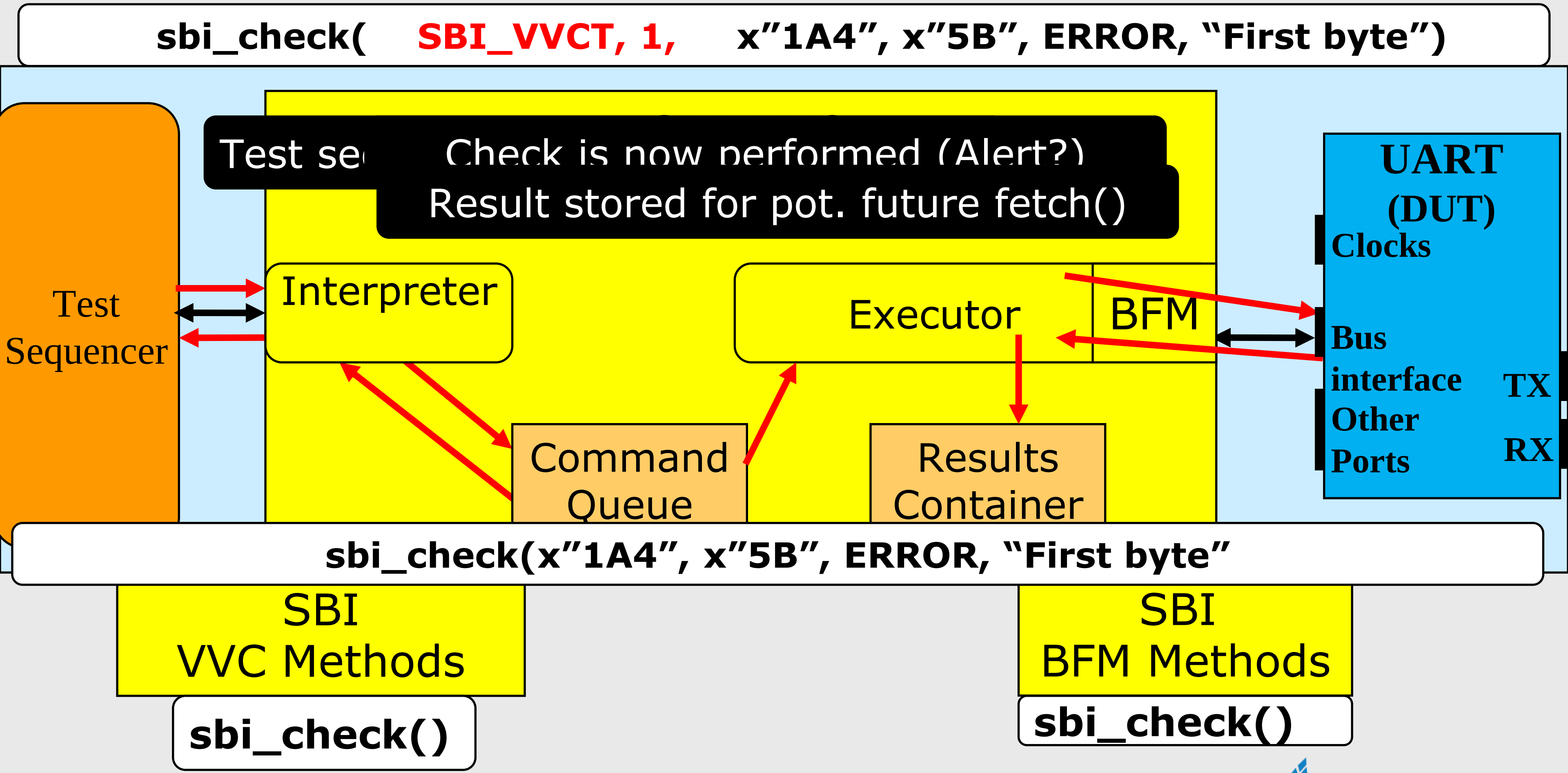
→ Standard protocol from sequencer to modules

→ Standard commands in sequencer



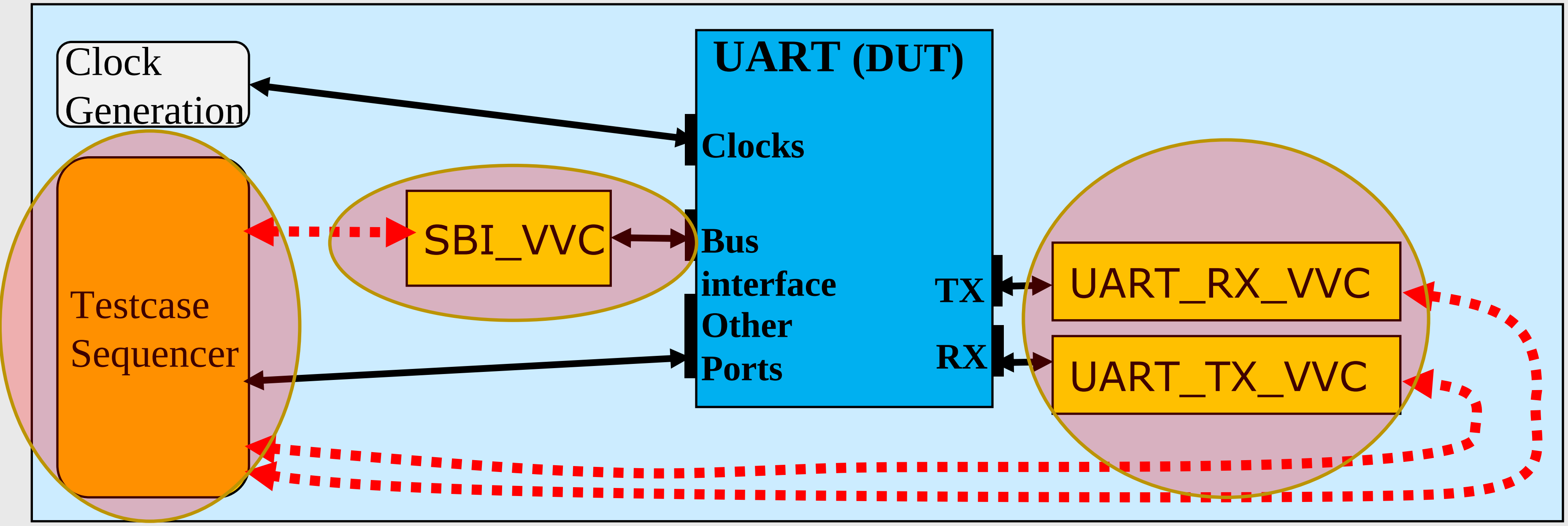
Verification component

Illustration of a simple check-command from sequencer


sbi_check(SBI_VVCT, 1, x"1A4", x"5B", ERROR, "First byte")
Test Sequencer
Test se
Check is now performed (Alert?)
Result stored for pot. future fetch()
Interpreter
Executor
BFM
Command Queue
Results Container
UART (DUT)
Clocks
Bus interface
Other Ports
TX
RX
sbi_check(x"1A4", x"5B", ERROR, "First byte"
SBI VVC Methods
sbi_check()
SBI BFM Methods
sbi_check()

DUT Verification
- Three main development areas

- 1: The complete Testbench with Test Harness
- 2: The Verification Components
- 3: The Central Test Sequencer

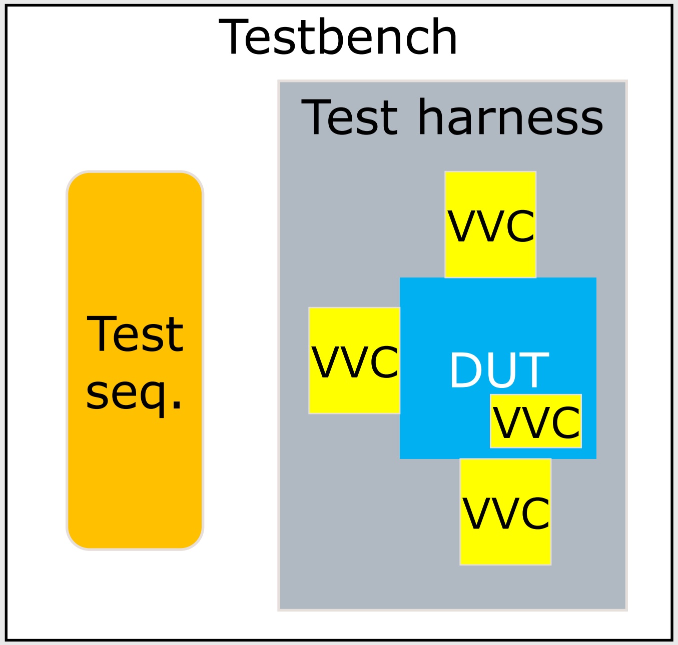


bitvis

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



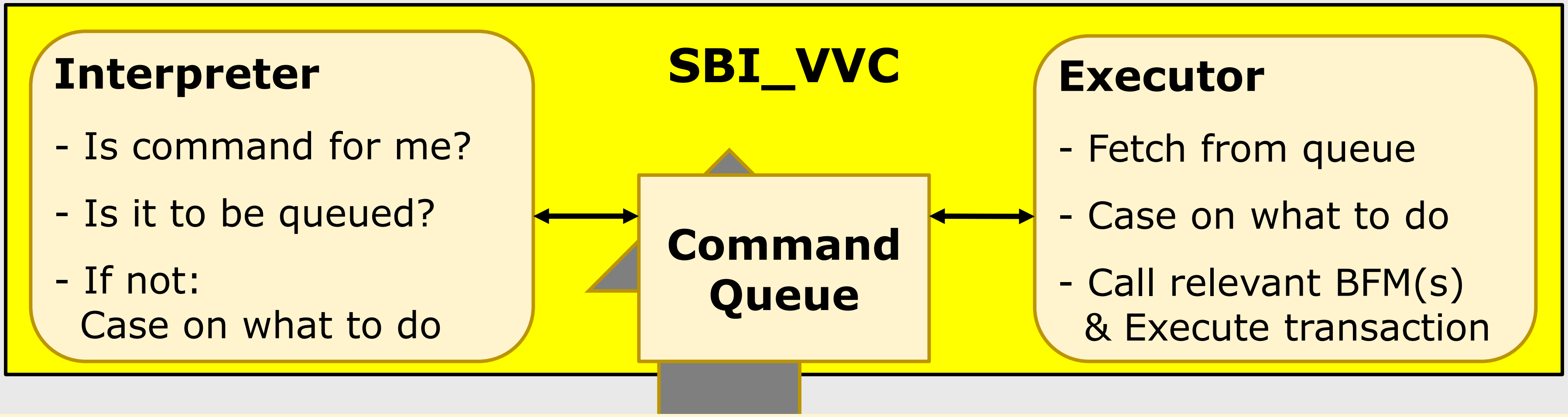
→ Standard global interface throughout test harness

→ Standard protocol from test sequencer to VVC

2: VVC: VHDL Verification Component

(1:Testbench : Easy to implement & understand by anyone)

- Now - what about these VVCs?



Same main architecture in every VVC

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

<u>VVC Generation</u>

UART BFM to UART_VVC:
less than 30 min

→ Standard VVC internal architecture

2: VVC: VHDL Verification Component

→ Standard Queuing system

→ Standard handling of multithreaded interfaces

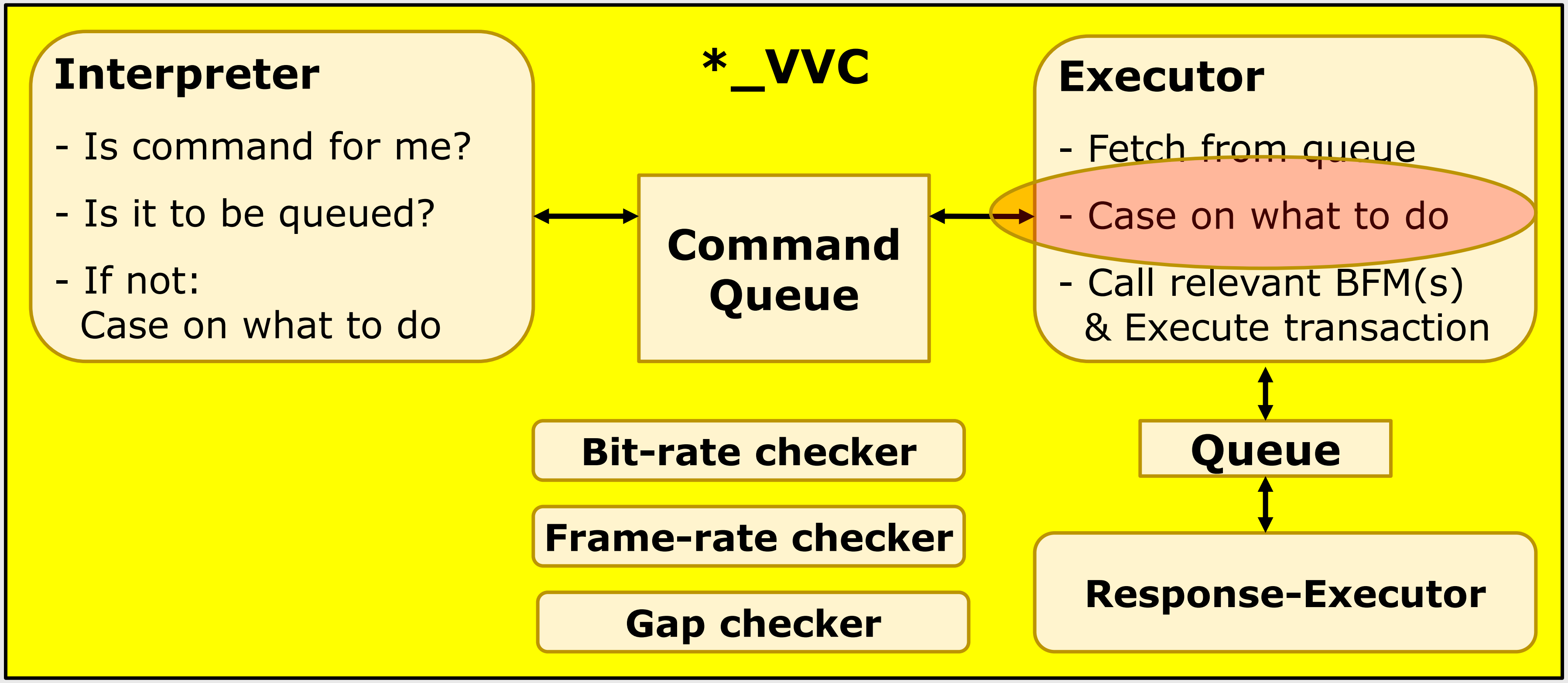
→ Standard control of parallel checkers

- Easy to handle split transactions

- Easy to add local sequencers

- Easy to add checkers/monitors/etc

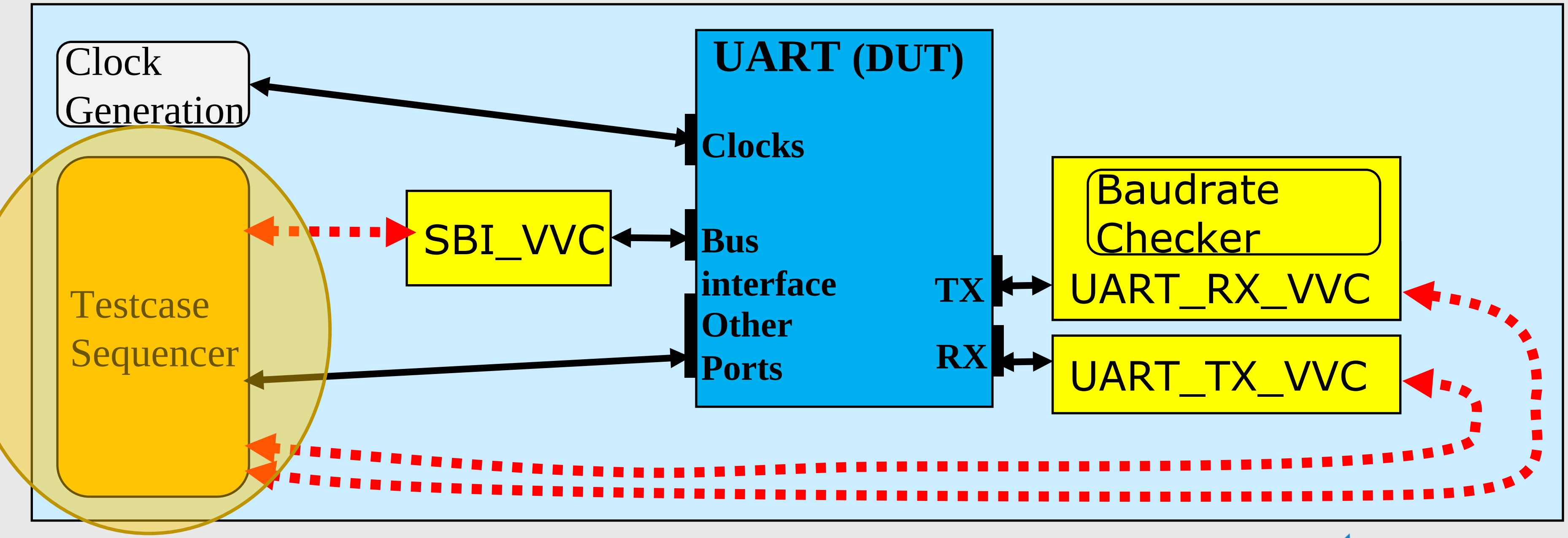
- Easy to handle out of/order execution



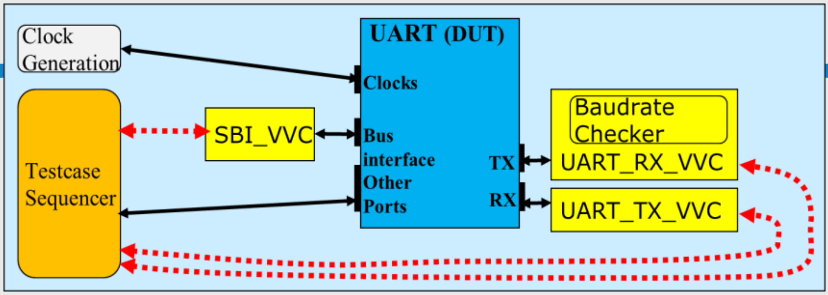
3: The test sequencer

(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:

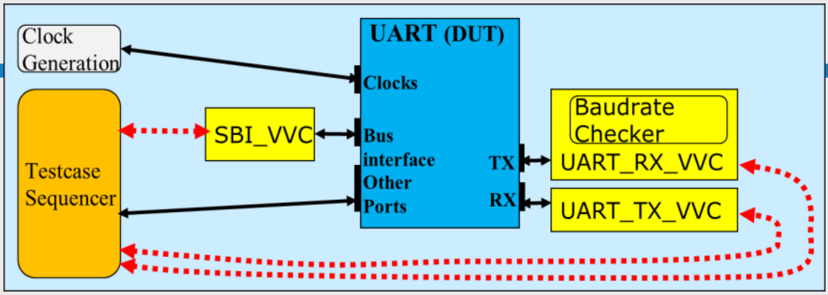
```
sbi_write(SBI_VVCT,1, C_ADDR_TX_DATA, x"A0", "Send byte UART TX");
uart_expect(UART_VVCT,1,RX  x"A0", "Check byte from UART TX");
uart_transmit(UART_VVCT,1,TX  x"A1", "Apply byte on UART RX");
wait for C_FRAME_PERIOD;
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "Check UART RX byte");
```

→ Standard command distribution syntax
→ Standard handling of multiple instances
→ Standard transfer of commands from sequencer 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")
```

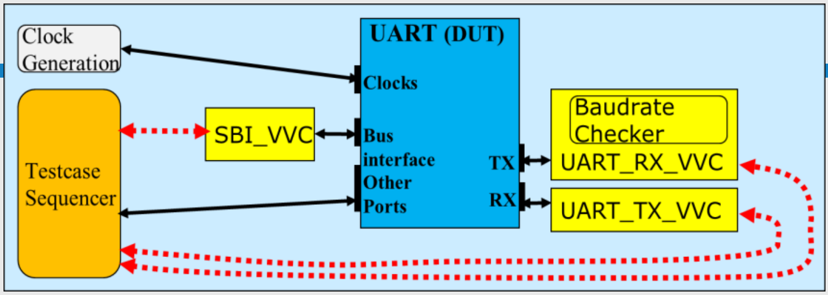
→ Standard synchronization between any process or VVC

→ 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”);
```

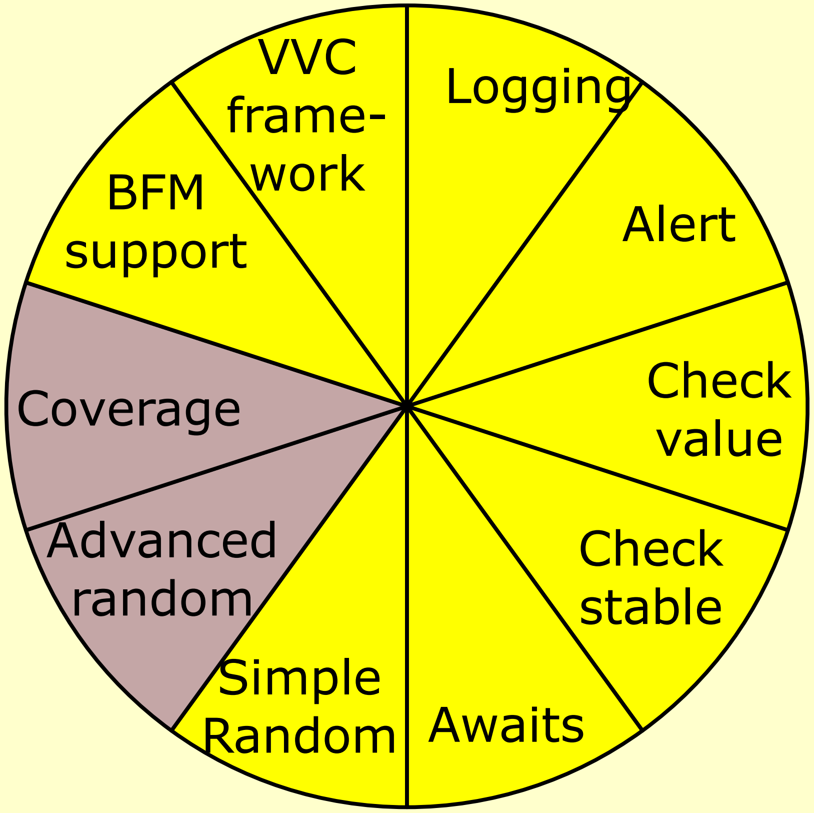
→ Standard set of common commands for all VVCs

→ Standard multicast and broadcast of common commands

Unified VHDL Verification Methodology

Included for handout version

UVVM vs **OSVVM** **(Randomisation and Coverage)**



Complementary
→ **Unified** VHDL Verif. Meth.

+ Various BFMs

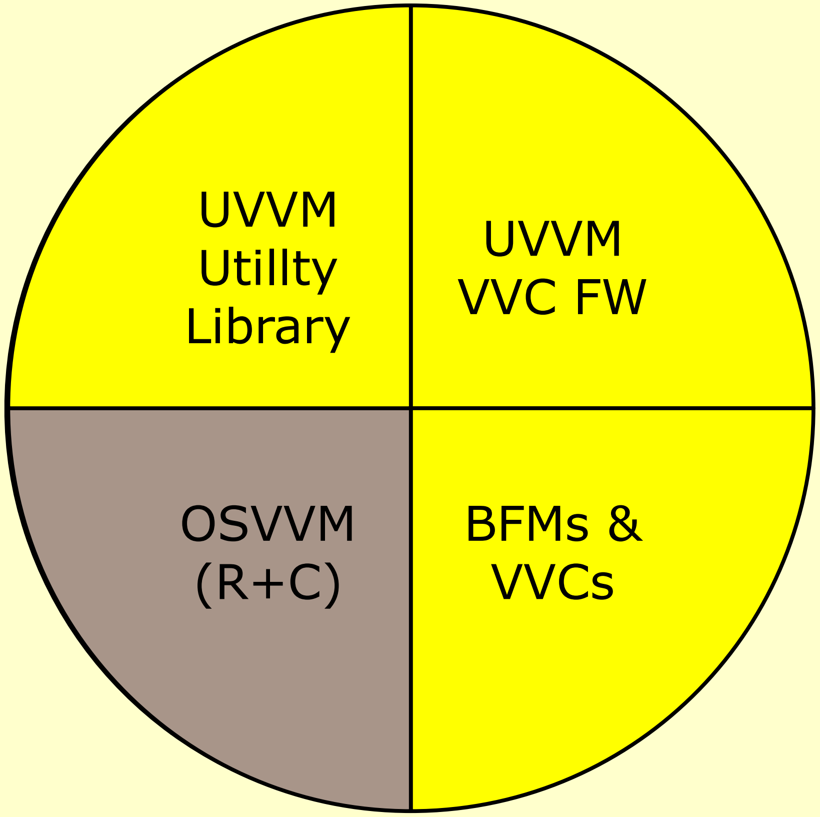
+ Various VVCs

AXI4-lite, AXI4-stream, Avalon MM, SPI, SBI, UART, I2C, ...

Unified VHDL Verification Methodology

Included for handout version

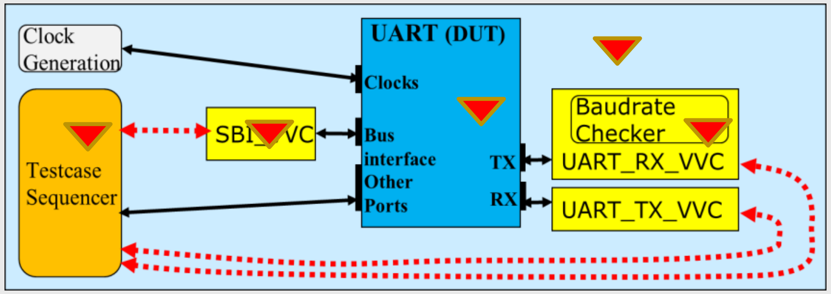
UVVM vs OSVVM (Randomisation and Coverage)

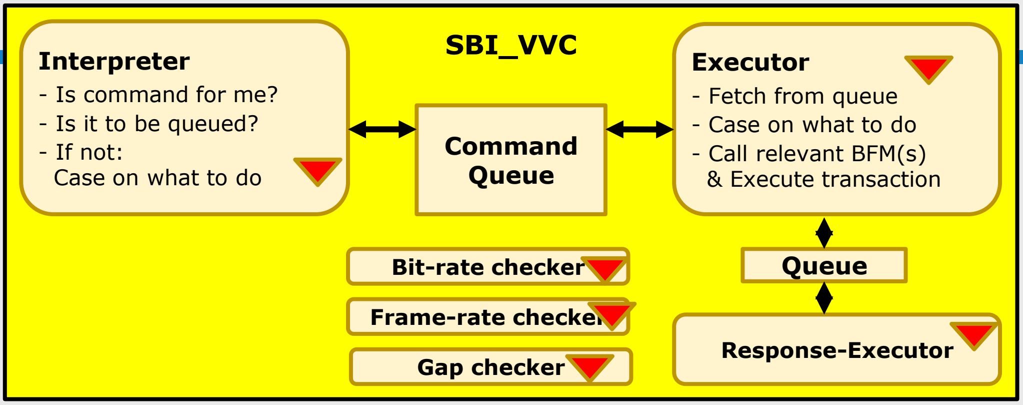


Complementary
→ **Unified** VHDL Verif. Meth.

Plug-in
OSVVM Random+Coverage
- with UVVM Utility Library
- inside UVVM BFMs
- inside UVVM VVCs
- in UVVM Test Sequencers

Randomisation and Functional Coverage - Using OSVVM





- Constrained random and Functional Coverage may be used anywhere
- May be started and stopped (manually or automatically)

```
uart_transmit(UART_VVCT,1,TX,   x"5A");
```

```
uart_transmit(UART_VVCT,1,TX,   RANDOM, C_BUFFER_2, 256);
```

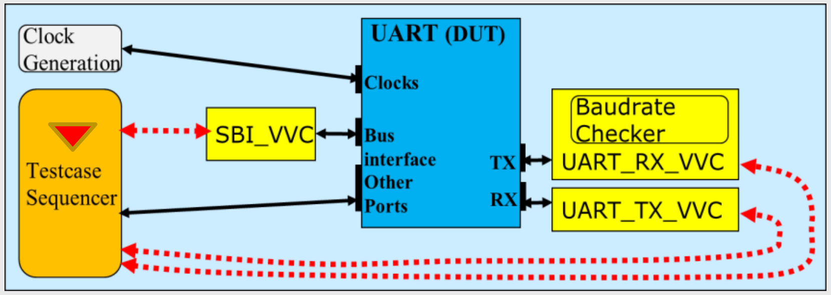
```
uart_transmit(UART_VVCT,1,TX,   FULL_COVERAGE, C_BUFFER_2);
```

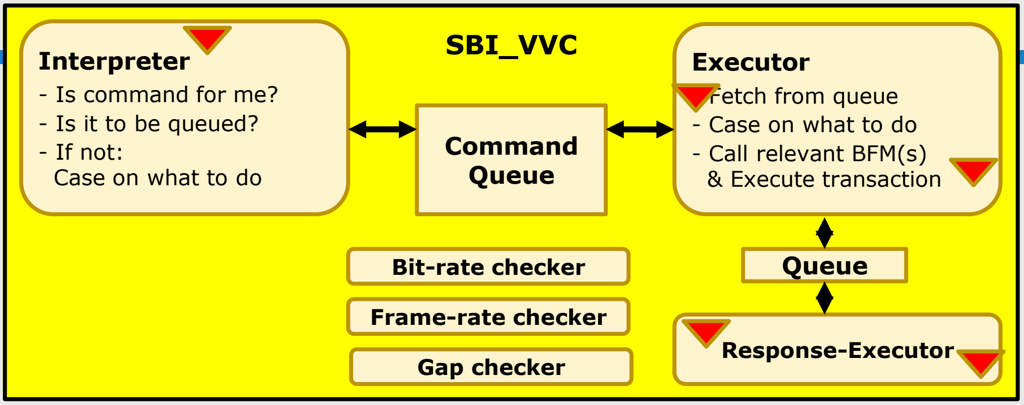
→ Standard command structure for any new command

→ Standard VVC architecture for executing any new command

Debugging Commands and new VVCs

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

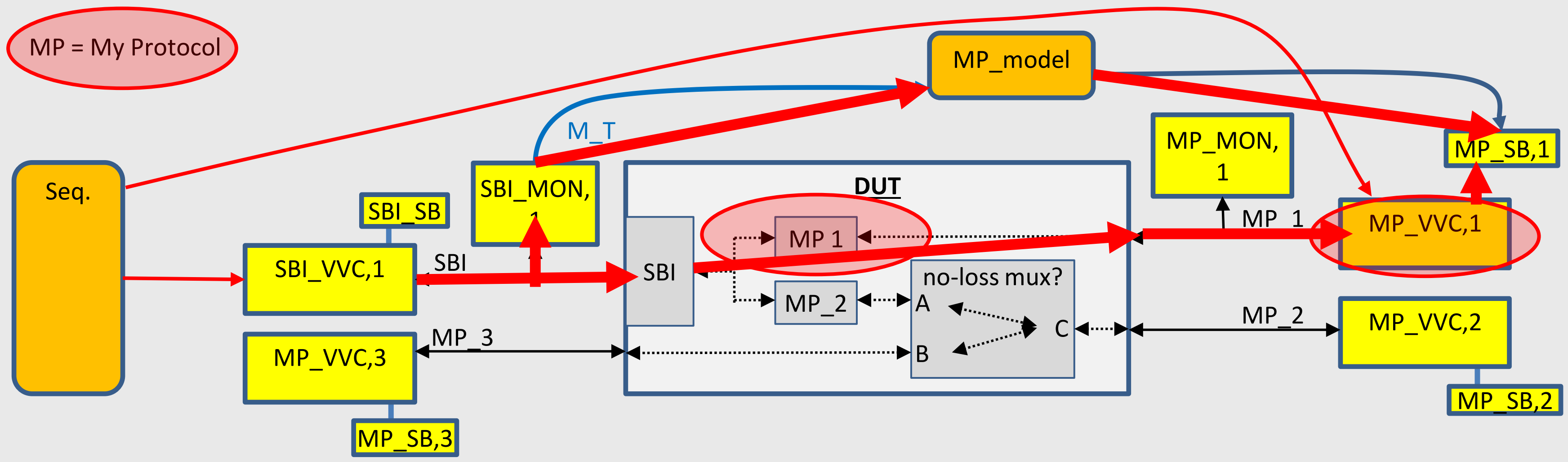
bitvis

The ESA extensions

- ESA (European Space Agency) sponsors new UVVM extensions
- Intention: Improve FPGA quality and verification efficiency

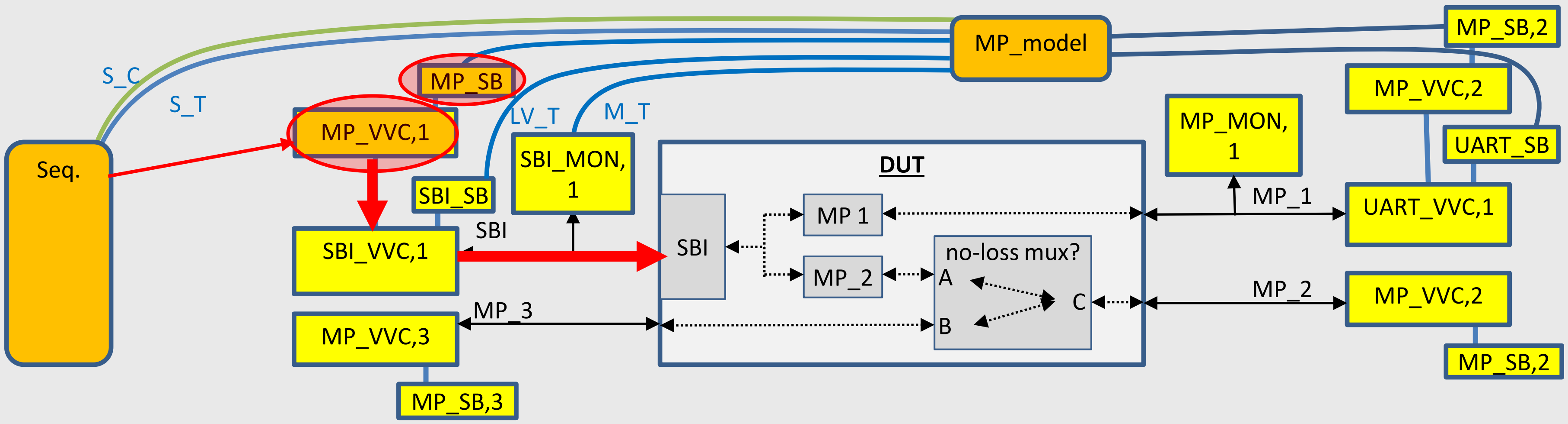
- The extensions
  - Scoreboarding
  - Monitors
  - Controlling randomisation and functional coverage
  - Error injection
  - Local sequencer
  - Watchdog
  - Controlling property checkers
  - Req. vs Verif Matrix (Test coverage)

ESA-UVVM: SB+Monitor



- Full functionality scoreboards
  - With generic types
- Monitor added to provide actual data to model
  - Actual data means data as seen on the actual interface
- Model **fetches** actual transactions from Monitor (model is not known to monitor)

Potential further extension



Currently evaluated

- Hierarchical VVCs
  - With their own Scoreboards
  - Sequencer can access any level at any time
- Sequencer can send transaction "objects" directly to model
- Model can fetch transactions directly from VVC
  - at any level

The results of the ESA extensions

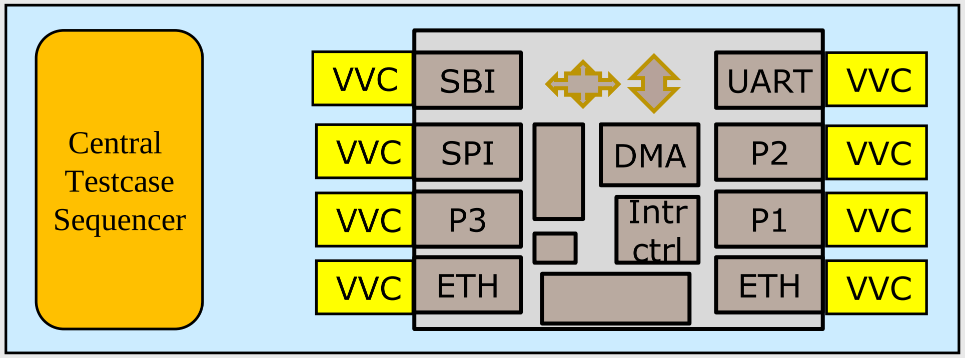
Included for handout version

- More automation using Monitors
  - (?) Plus option not to implement Monitors - to save time
  - (?) Send transaction "object" from sequencer/VVC to model
- An even more structured TB architecture
  - Full functionality scoreboards
  - Prepared integration and interfacing of models
  - (?) Hierarchical VVCs
- Structured error injection, watchdogs, etc...
- Standardised control of Constrained Random ++

➔ A unified full VHDL verification environment

➔ Contributions from the VHDL community...

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 BFMs and VVCs may easily be wrapped to UVVM

UVVM BFMs and VVCs may be used anywhere - exactly as is

Included for handout version

Wishful thinking

Wouldn't it be nice if we could ...	**UVVM**
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	✓

bitvis

Included for handout version

Benefits of standardisation

- 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**

Included for handout version

Roadmap

- Unprioritised further VVC roadmap
  - VVC Wishbone
  - VVC Avalon ST
  - VVCs for Clock generator and reset
  - more....
  - + Community VIPs
- UVVM VVC functionality
  - More examples on complex constrained random and coverage
  - More support for error injection and monitors
  - More advanced scoreboards
  - Standard solutions for pipelined transactions and out-of-order

Available UVVM Open Source BFMs & VVCs:

AXI4-lite
AXI4-stream
SBI
SPI
I2C
Avalon MM
UART
GPIO

UVVM is gaining momentum

- UVVM VVC Framework - Released February 2016
- Great feedback from users
- Recommended by Doulos for Testbench Architecture
- Being used world wide
  - with courses on 3 continents
- ESA (European Space Agency) sponsors extension

3-day course on 'Advanced VHDL Verification - Made simple'

- Munich, Germany, June 19-21 (coming on bitvis.no soon)
- Stockholm/Gothenburg, Sweden, Sept. TBD
- Ankara, Turkey, Oct/Nov TBD

More to come...

Info soon under www.bitvis.no

bitvis

bitvis

Make better testbenches - **and** save time

Let's start sharing VVCs

https://github.com/UVVM

Your partner for Embedded software and FPGA

bitvis
Quality in every bit

3-day course: **Accelerating FPGA VHDL Verification**

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 www.bitvis.no