bitvis

www.bitvis.no

Independent Design Centre for SW & FPGA/ASIC
- 16 designers (Embedded SW: 6, FPGA: 10)
- Specification, Design, Implementation, Verification, Test
- Methodology partner
- Sparring and review partner

Embedded software

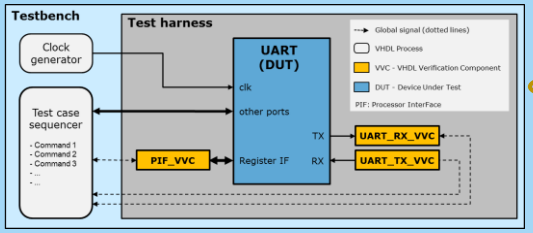
FPGA

Register Wizard
→ Freeware
➔ Fast and safe generation of registers: VHDL Bus IF, C, Doc, TB

FPGA and Digital ASIC Design
- Design Architecture & Structure
- Clock Domain Crossing
- Coding and General Digital Design
- Reuse and Design for Reuse
- Quality Assurance - at the right level

Accelerating FPGA and Digital ASIC Design:
→ **Oct 12-13, Stockholm**
→ Nov 2-3, Munich

Accelerating FPGA VHDL Verification:
→ Dec 6-8, Munich



UVVM Utility Library and
UVVM VVC Framework
→ Free and Open source
→ Improves TB efficiency, overview, maintainability and Reuse

FPGA VHDL Verification
- TBs & Verification essentials
- Structured TB from scratch
- BFM, TLM, VIP, VVC
- Assertions, randomisation, functional coverage
- Structure, Overview, Reuse

Verifying corner cases in a structured manner
- using VHDL Verification Components

FPGAworld 2016

bitvis

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.

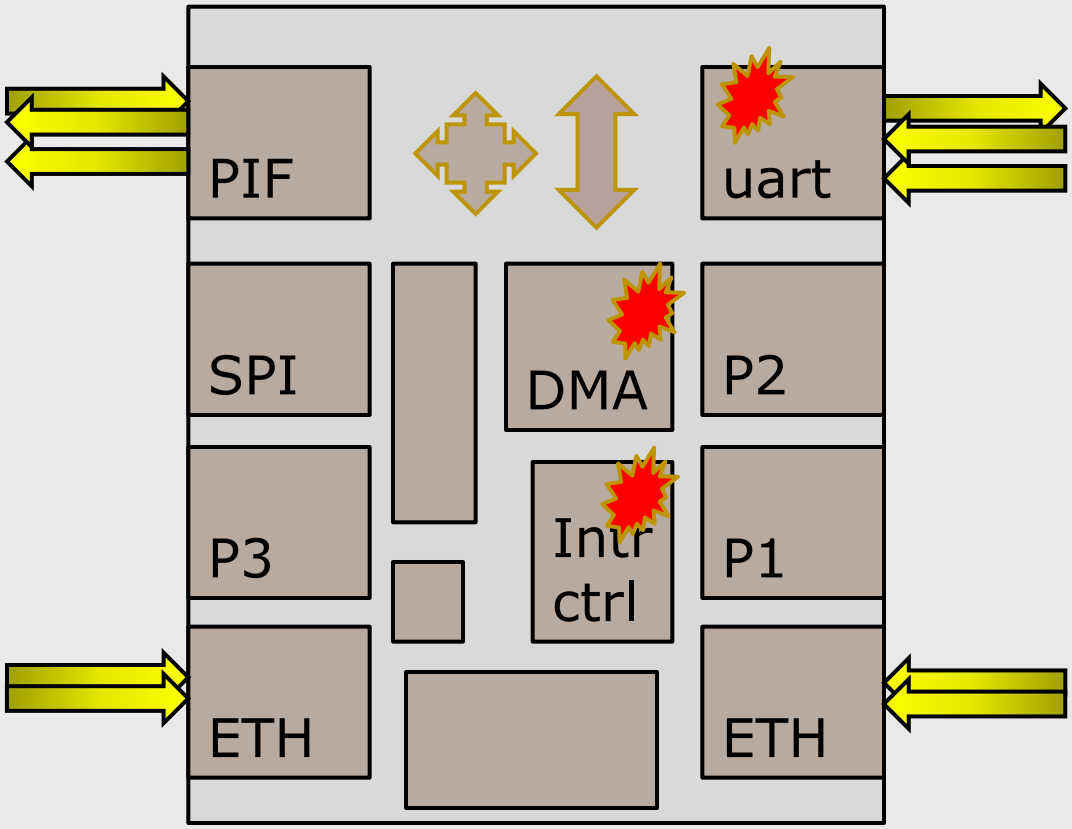
It is only a problem when the design doesn't work for this case.

AND this is detected late in the FPGA development

OR even worse - not detected until delivered

bitvis

Chip level functional scenario



Sequential testing

➔ Find some bugs?

➔ Then behaves fine

➔ Value related corners?

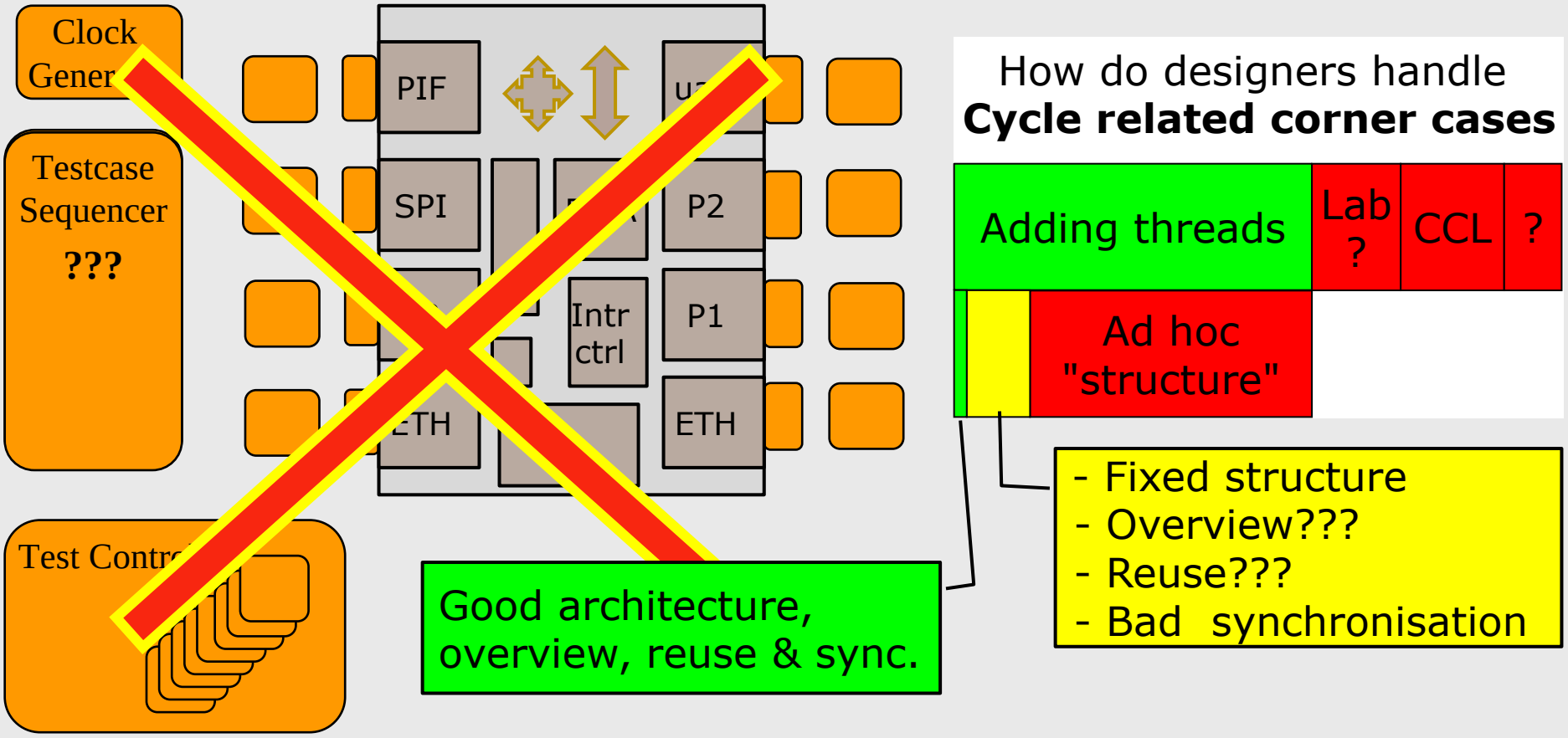
Parallel operation

➔ Multi-end cycle related corners

➔ Bugs are hard to find

bitvis

Typical testbench approaches



bitvis

Wishful thinking for a testbench?

In addition to normal and advanced stimuli and checking -

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

Is this feasible at all?

BFMs to handle interfaces

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

Exa … r

E.g.

BFM: A model or model set (or API)
for handling transactions on a physical interface.
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");
```

SBI: Simple Bus Interface
- Single cycle
- (Optional ready)
- Dead simple

bitvis

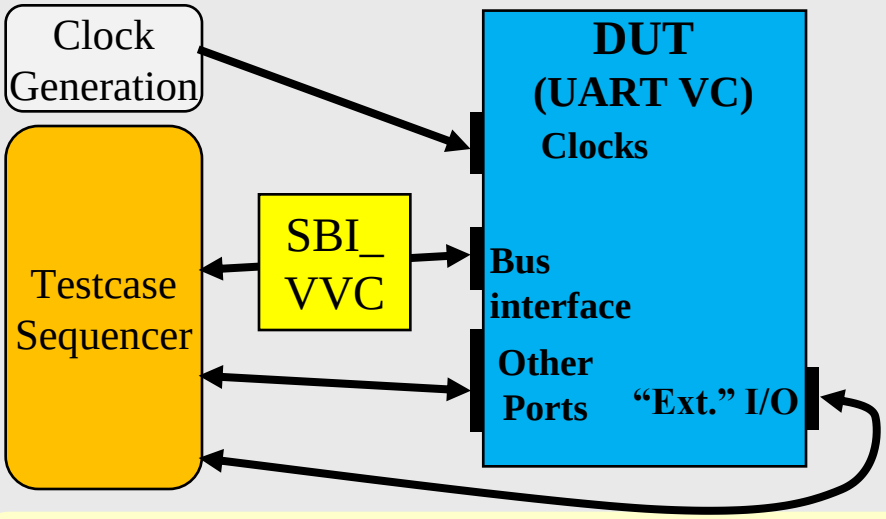
Parallelism & Encapsulation

- BFM procedures only allow sequential operation (in test sequencer)
- Parallelism requires BFMs in separate processes or entities
- Processes may only have one single thread of execution
  - Fine for simple applications or **really deep** insight & understanding
- Entities (= Components) allow Multiple active threads
  - → Simple handling of independent command and response
  - → May add dedicated processes for extended functionality
  - → Enables encapsulation
  - → Makes re-use far more efficient

→ Using Components is the best solution - exactly as for Design

➔ **'VHDL Verification Components' : VVC**

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

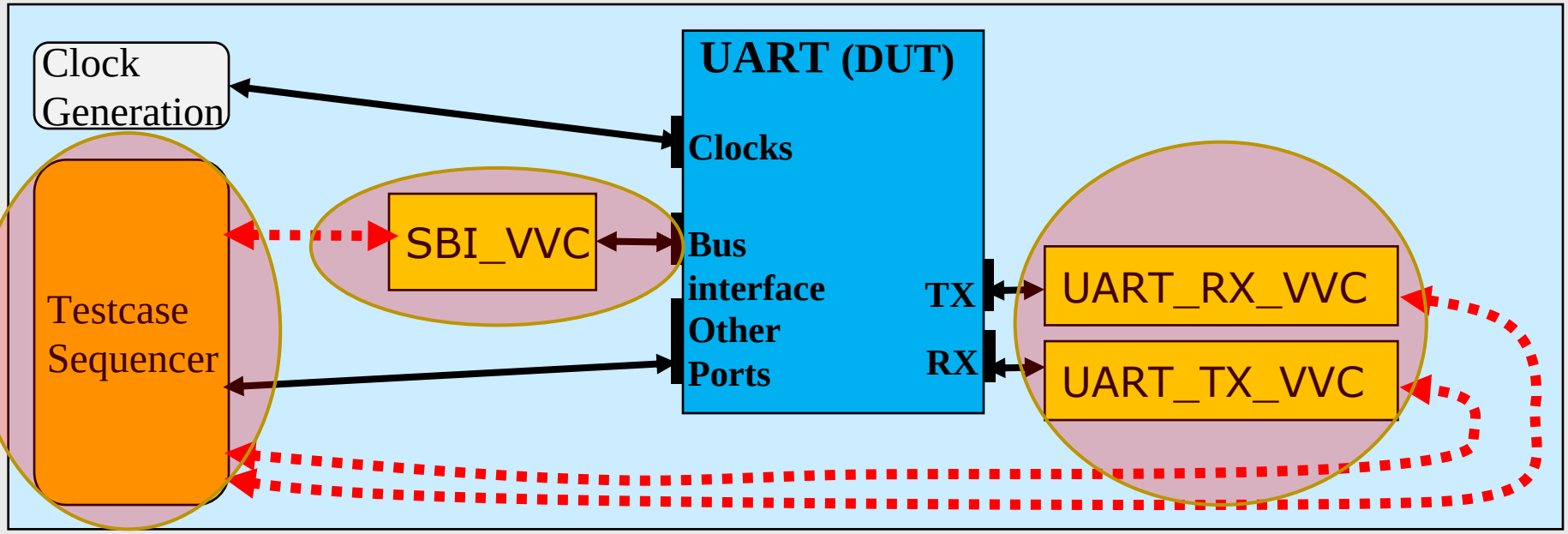
Sequencer command using VVC: sbi_`write(SBI_VVCT, C_ADDR_TX, x"2A");`

→ Results in above BFM being executed from VVC towards DUT

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

TB implementation & understanding - Three main areas

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

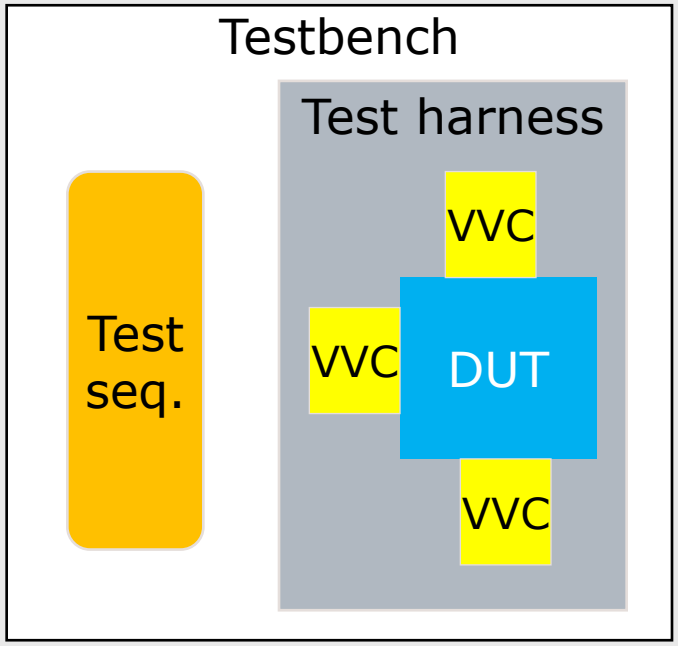


bitvis

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
- May have VVCs **anywhere**

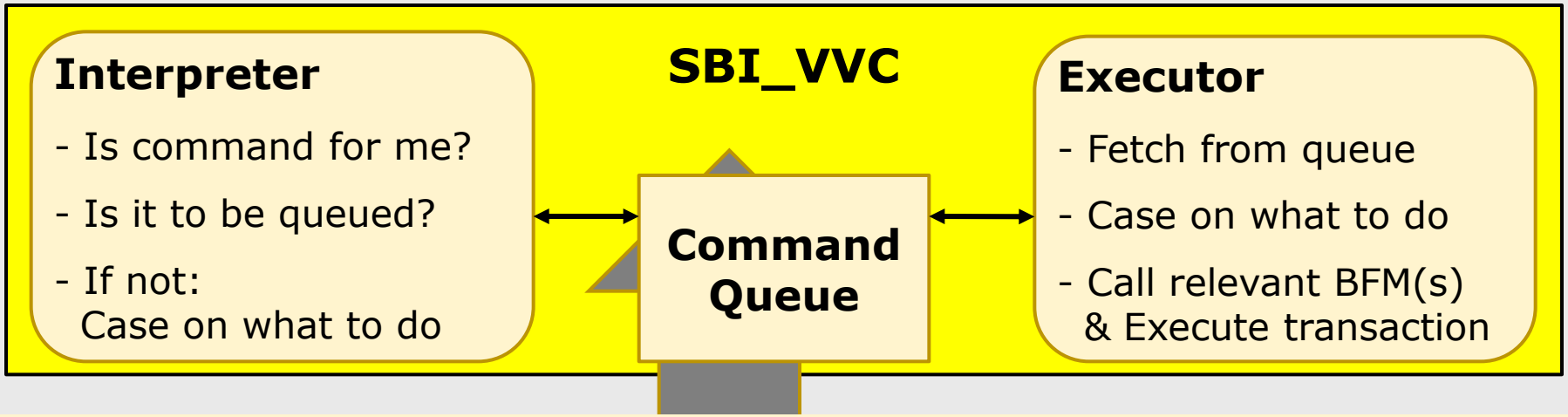


bitvis

2: VVC: VHDL Verification Component

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

- Now - what about these VVCs?



Same main architecture in every VVC

- >90% same code in Interpreters
- Same command queue
- 90% same code in Executors - apart from BFM calls

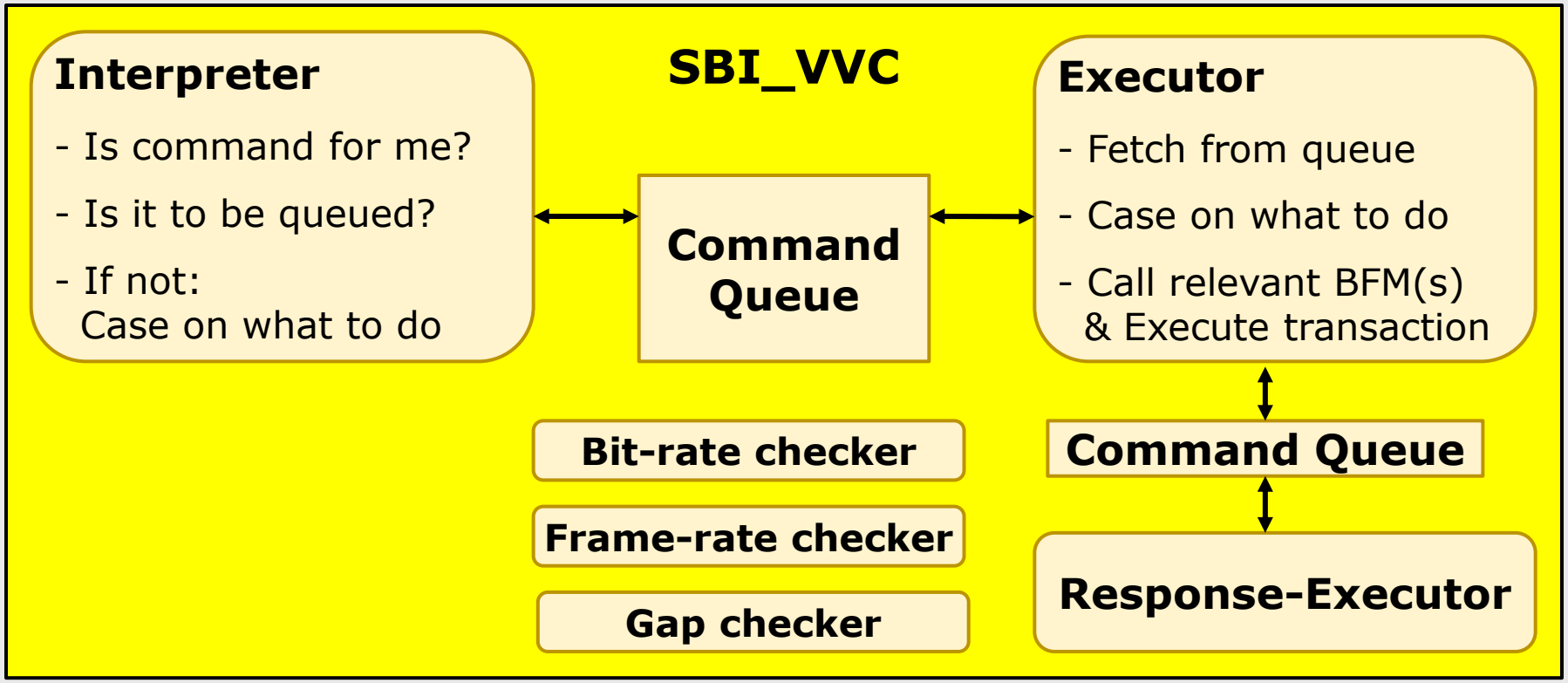
VVC Generation

UART BFM to UART_VVC:
less than 30 min

2: VVC: VHDL Verification Component

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

- Now - what about these VVCs?

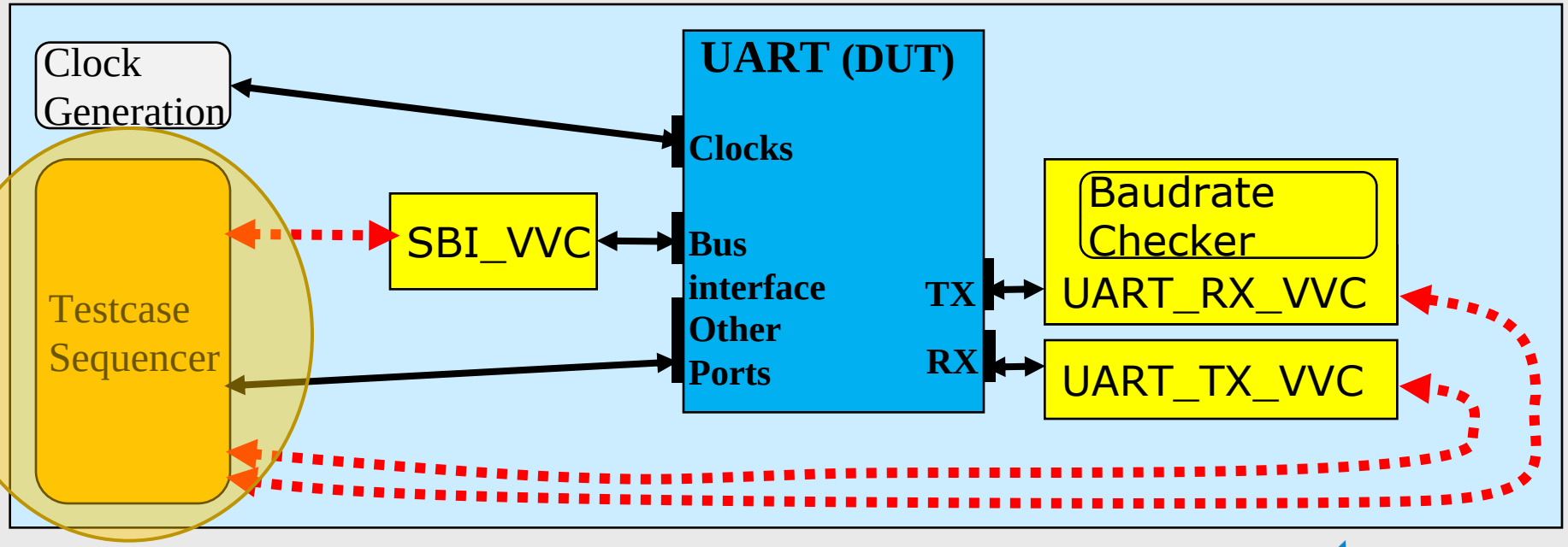


bitvis

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, ...



bitvis

Verification plan - In plain English (1)

1. A standard serial sequencer:
   - First Apply data to UART RX,
   - then Wait for RX interrupt,
   - then Read reg RX_DATA.

For each verification plan issue in English, show UVVM code in test sequencer

Pure sequential - Using UVVM 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 VVC Framework + VVCs

```
uart_transmit(UART_VVCT,1,  x"A1", "First byte on UART RX");
await_value(rx_empty, '0', 0, 12*bit_period, ERROR, message);
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "UART RX first byte");
-- await_completion(SBI_VVCT,1,  "Finish");
```

Verification plan - In plain English (2)

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

Parallel operation - using UVVM VVC Framework + VVCs

```
uart_transmit(UART_VVCT,1,  x"A1", "First 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 reg RX_DATA when this has just been up-loaded

Parallel operation - using UVVM VVC Framework + VVCs

```
uart_transmit(UART_VVCT,1,  x"A1", "First 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");
```

Verification plan - In plain English (4)

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

Parallel operation - using UVVM VVC Framework + VVCs

```
for i in -C_CYCLES_BEFORE to C_CYCLES_AFTER loop
  v_data := std_logic_vector(to_unsigned(100+i, 8));
  uart_transmit(UART_VVCT,1,  v_data, "First 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, v_data, message);
  await_completion(UART_VVCT,1,  "Finish before next transmit");
end loop;
```

Even better: Make the VVCs handle the sequences

```
sbi_write(SBI_VVCT,1, C_ADDR_TX_DATA, RANDOM_TO_BUFFER, 1, 256);
uart_expect(UART_VVCT, 1, RX, FROM_BUFFER, 1, 256);
```

Value related corner cases

- Direct testing
- Constrained random
  - Use std VHDL, Utility Library or OSVVM to generate values
  - Apply from test sequencer as follows:
    - 1. As time divided single accesses - as shown earlier
    - 2. As a chunk of single accesses distributed simultaneously
    - 3. As a single command to initiate multiple accesses from VVC
- C

Make your own super-procedure (re-cap)

```
uart_transmit_sbi_check(
    C_START_VALUE, -C_CYCLES_BEFORE, C_CYCLES_AFTER);
```

- C
- Work fine with Utility Library and VVC Framework
- Ext. Randomisation and Coverage packages may be used as is
- UVVM version of OSVVM will enhance Coverage functionality

bitvis

Debugging (1) - Verbosity ctrl

- Important for both the DUT, TB and VVCs
- UVVM built-in messaging simplifies TB/VVC debugging

Showing distribution and execution - together

```
2045  TB seq.(uvvm)   ->uart_transmit(UART_VVC,1,TX, x"AA"): . [15]
2045  TB seq.(uvvm)   ->await_completion(UART_VVC,1,TX, 2080 ns): . [16]
3805  UART_VVC,1,TX   uart transmit(x"AA") completed.  [15]
4005  TB seq.(uvvm)   ->sbi_check(SBI_VVC,1, C_A_RX, x"AA"): RX_DATA. [17]
4005  TB seq.(uvvm)   ->await_completion(SBI_VVC,1, 2080 ns): . [18]
4017  SBI_VVC,1       sbi_check(C_A_RX, x"AA")=> OK, read data = x"AA". RX_DATA  [17]
```

Debugging (2) - Follow command

Showing all log messages but for uart_transmit() only

Debugging the testbench of VVC
(Only showing uart_transmit)

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

Normal view: Showing executed commands only

```
3805  UART_VVC,1,TX   uart transmit(x"AA") completed.  [15]

4017  SBI_VVC,1       sbi_check(C_A_RX, x"AA")=> OK, read data = x"AA" RX_DATA [17]
```


bitvis

Wishful thinking? - Revisited

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 ☑

Levels of Freedom with UVVM

- You may use UVVM for all or parts of your TB
- You may use one or more UVVM compatible VVCs
  - and combine with any other approach
- You may use just a BFM - without the corresponding VVC
  - and combine with any other approach
- You may combine with OSVVM for all or parts of your TB
  - Using all or parts of OSVVM
- UVVM gives you a very well structured methodology
  - You can choose which parts you want to use

➔ UVVM is also the **'Unified VHDL Verification Methodology'**

UVVM is gaining momentum

- UVVM VVC Framework - Released February 2016
- Great feedback on LinkedIn FPGA/ASIC/VHDL groups
- Really good feedback from users (industrial and academia)
- Presented at various conferences:
  - FPGA-Forum in Trondheim, February
  - FPGA-Kongress in München, July

  - FPGAworld in Stockholm, September
  - NMI in Swindon, October
  - DVCon-Europe in Munich, October
  - Aldec Webinar, October/November

Next course

- Dec 6-8, Munich: Accelerating FPGA VHDL Verification
(In Cooperation with Trias Mikroelektronik Gmbh. A Mentor Graphics distributor)

Summary

- UVVM is Free & Open Source
  - the only openly available solution for a good TB architecture
- UVVM is setting a Standard
- Several free (open source) VVCs available - and more to come
  (AXI4-Lite, AXI4-Stream, Avalon MM, UART, I2C, SPI, SBI, GPIO)
- UVVM structure yields major benefits
  - Overview, Readability, Maintainability, Extendibility
  - Reuse at all levels

UVVM ➔ Yields better quality - or reduces risk

UVVM ➔ Significantly reduces verification time

This slide was Added after presentation

Accelerating FPGA and Digital ASIC Design

- Digital design for FPGAs and ASICs has a huge improvement potential with respect to development time and product quality.
  A lot of time is wasted on inefficient design and lack of awareness and knowledge of the most critical digital design issues. This also seriously affects the quality of the end product. The really good thing is that this huge improvement potential can be realised just by making a few important changes to the way we design.
- **The most important design related issues to improve are:**
  - Design Architecture & Structure
  - Clock Domain Crossing
  - Coding and General Digital Design
  - Reuse and Design for Reuse
  - Timing Closure
  - Quality Assurance - at the right level
- **These issues are the main subjects of this course**
  See www.bitvis.no under 'Events' for more info
  (or see one course example here: http://bitvis.no/events/accel-fpgaasic-design,-berlin-2016/)

This slide was Added after presentation

Accelerating FPGA VHDL verification

- **On average half the development time for an FPGA is spent on verification.** It is possible to significantly reduce this time, and major reductions can be accomplished with just minor adjustments. It is all about Overview, Readability, Maintainability and Reuse at all levels – and you achieve all of this with the right methodology and a good structured architecture.
- **Agenda**
  - Making a simple VHDL test bench step-by-step
  - Using procedures and making good BFMs
  - Applying logs, alerts, value and stability checkers, awaits, etc...
  - Making an advanced VHDL test bench step-by-step
  - Assertions, randomisation, constrained random, coverage, debuggers, monitors
  - Verification components and testbench architecture for advanced Verification
  - Verification reuse and preparations for reuse
  - Making testbenches as simple as possible – adapting to the DUT complexity
  - Structuring, Debugging, Overview, Maintainability, Extendibility
  - Examples and labs using UVVM
- **These issues are the main subjects of this course**
  See www.bitvis.no under 'Events' for more info
  (or see one course example here: http://bitvis.no/events/accel-fpga-verifi,-berlin-2016/)

bitvis

Additional free tool from Bitvis

This slide was Added after presentation

Register Wizard

- Free tool for automatic generation of
  - Full VHDL register interface (SBI)
  - VHDL testbench for register interface
  - C header file
  - Documentation (Register overview tables)
- All from a single register description source file (JSON)
- Download and info under www.bitvis.no

bitvis

Verifying corner cases in a structured manner
- using VHDL Verification Components

Thank you

Your partner for Embedded software and FPGA