Budapest University of Technology and Economics

Faculty of Electrical Engineering and Informatics

Functional Verification Basics

The concept of functional verification

Self-checking testbenches

Qualifying the verification process

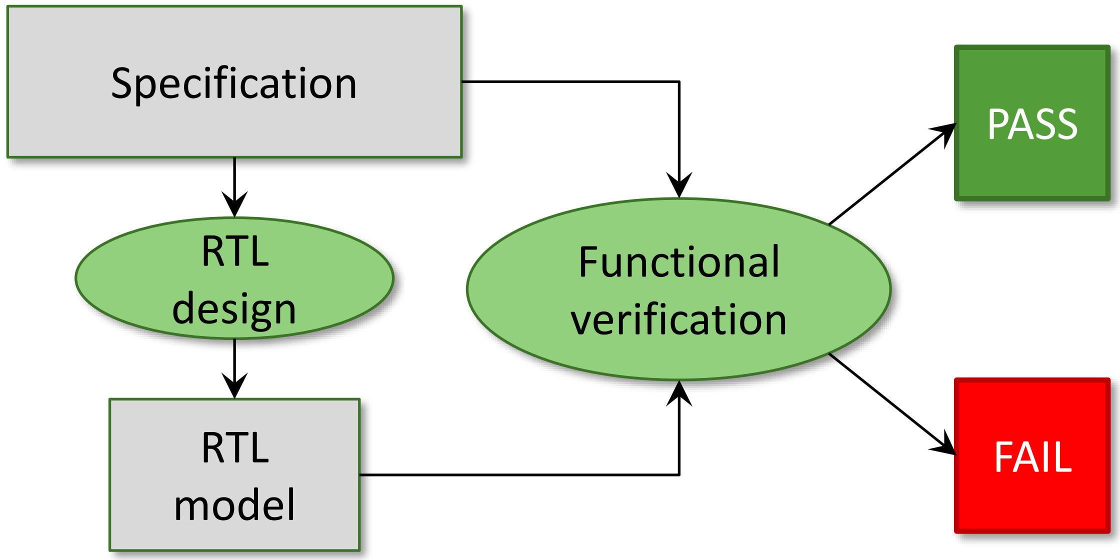
Reusability in verification

Dr. Lázár Jani, Dr. Péter Horváth

Department of Electron Devices, 2022

The objective of functional verification

- Verifying that the **HDL model** fulfills the requirements from **functional** point-of-view.
  - „**HDL model**”: Only the abstract model is checked, intended circuit structures and synthesis issues are not investigated.
  - „**functional**”: No physical characteristics are taken into account. E.g. verification of timing characteristics is not part of functional verification.



The significance of functional verification

- The **complexity of verification infrastructure increases more rapidly** than the complexity of the verified entity itself.

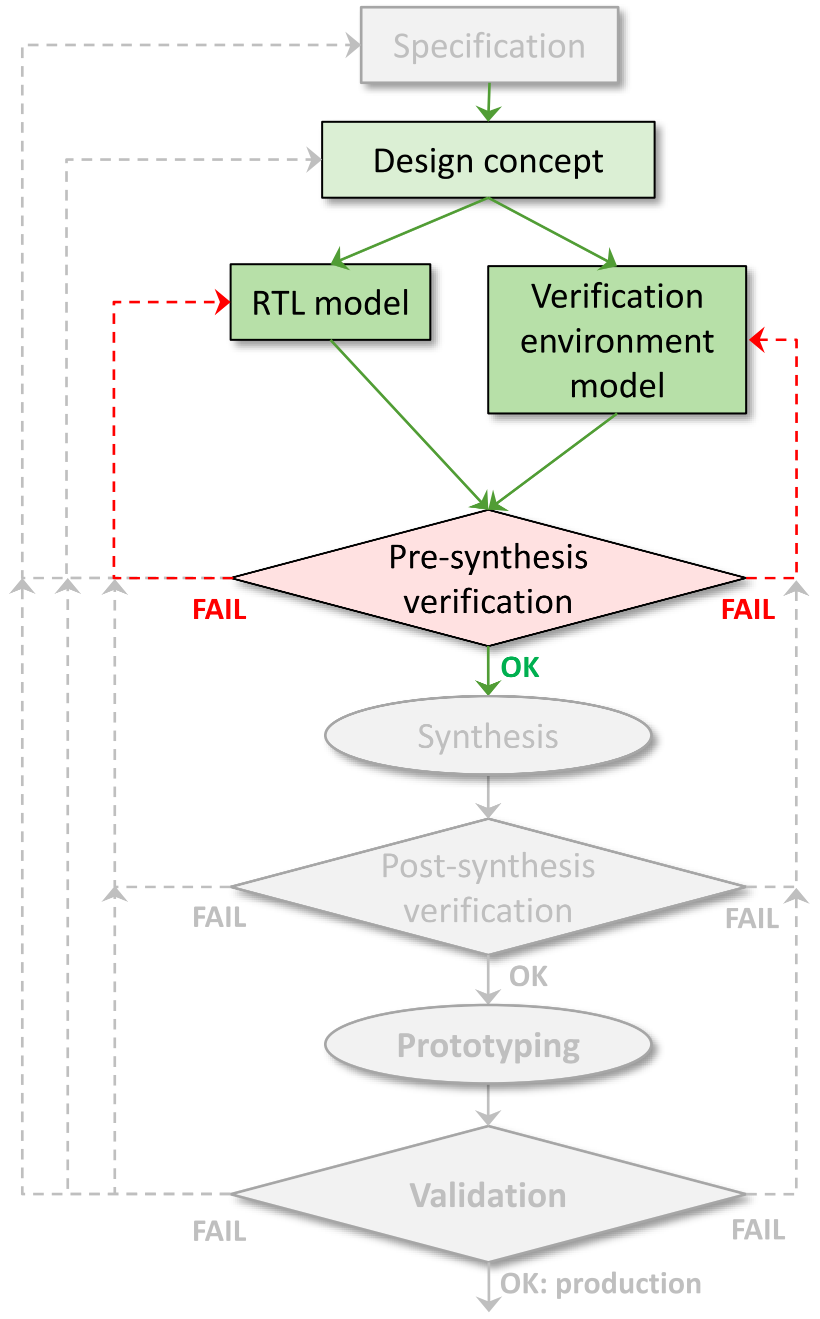
- In case of complex RTL IP cores
  - **70% of the NRE** goes into verification
  - the number of **verification engineers** is **double** that of RTL designers
  - only 20% of the **code base** is part of the synthesizable RTL model**, 80%** describes the **verification infrastructure**

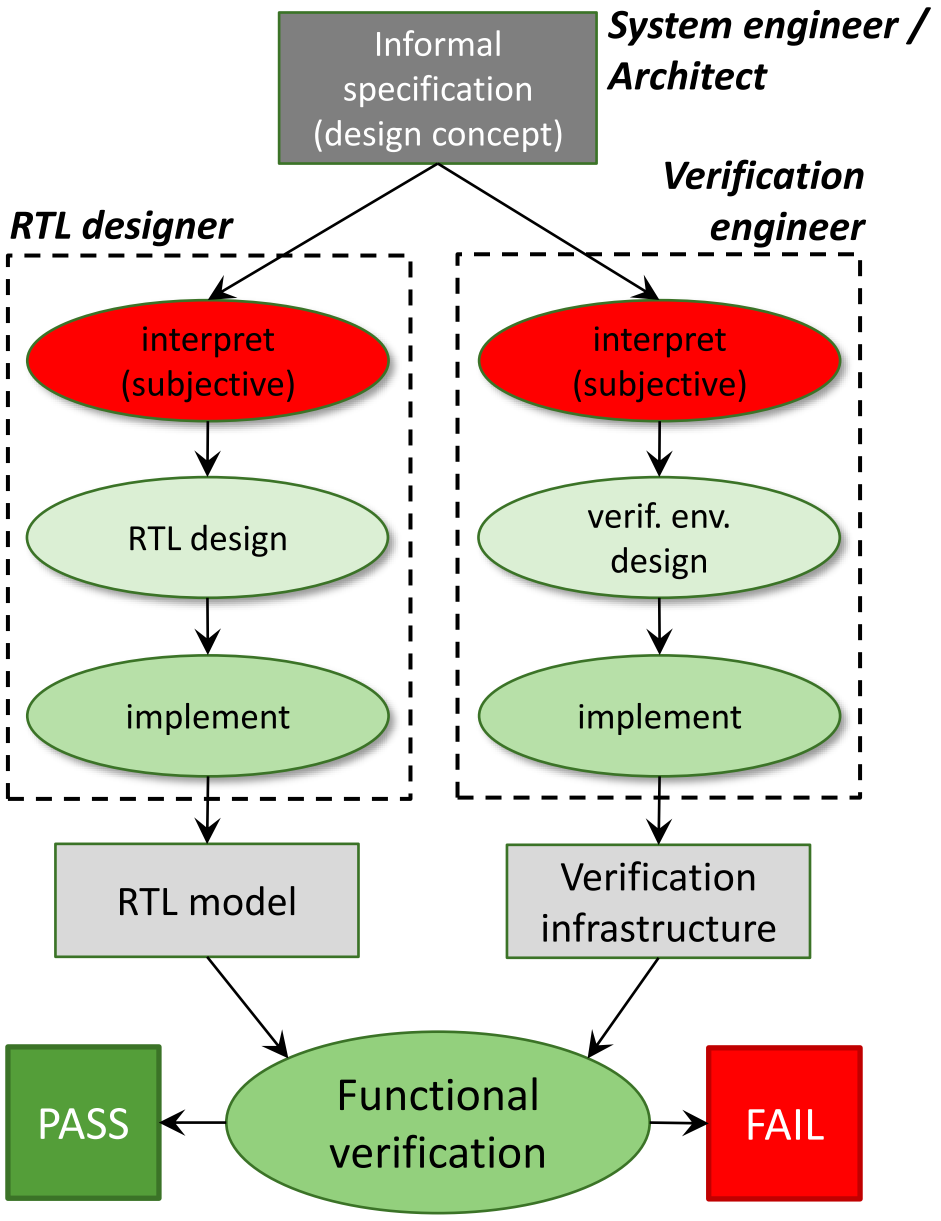
„Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

Brian W. Kernighan, 1974

- **BUT! Verification engineers do not need to find the causes of bugs!**

Functional Verification Basics

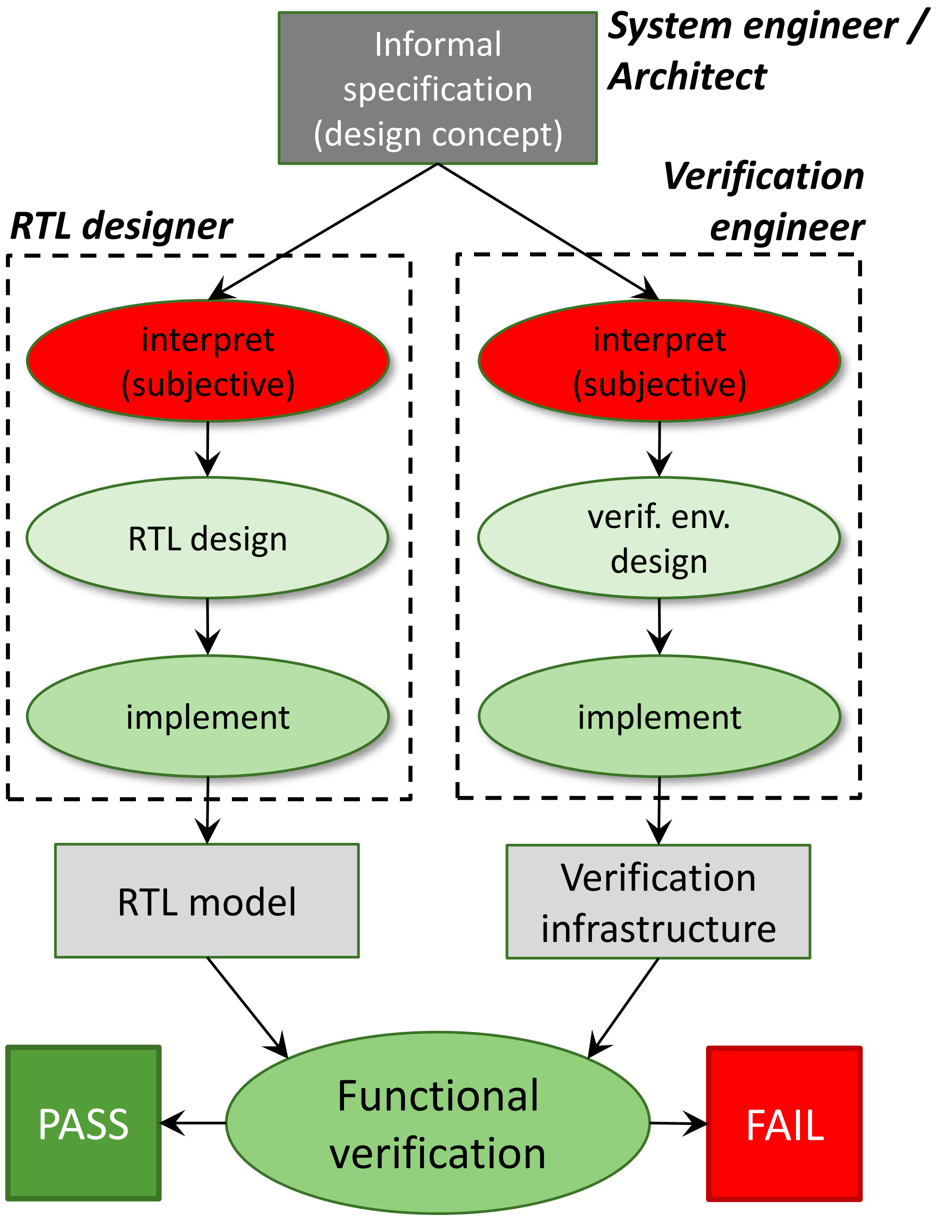




The „who's to blame" game

- Specification, RTL design and verification are done by **separate people / groups**
- Possible causes of failing verification process
  - The **RTL model** is incorrect
  - There is a flaw in the **verification infrastructure**
  - The **specification** is inaccurate, ambiguous

- The objective is to **find discrepancies → improve the design**



Functional verification and accessibility

Types of verification based on accessibility

the RTL designer usually takes part

- **White-box verification**
  - the *HDL model is known* to the verification engineer

not performed by the designer *(at least it shouldn't be…)*

- **Grey-box verification**
  - only *limited access* to the HDL model
  - debug/error-injection interfaces
- **Black-box verification**
  - the *HDL model is not known*
  - only the primary interfaces can be used to interact with the module

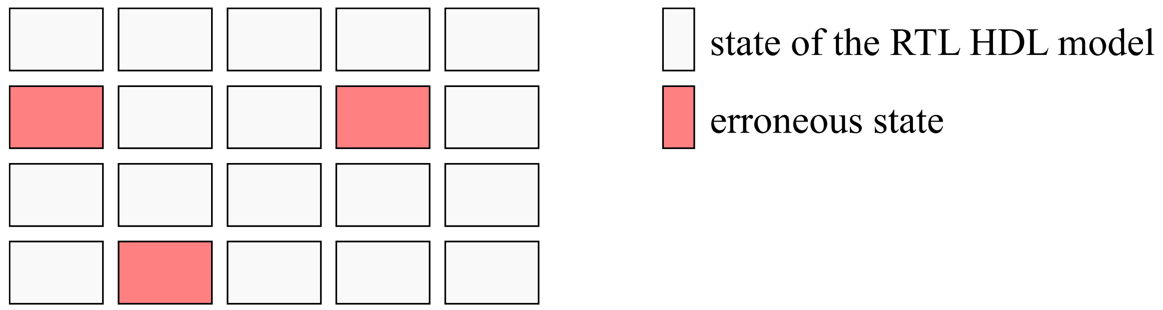
increasing controllability and observability

Types of verification based on accessibility

- **White-box**
  - It is easy to locate the problematic details. The critical states are known and they can be reached easily. The verification infrastructure is **not reusable**.
- **Black-box**
  - The verification infrastructure (e.g. excitation patterns, reference models, etc.) are reusable. It is hard to **induce critical circumstances** within the module. The *observable error* and the *cause* can be **very far from each other** in space and time as well.
- **Grey-box**
  - Some implementation details are known by the verification engineer
  - Some internal signals (e.g. FSMs' state registers) are accessible through the user interfaces
  - Technically, it is the mixture of the white-box and black-box, from advantages and disadvantages point-of-view.

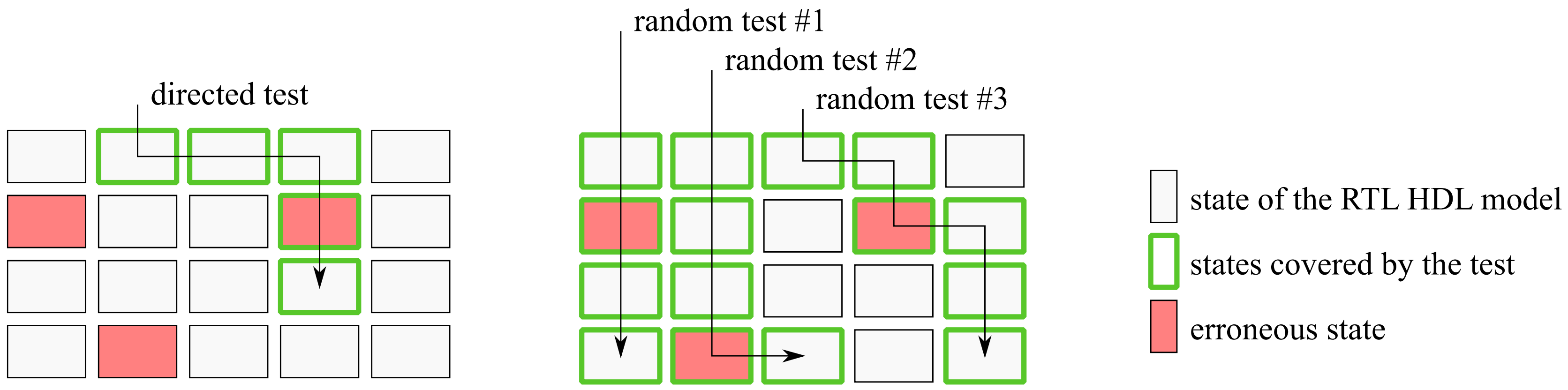
Stimuli generation

- **State space**: All possible *combinations* of all *possible values* of the *storage elements* and *input vectors* within the RTL HDL model. → **astronomical number of states**…
- Some of the states are **erroneous**; **incorrect behavior** can be observed
- Functional verification shall find the erroneous states
  - Usually it cannot be proved that the model is correct (state space is too large)
  - Verification must be continued, until the next revealed bug *„doesn't worth it"*



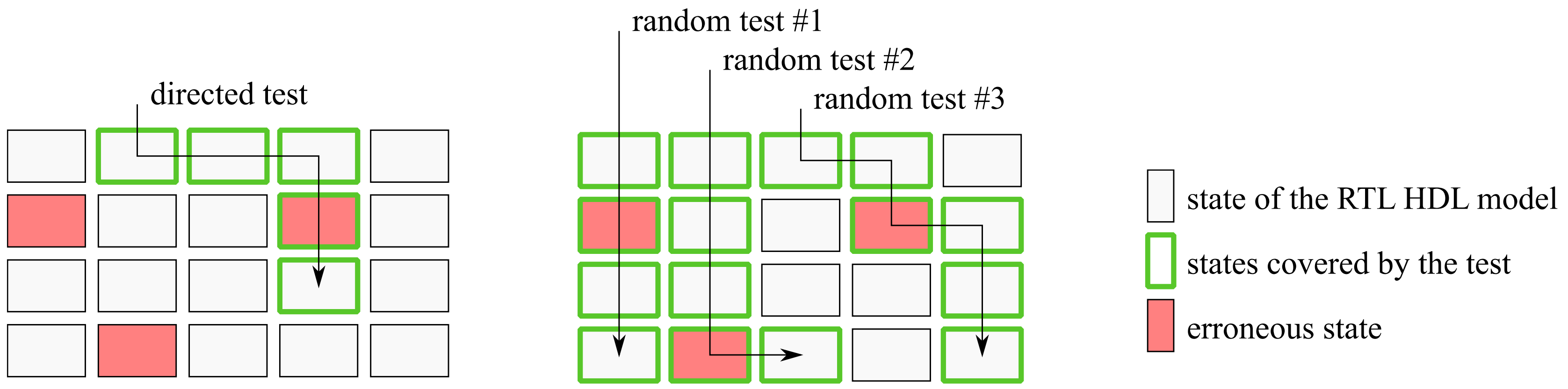
Stimuli generation

- **Directed tests**: Specific excitation patterns are manually generated, according to the requirements
  - it can only find the bugs, which are specifically expected by the verification engineer
- **Random tests**: Efficiency can be improved by **generating** many *„similar”* excitation patters **with random data**.
  - **Constrained random tests**: Random testing efficiency decreases as the number of states increases. Limiting the state space improves the efficiency of random testing.



Stimuli generation – example

- DUV: an RTL model **dividing two unsigned numbers**
- Stimuli generation practice
  - **Directed** tests for **initial** functional checks: checking reset, host interface hand-shake, enable signals, etc…
  - Automated **random tests** for **improving** state **coverage** efficiently: generate test cases using the host interface excitation pattern with many pairs of random operands
  - **Directed** tests for covering **corner cases**: check the „exceptional" behavior by formalizing a directed tests with the divisor set to zero.



Self-checking testbenches

Beyond the waveform – increasing the automation

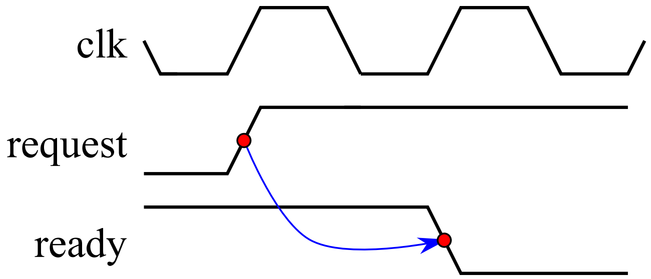
- The excitation patterns are generated **manually**
- A model's responses are checked in the **waveform**

- Problems
  - As complexity increases, the waveform becomes **unmanageable**
  - Waveform analysis is very **time-consuming**
  - In case of RTL model changes, **ALL** responses shall be **checked again!**

- Solution: increase the **automation**!
  - Self-checking testbenches
  - Automated regression testing

Beyond the waveform – increasing the automation

- Self-checking testbenches & regression testing
  - the testbenches include **code snippets**, which **observe** the DUV’s output(s) and **check the behavior** against the specification (the checks are high-level, **formal** models of the specification)
  - if the RTL model is changed (e.g. bugfix), all tests shall be performed again (regression testing) to ascertain that **previously checked features are intact**
  - with self-checking testbenches, **regression testing can be done automatically** by scripts



```
-- check #2: After generating a request, the ready output
-- of the FSM shall become deasserted at the next rising
-- edge of the clock.
L_CHECK_2: process
begin
  wait until rising_edge(request);
  wait for clk_period + 1 ns;
  if ( ready /= '0' ) then
    report "-----------------------> check #2 FAIL";
    wait;
  end if;

  report "-------------------------> check #2 PASS";
  wait;
end process;
```

End of topic

Key concepts

- The digital design flow relies on the correctness of the RTL HDL models – functional verification is an **essential**, but **time-consuming** (expensive) process
- Three players; system engineer / architect, RTL designer, verification engineer – functional verification is *about to achieve consensus* among them *regarding the functionality* to minimize failure probability
- To increase verification productivity
    - **automated regression testing** is unavoidable
    - **self-checking testbenches** are needed

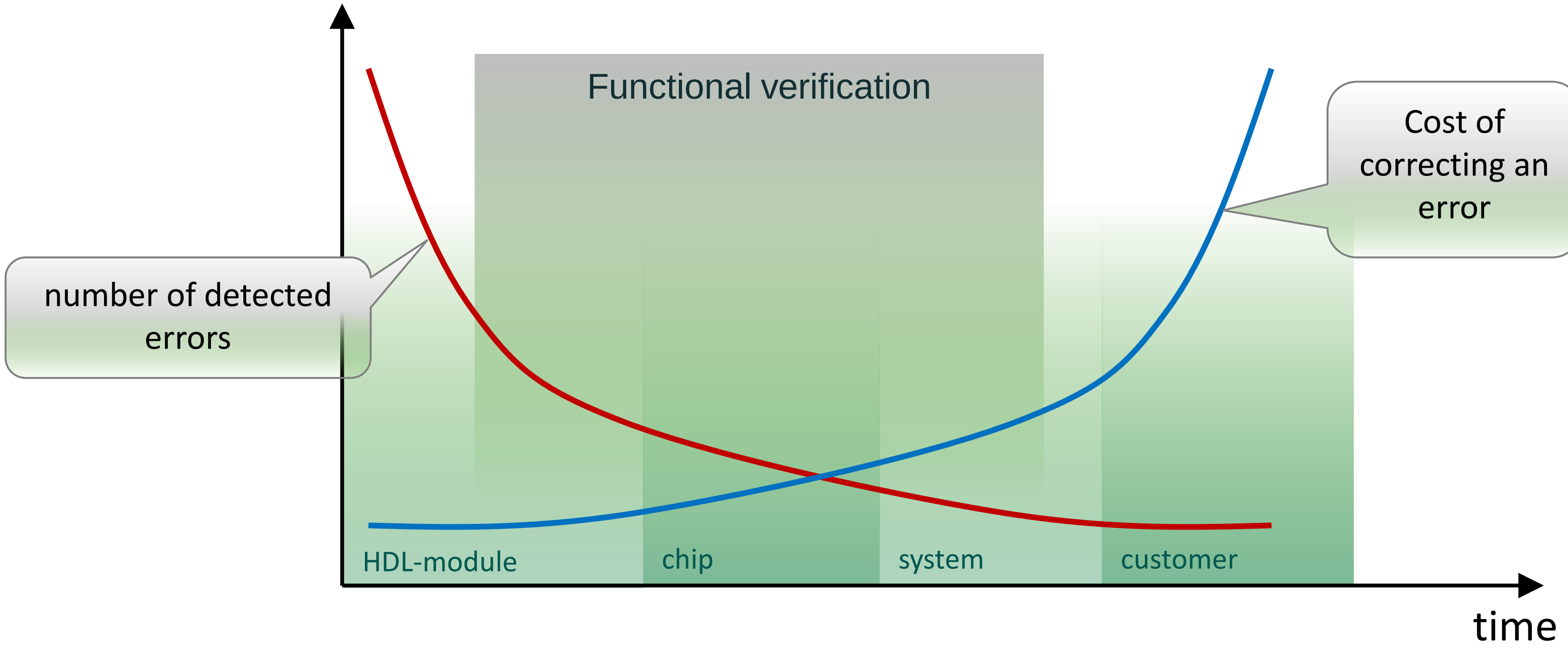
Questions

- Who are the three stakeholders in the design flow? What is the purpose of the functional verification?
- What are the three types of verification from the accessibility perspective? What is the main difference between them?
- What methods are available to apply stimuli to a DUV (design under verification)?
- What tools and methods are available for the verification engineer, to check the behavior of the DUV, and if it works as expected? Is it possible to automate this process?

QUALIFYING THE VERIFICATION PROCESS

Qualifying the verification process

- Functional verification **cannot prove** if a design is *"correct"*
  - What terminates the process?
- There are always some limiting factors
  - Time
  - Money



Qualifying the verification process

- There is a need to *'measure'* the **verification progress**
- Number of *discovered errors per time period*?
    - Not an objective measurement
    - But it make sense when other metrics are met
- Some objective metrics, e.g. code coverage?
  - Code coverage percentage does not correlate well with discovered errors
    - Errors still can be revealed after achieving 100% code coverage
    - But low code coverage indicates that the test sequence should be extended
  - Functional coverage
    - Which **features of the design is verified** by simulation?
    - Features are **determined from the specification**
    - The coverage percentage can be collected automatically but the features are derived from the specification by an engineer (=human)

Qualifying the verification process

- There is more than one type of **code coverage**
  - Statement (assignment, instantiation)
  - Branch (if … else …)
  - Condition (which condition triggered the branch)
    - if x=3 or y=3 then …
  - Expression
    - a <= (b or c) and (d or e);
  - Finite State Machine (FSM)
    - State coverage
    - State transition coverage

Qualifying the verification process

- Code coverage can be collected automatically during simulation
- Achieving 100% code coverage is **almost impossible**
  - In some cases it may be **unnecessary**
  - Statement cov.: e.g. generic parameters and their effect on the DUV/circuit
  - State transition coverage: async reset from every other state
- Exclusion to code coverage may be added if it is justified
  - Every exclusion needs to be explicitly justified!
- Spoiler: achieving ~100% code coverage does not mean that the RTL model is bug free

Qualifying the verification process

- **Functional coverage** collected semi-automatically
  - Which features are **covered by simulation**?
- What is a feature? -> determined by an engineer, from the specification
  - e.g. the module shall have UART transceiver with specific baud rate, etc.
- Coverage can be assessed by implementing checks, that verifies some part of the feature
  - A set of checks can verify a feature
- Simulator tool may help collecting the necessary information
  - e.g. QuestaSim

Reusability in verification

Reusability in verification

- How to **lower the verification effort**?
  - **Create and reuse** reusable verification components ☺
- What makes a component reusable?
  - The component models some *standard behavior*
    - CRC/parity calculation
    - UART frame generation
    - External devices' behavior model
  - Basically a component can be reused if it implements some **non design specific behavior** and it **has standard interfaces**
- The bigger the reusable components, the better…
  - Implementing a parity bit calculator is not a huge effort

Reusability in verification

- Test sequence generating a single UART frame

```
(…)
rx <= '0'; wait for 8.67 us;  -- start bit
rx <= '1'; wait for 8.67 us;  -- bit 0
rx <= '0'; wait for 8.67 us;  -- bit 1
rx <= '1'; wait for 8.67 us;  -- bit 2
rx <= '1'; wait for 8.67 us;  -- bit 3
rx <= '0'; wait for 8.67 us;  -- bit 4
rx <= '1'; wait for 8.67 us;  -- bit 5
rx <= '0'; wait for 8.67 us;  -- bit 6
rx <= '0'; wait for 8.67 us;  -- bit 7
rx <= '1'; wait for 8.67 us;  -- parity bit
rx <= '1'; wait for 8.67 us;  -- stop bit
(…)
```

- Hand-crafted frame
  - Low reusability
  - Replicating the test sequence -> Copy & Paste
  - Error prune

Reusability in verification

- Better approach is to implement *"something"*, that will **generate** the UART frame
  - A simple *"function call"* should replace the hand-crafted mess
  - VHDL: **procedure**, SystemVerilog: **task**
- 'Bus' functional model (BFM)
  - **Replicates** the *UART signal waveform*

```
procedure uart_8o1_transmitter_bfm (
    data:                   in  std_logic_vector (7 downto 0);
    signal tx:              out std_logic
) is
    variable P:     std_logic := '0';
begin
    tx <= '0'; -- start bit
    for i in 0 to 7 loop -- data bits
        tx <= data(i);
        P := P xor data(i);
        wait for 8.67 us;
    end loop;
    tx <= not P; -- odd parity bit
    wait for 8.67 us;
    tx <= '1'; -- stop bit
    wait for 8.67 us;
end procedure;
```

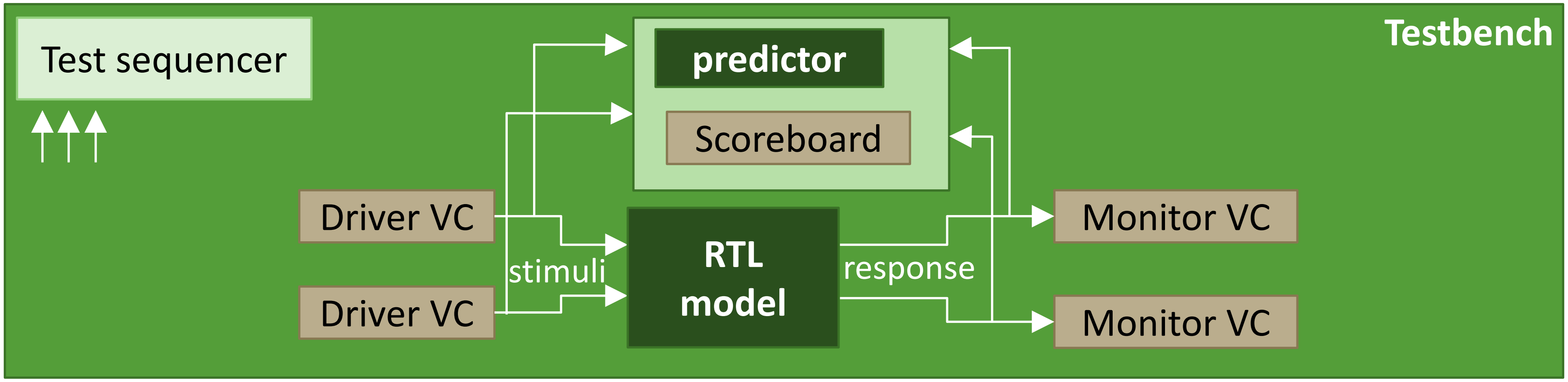
- One of the *simplest reusable component*
- Not generic
  - Other than 8 data bits?
  - Even parity?
  - Different baud rate?
- Could be improved

Reusability in verification

- There are frameworks, which helps **implementing reusable components** and **reusable verification environments**
  - **Universal/Unified Verification Methodology** (UVM)
    - SystemVerilog framework
  - **Universal VHDL Verification Methodology** (UVVM)
    - VHDL framework
- **Raise** the testbenches' **abstraction level**
  - Transaction level modeling (TLM)
  - Test sequence and the DUV does **not interact directly**
  - Test sequence contains a **list of commands**, and abstract **verification components executes** them
  - (continued on next slide)

Reusability in verification

- **Raise** the testbenches **abstraction level**
  - Transaction level modeling (TLM)
  - Test sequence and the DUV does **not interact directly**
  - Test sequence contains/generates **a list of commands**, and abstract **verification components executes** them
  - Predictor is a reference model which provides the **expected responses** to the scoreboard
  - Scoreboard stores the stimuli and responses from the DUV and the predictor



Systematic functional verification

- Using all the methods and tools mentioned earlier
  - **Systematic** -> the verification process is **documented and followed** during the design cycle
- There is a plan on how the design will be verified – **verification plan**
  - What *tools and frameworks* will be used for functional verification
  - What *verification components* will be used
  - How is the *quality of the verification measured*
  - What is the **stopping criteria** of the verification process
  - Additional information
    - How the CDC issues will be investigated

End of topic

Key concepts

- Complex verification environments
  - Self checking testbenches
  - Randomized stimuli
  - Behavior models of the DUV’s environment
- Qualifying the verification process
  - Code coverage
  - Functional coverage
- Reusability
  - Reducing the effort by re-using verification IPs

Questions

- Describe the verification process! How the progress can be measured and what is the stopping criteria?
- Why is it impossible to prove a design “correctness” with functional verification?
- Why is the functional verification expensive? How can the verification effort be reduced?
- Describe an advanced verification environment!