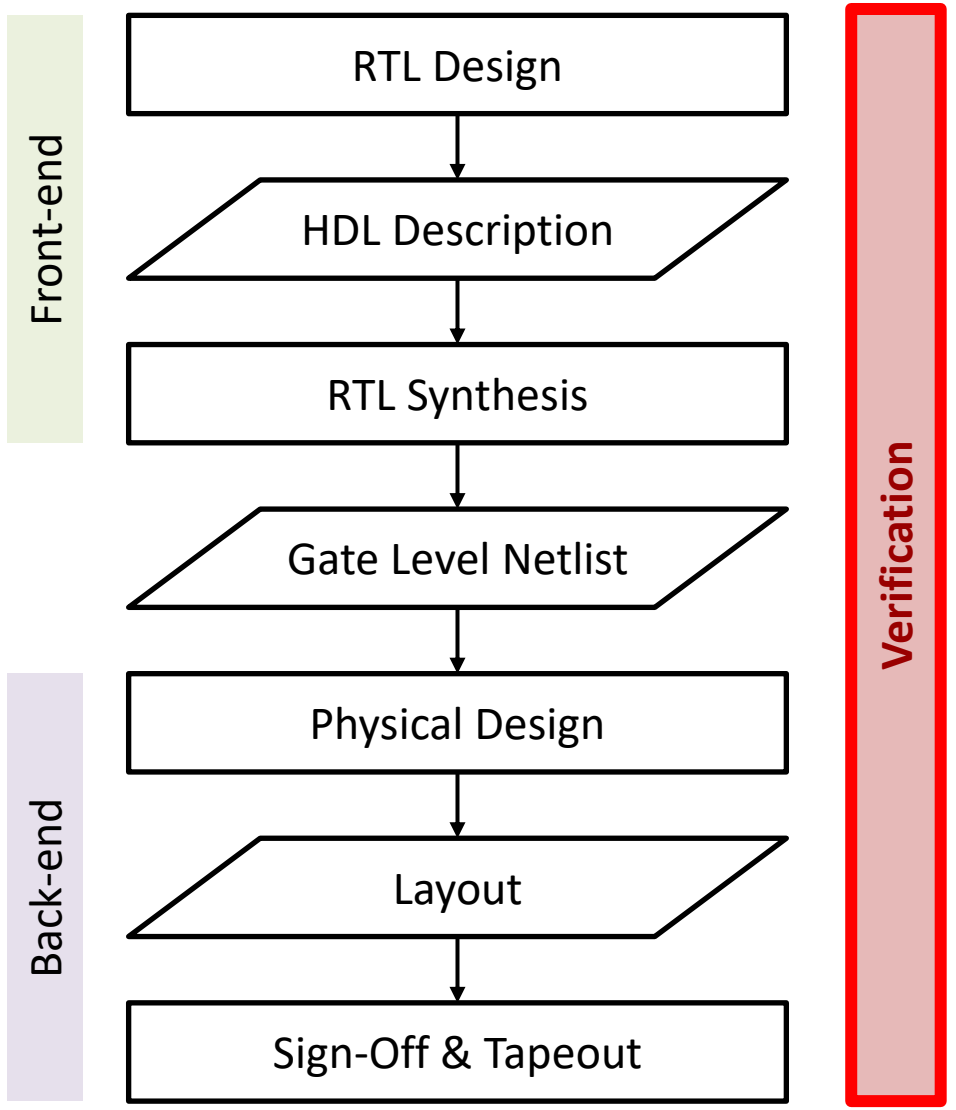
EE-429
Fundamentals of VLSI Design

The Semi-Custom Frontend: Functional Verification

Andreas Burg

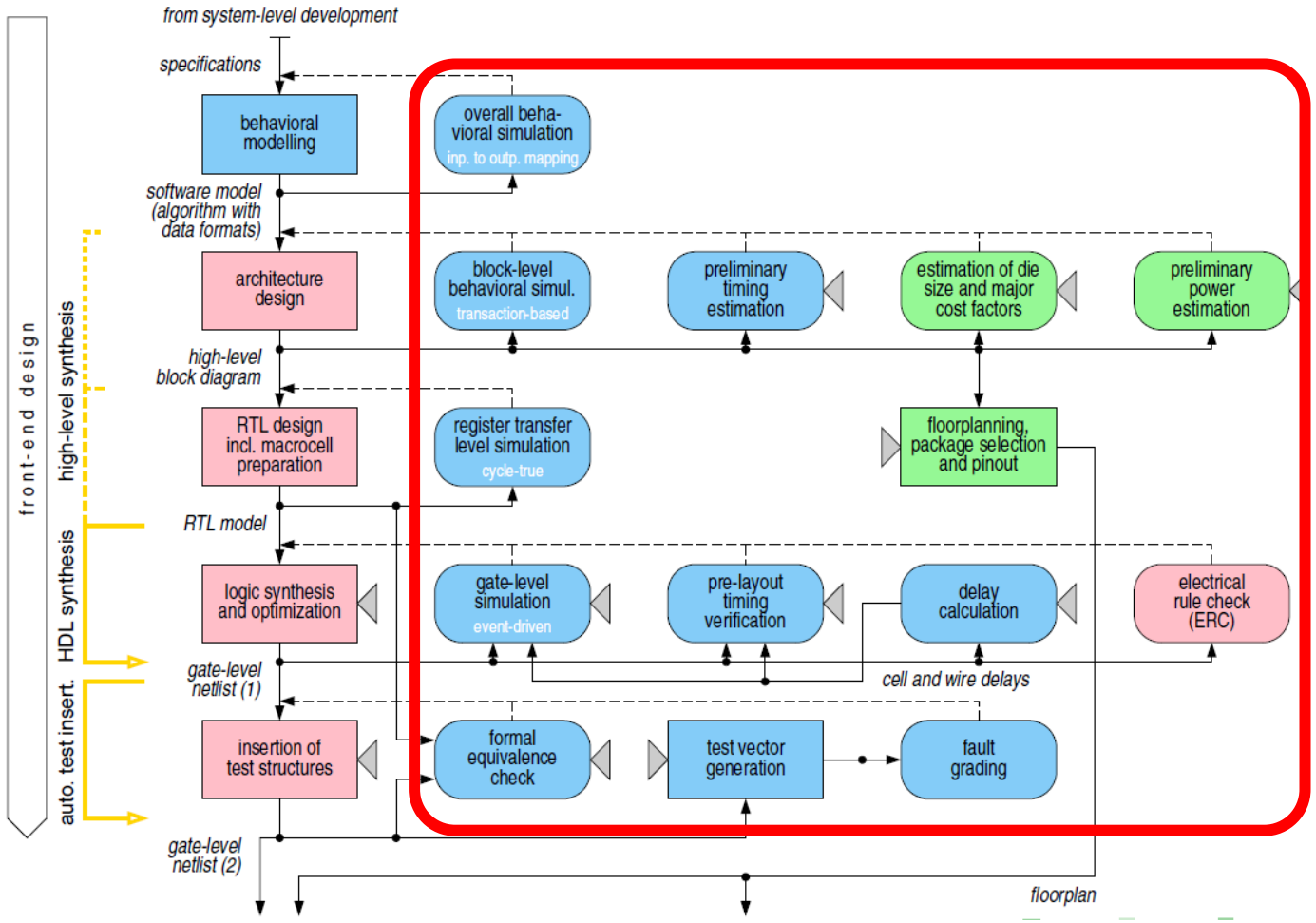
Semi-Custom (Digital) ASIC Design Flow

- **Semi-custom design flow:**
  - Starts from a Register Transfer Level description in a hardware description language (HDL)
  - **Front-end flow**: handles the transition from RTL to the gate level
  - Back-end flow: handles the transition from a netlist to physical design data
- **Each step is** always **accompanied** by **verification**
  - Check functionality, timing, and physical constraints

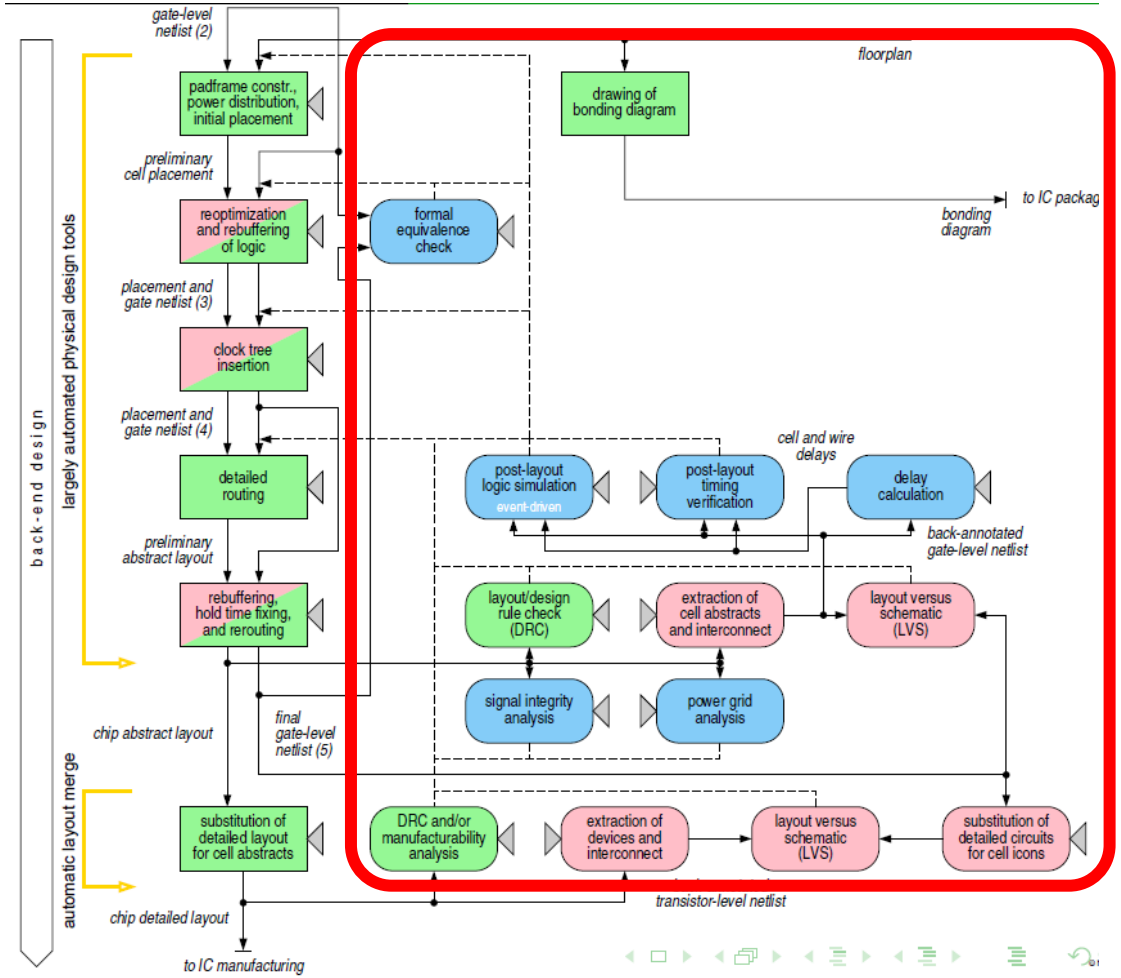


Verification Continues Throughout the Design Flow

Frontend design flow



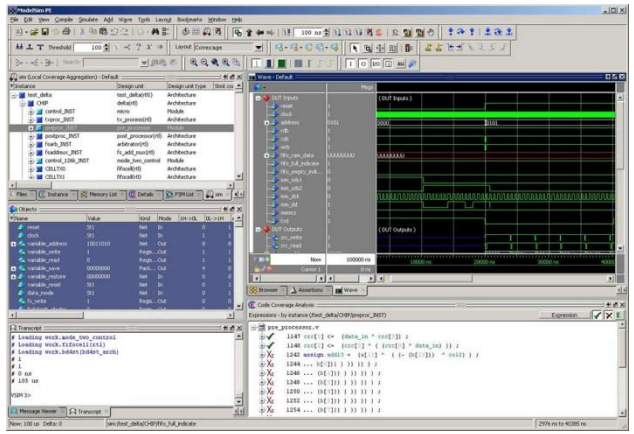
Backend design flow



Goals of Design Verification in VLSI

- **Verification can have three distinct motivations (after A. Richard Newton):**
  - During specification: “Is what I am asking for what is really needed?"
  - During design: “Have I indeed designed what I have asked for?"
  - During testing: “Can I tell intact circuits from malfunctioning ones?“
- **Verification should be done**
  - **against formal specifications** (as available): often partially possible even with formal methods and simulations
  - **against «examples»** from less formal sprecifictions: typically by means of simulations

Verification vs. (Production) Test



Verification

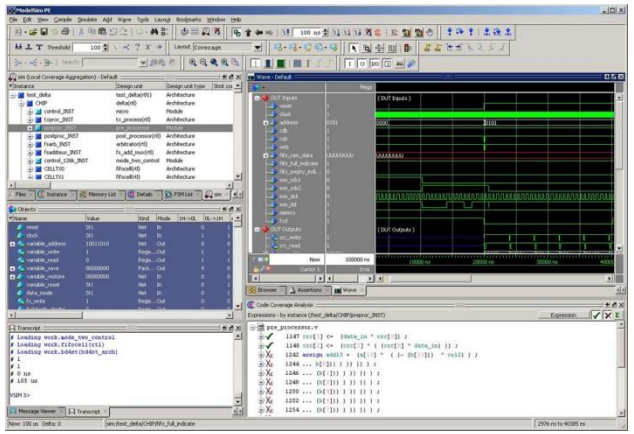
- Carried out prior to production
- Application of stimuli through a «software» testbench in a simulator
- Checks conformance with specification
  - Testpatterns designed to uncover logical bugs in the implementation

(Production) Test

- Carried out after production
- Application of stimuli through «Automatic Test Equipment» (ATE)
- Checks physical implementation integrity
  - Testpatterns designed to uncover physical production defects

Main difference between Verification and (Production) Test is the type of errors to be discovered and the design of the test patterns

Connections between Verification and Test



Verification

(Production) Test

- Both check conformity of a realization with some form of expectations
- Both apply test patterns and check conformance of the output with expected results and specifications
- (Production) Tests often reuse some testpatterns generated for functional verification as (small) part of the test sequence

Functional versus Parametric Verification

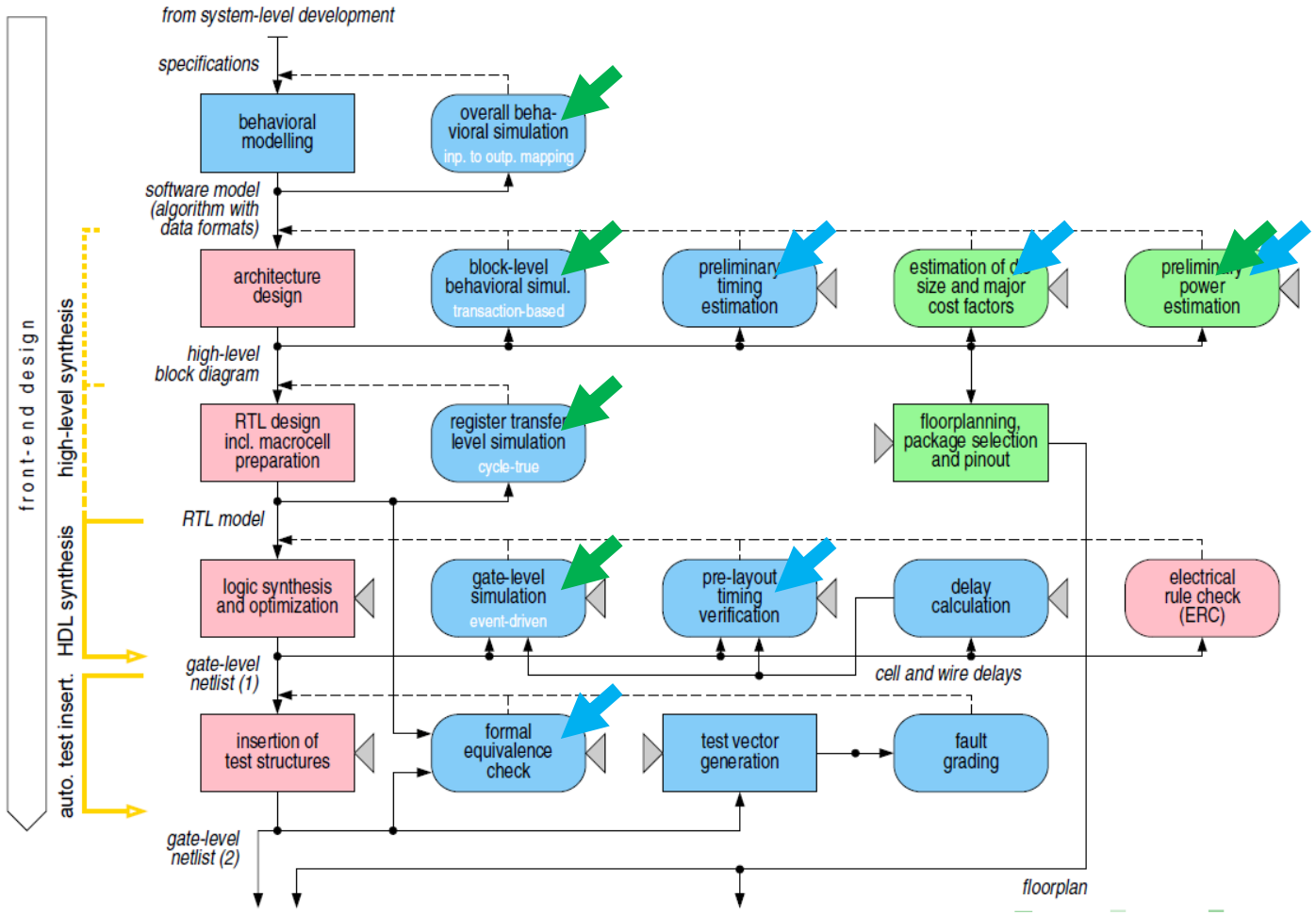
- Verification checks two different type of requirements:
  - Functionality describes what response a circuit produces at the output pins when presented with some stimuli at the input pins
    (aka logic behavior, input-to-output mapping).
    Expressed in terms of algorithms, equations, state graphs, truth tables, and the like.
  - Parametric issues relate to physical quantities measured in units such as Mbit/s, ns, V, A, mW, pF, etc.
- **A design's functionality and its parametric properties are best checked separately since goals, methods, and tools are quite different.**

Dynamic vs Static Design Verfication

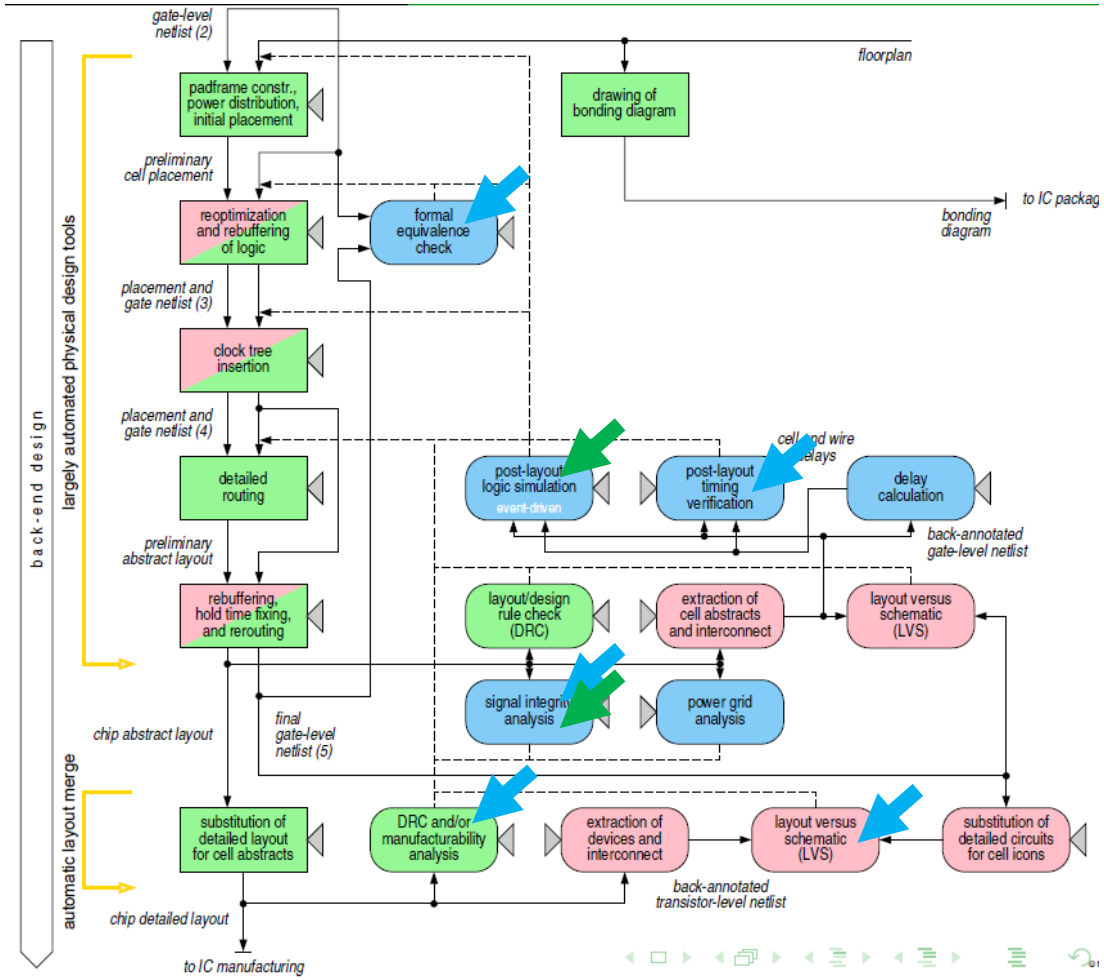
- **Dynamic verification** techniques
  - work by applying stimuli and by observing responses
  - involve time (either as physical time or as simulation time)
    Examples:
    Simulation, circuit testing.
- **Static verification** techniques
  - do not depend on signals, waveforms or test patterns in any way
  - involve no notion of time, they operate at “time zero“
    Examples:
    Code inspection, equivalence checking,
    formal verification and property checking (well known in SW development),
    electrical rule check (ERC), static timing analysis;
    layout rule check (DRC), layout versus schematic (LVS).

Dynamic vs Static Design Verfication

Frontend design flow

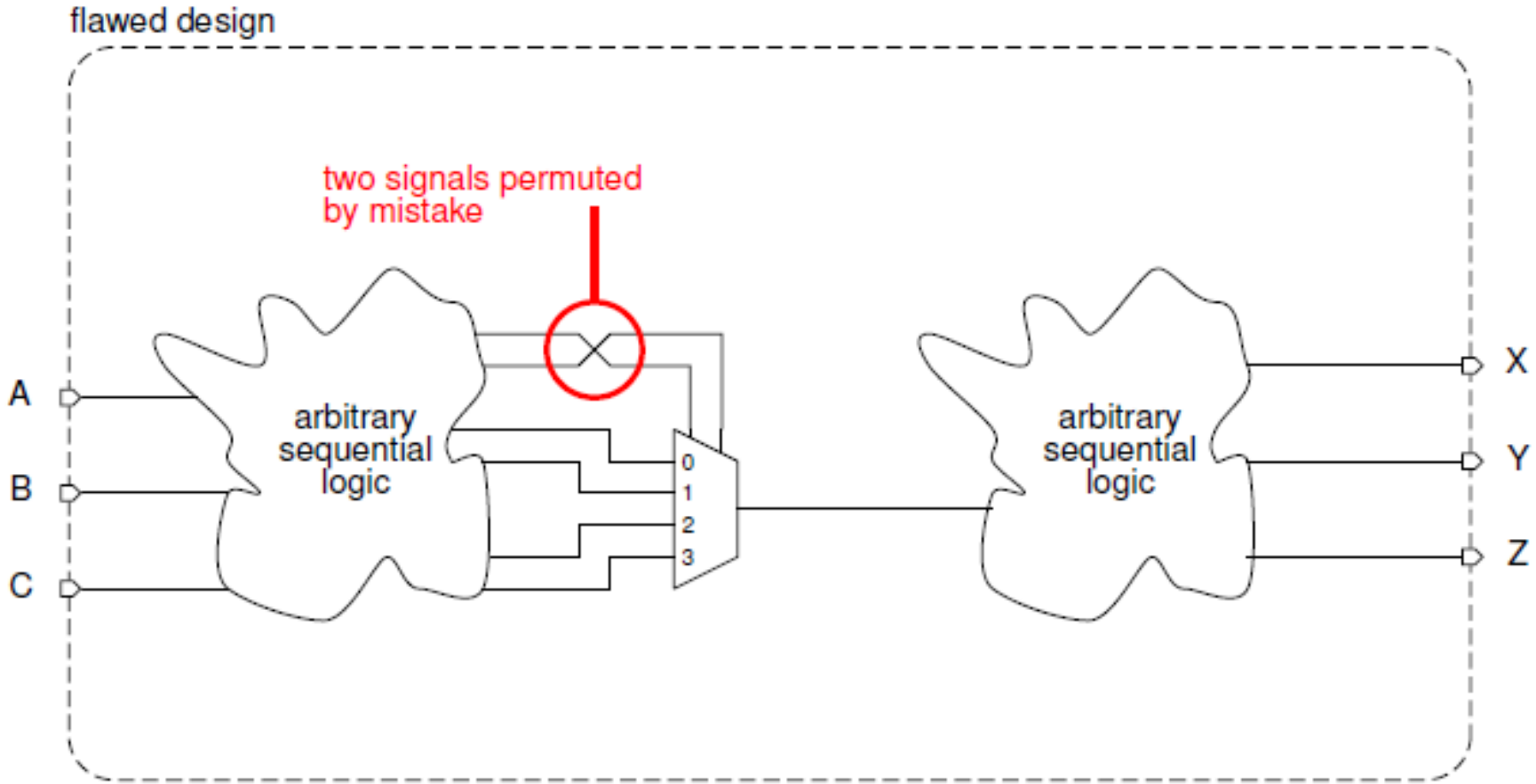


Backend design flow



How to Uncover a Bug with Simulation or Testing?

- **Example:**

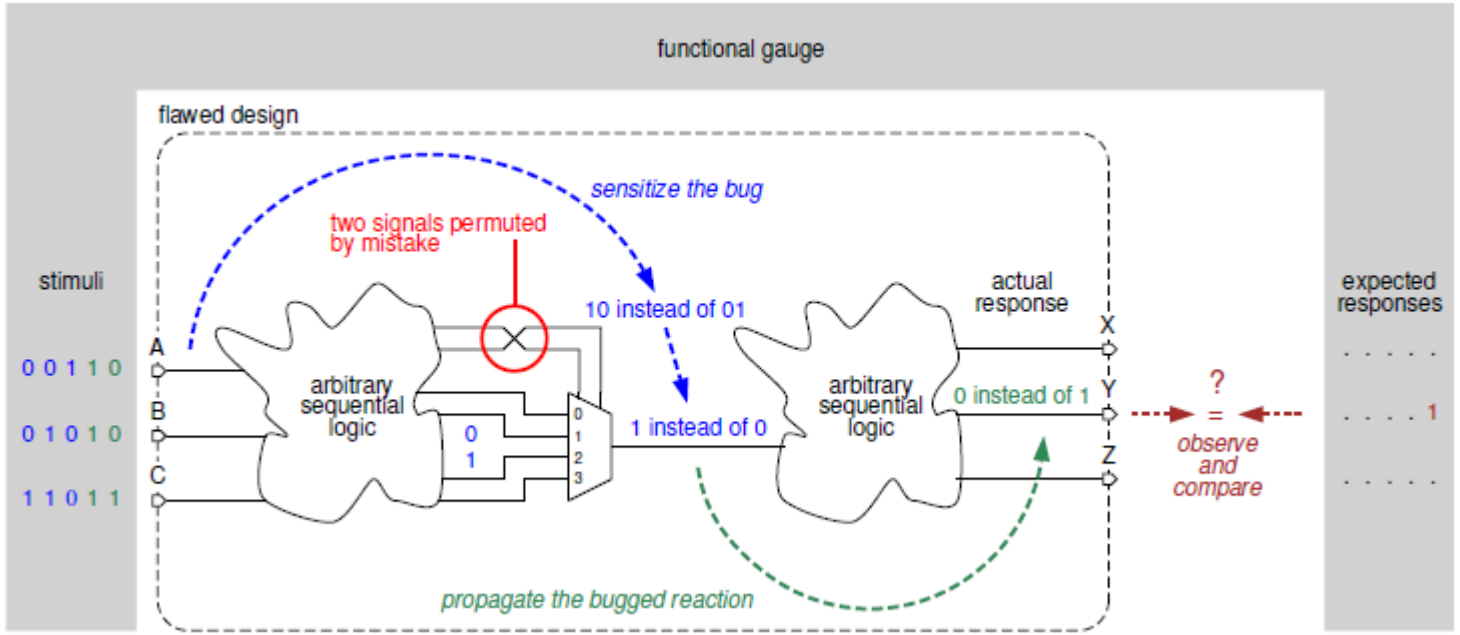


Two bits exchanged on select input of a 4->1 MUX

- Used «1 to 0» instead of «0 downto 1» in VHDL

How to Uncover a Bug with Simulation or Testing?

- **Preconditions necessary for uncovering the mistake:**
  1. Make the design error provoke a condition other than the normal one.
  2. Propagate the erroneous condition to observable nodes.
  3. Check the observed values against expectations for a correct design.

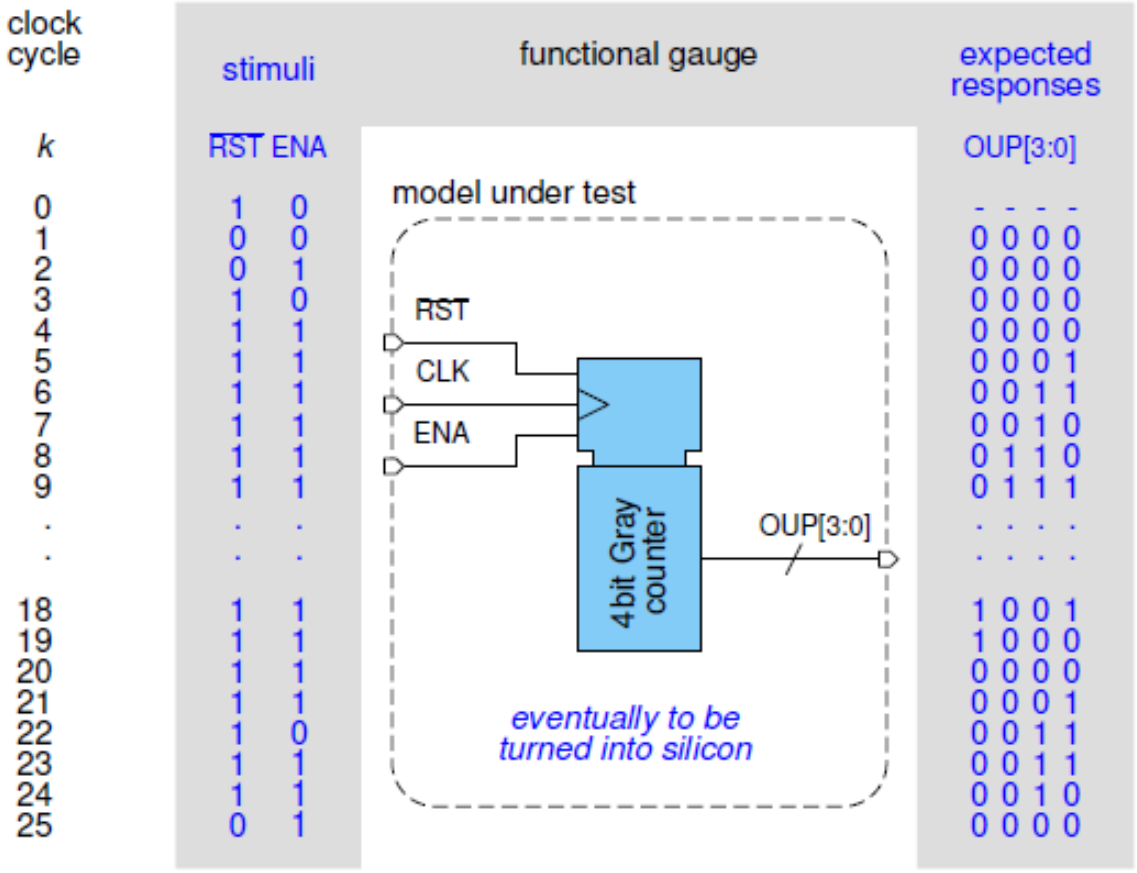


Automated Stimulation and Response Checking

- Each simulation run generates waveforms, tabular printouts, event lists, …
- **For reasons of efficiency and quality, purely visual inspection of simulation data is not acceptable in VLSI design.**
- Hence, designers must arrange for the simulator to automatically
  - apply stimuli to the model under test (MUT)
  - acquire the actual responses from the MUT
  - compare them against the expected responses
  - report any differences in a meaningful way
- We refer to
  - the set of stimuli and of expected responses as a functional gauge.
  - The means to apply the stimuli, check responses, and provide a proper environment for operation as the testbench

Functional gauges are specs that have materialized

- **Example: 4 bit Gray counter with enable and asynchronous reset**



Objective: cover all possible cases

Testing Distinct Functional Mechanisms Separately

Consider a 4 bit Gray counter example.

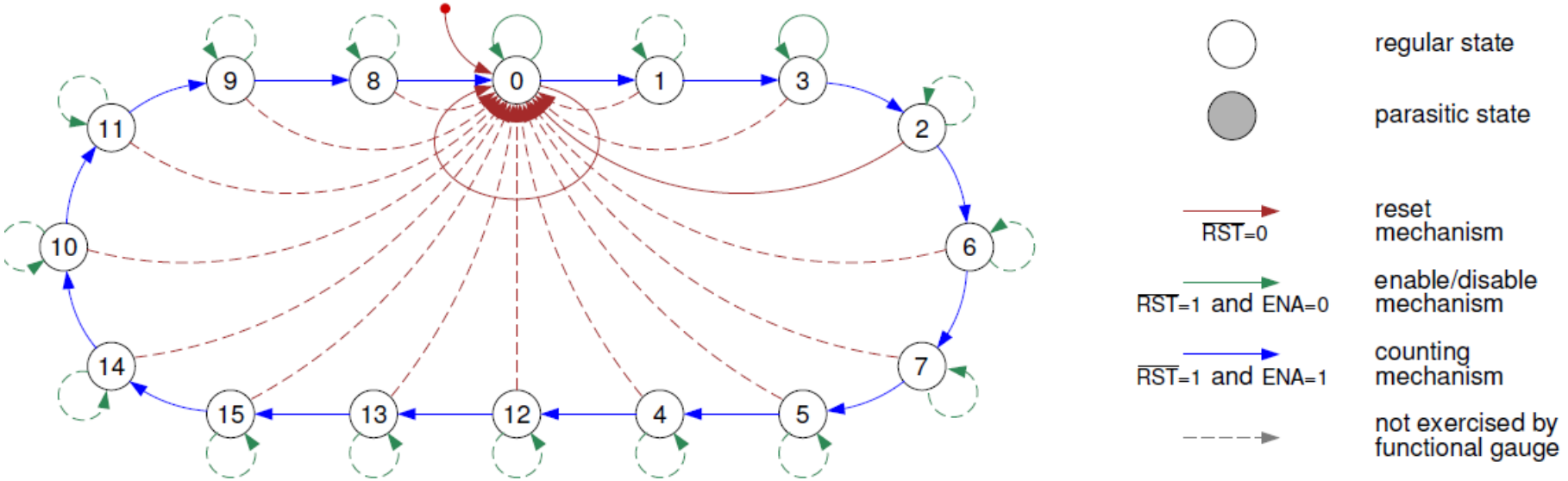
$\overline{\texttt{RST}}$	CLK	ENA	OUP
0	-	-	00...0
1	$\downarrow$	-	OUP
1	$\uparrow$	0	OUP
1	$\uparrow$	1	$\text{graycode}((\text{bincode}(\texttt{OUP}) + 1) \bmod 2^w)$

- $2^{w_i+w_s} = 2^{2+4} = 64$ cycles is a lower bound for exhaustive verification
- Truth table suggests the existence of **three distinct mechanisms**:
  - Asynchronous reset mechanism
  - Enable/disable mechanism
  - Gray-coded counting mechanism

Testing Distinct Functional Mechanisms Separately

Reconsider the 4 bit Gray counter example.

- State graph of a 4 bit Gray counter with edges colored according to the functional mechanism they implement.



- **Selection of few state transitions for verification** from “similar” groups **reduces the number of test patterns** significantly**, but leaves also many transitions untested**

Testing Distinct Functional Mechanisms Separately

- Though a workable solution in the Gray counter example, partial verification suffers from four limitations:
  - We have refrained from traversing all edges. Yet, the problem of combinatorial explosion persists as visiting all states rapidly becomes impractical on more substantial sequential circuits.
  - Identifying mechanisms and subcircuits for verification requires partial insight into its inner organization and working.
  - Verifying each mechanism and subcircuit separately holds the risk of missing those problems that relate to the interaction of two (or more) of them.
  - There exists many buggy implementations where identical functional specifications of the original truth table behave differently: Functionally identical mechanisms may be based on different pieces of code that may differ due to bugs (see next slide)

Testing Distinct Functional Mechanisms Separately

- **Writing HDL code that is safe when testing distinct functional mechanisms separately?**
  - Expressing multiple instances of the same condition in separate statements is prone to errors (statements that should be the same may accidentally differ = bug)

```
p_comb : process (STATExDP, ENxS)
begin  -- process p_comb
  case STATExDP is
    when 0 =>
      if ENxS = '1' then
        STATExDN <= 1;
      else
        STATExDN <= STATExDP;
      end if;

    when 1 =>
      if ENxS = '0' then
        STATExDN <= 3;
      else
        STATExDN <= STATExDP;
      end if;

    when 3 =>
      if ENxS = '1' then
        STATExDN <= 2;
      else
```

Bug

```
p_comb : process (STATExDP, ENxS)
begin  -- process p_comb
  if ENxS = '0' then
    STATExDN <= STATExDP;
  else
    case STATExDP is
      when 0 =>
        STATExDN <= 1;

      when 1 =>
        STATExDN <= 3;

      when 3 =>
        STATExDN <= 2;

      when 2 =>
        STATExDN <= 6;

      when 6 =>
        STATExDN <= 7;
```

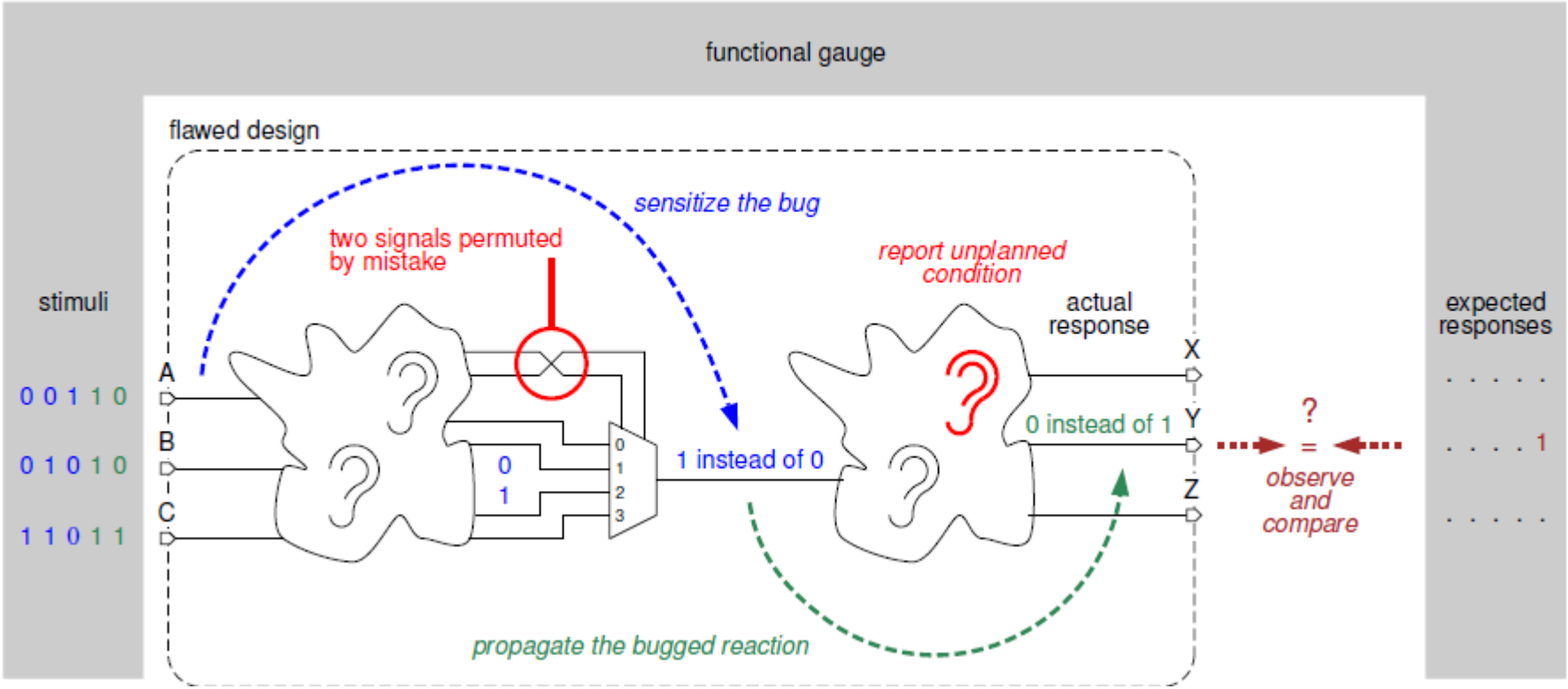
Use Stimuli Collected from the Real-World

- **Collect stimuli from the target environment.**
  - Often difficult when DUT interacts with components that generate the stimuli
- **Integration of prototypes into the target environment.**
  - Limited by limited speed of prototypes (no real-time operation)

- **Problem:**
  - Some bugs are limited to very rare events
  - No expected result available: manifestation of a bug may be subtle
  - Example: Pentium division bug (1994)
    By mistake, 5 out of 1066 table entries had been omitted from a lookup-table.
    Fraction of the total input number space prone to fail estimated to be $\mathbf{10^{-10}}$
    Several months to become aware of the problem

Assertion-Based Verification

- **Many errors are not spotted due to limited visibility on primary outputs**



Preconditions for uncovering the mistake are often complex

Assertion-Based Verification

- **Assertions are "in-code sanity checks" facilitate providing visibility**
  - Assertions can further be used to ascertain that the result of a calculation is correct or at least plausible and inoffensive.

- Three approaches:
  - Cross check data produced by the DUT or a sub-circuit against others computed by a different piece of code that is made part of the verification code.
    - Code used for checking is not subject to synthesis and can therefore be behavioral and high level. Hence, it is less prone to errors and can even use verified (but not synthesizable) libraries and functions
  - Check for logical conditions that can easily be derived from specifications
  - Checking by way of reverse computation.
    Example: Computing the square root is fairly complex and thus prone to error while checking the result is straightforward.

Assertion-Based Verification

- **Assertions** are statements embedded within a MUT that do not affect functionality.
- Instead, they report anomalous or unexpected conditions such as
  - Memory addresses that point outside their legal range,
  - FSMs that assume parasitic, illegal or otherwise suspect states,
  - Unforeseen input symbols and other out-of-the-ordinary conditions,
  - Illegal instruction codes (opcodes) and unexpected status codes,
  - Numeric over/underflows, out-of-range values, and other scaling problems,
  - Event sequences unforeseen by the application or protocol,
  - Resource conflicts and other situations of mutual lock-up,
  - Excessive iteration counts or other unexpected variable values.
  - ...

Assertion-Based Verification

- **Example: FIFO queues,** popular as temporary buffers in data processing chains.

- **Typical design flaws**
  - Overrun (overwriting locations prior to readout) => loss of data
  - Underrun (reading where no valid data have been stored before) => bogus output
  - Imperfect FSM code => hangup or other unpredictable malfunction

- **Imagine a faulty FIFO in a substantial data processing chain:**
  - Difficult to locate the culprit.
  - Many clock cycles before erroneous FIFO data affects output pins.

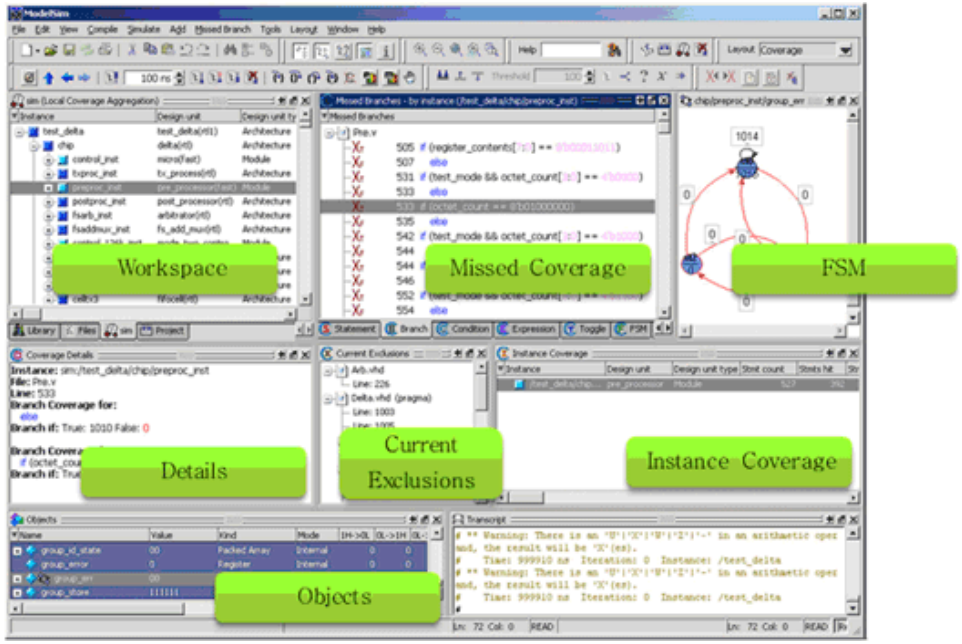
- **Problem fix: A few well framed assertions involving the address pointers**
  - can help monitor the FIFO's fill level for comparison against expectations.
  - will report any anomalies in FIFO operation as soon as they occur,

Benefits of Assertion-Based Verification

- In-code sanity checks nicely complement response checking:
  - Immediate feedback. No need for an abnormal condition to propagate to some distant node placed under constant monitoring.
  - Short link from problem manifestation to cause. No need to trace back a mismatching output over thousands of cycles and statements in an attempt to locate its place of origin.
  - Lasting investment. No need to repeatedly adjust assertions when submoduls are being assembled to form larger design entities.
- Enter an assertion into your HDL code wherever you explicitly or implicitly assume that a certain property should hold in real life service.
- Assertions can reduce the number of unit tests (separate testbenches on small sub-units/components)

Criteria for Quality / Coverage of the Verification

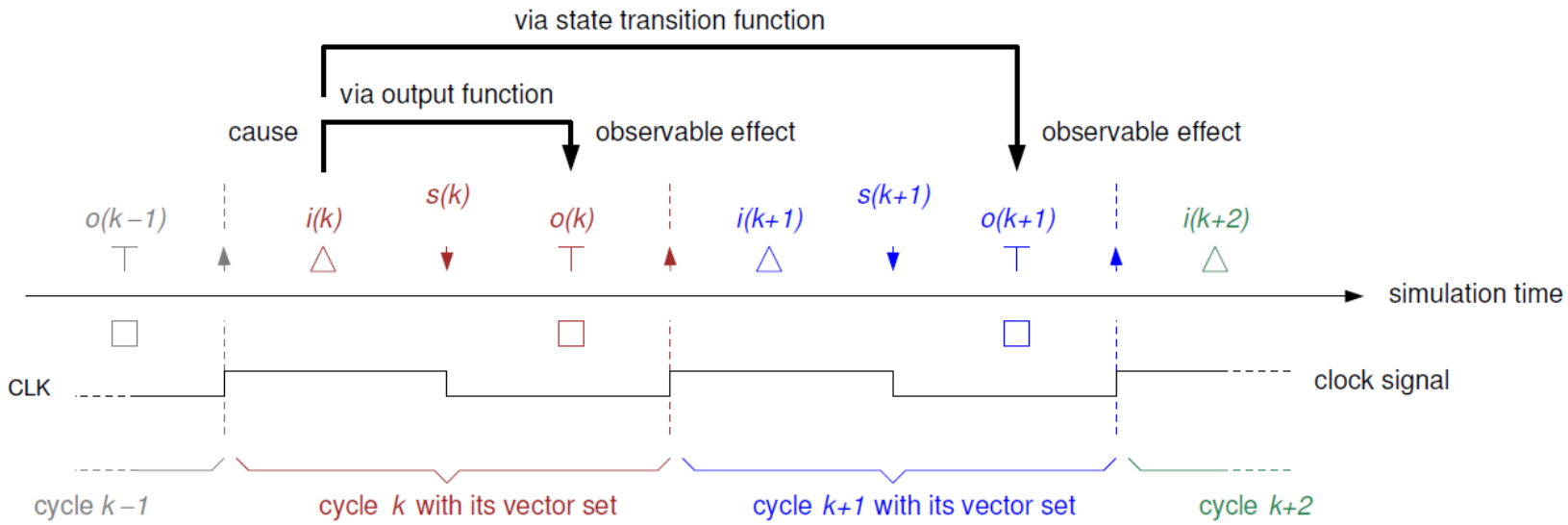
- **State coverage**
  - Number of states explodes with number of registers
  - FSM coverage:
    - Identifies FSM (as opposed to dataflow registers)
    - Considers individual, but coupled FSMs independently



- **Code coverage**
  - Checks for excitation of all parts of the HDL code
  - Different granularities: statement, expression, condition
  - Ignores complex dependencies
- **Toggle activity**
  - Ensures that all signals toggle at least a certain number of times
  - Ignores correlation between signals

Alternative Ways to Build a Testbench

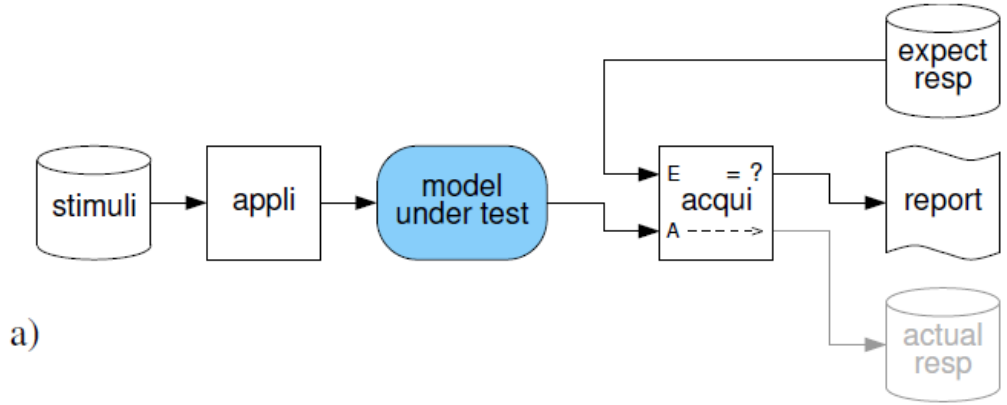
- Cycle accurate stimuli application
  - Stimuli and expected responses are prepared before simulation begins and are applied and checked on a cycle-by-cycle basis



  - Checks not only for functionality, but also for precise expected latency
  - Stimuli generation is complex since all latencies must be known

Alternative Ways to Build a Testbench

- **Two** alternative **approaches** for Cycle accurate stimuli application
  - **Stimuli and expected responses** are included **in the source code of the testbench** itself:
    - VHDL testbench often eases the cycle accurate stimuli generation
    - not very flexible and only for small and simple blocks
  - **Stimuli and expected responses** are prepared outside the testbench and **stored in files**: «Application» and «reponse acquisition» processes read files, apply stimuli, check responses and record response traces in files
    - Generation of cycle accurate stimuli and responses is often difficult from abstract specifications and high level models that serve as a golden model



a)

Often too limited and too rigid to be of any practical value.

Alternative Ways to Build a Testbench

- Example: JPEG image compression
  1. Accept an image frame,
  2. subdivide into square blocks,
  3. do 2D DCT on each bock to calculate a set of spectral coefficients,
  4. suppress all coefficients with minor impact on image quality.
- Relevant in the context of functional verification:
  - Large data items such as image frames, blocks, and coefficient sets.
  - Larger operations such as the compression of one block or frame.
- Low-level details such as the reading in of pixels or handshake procedures would just distract your attention.
- From an application point of view, JPEG compression is a combinational function and correctness is defined only by the result.
  However, typical image compression hardware spreads the operation over many clock cycles and involves interactions with many other components that are tricky to model when preparing the relevant expected response.

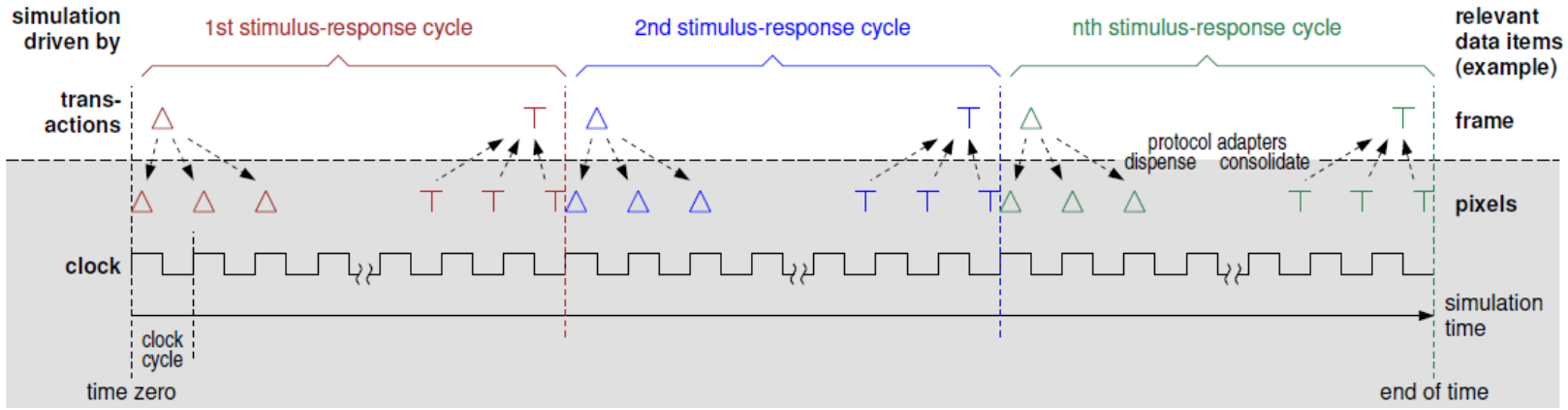
Alternative Ways to Build a Testbench

- **The duties of a testbench are**
  - translate stimuli and responses across levels of abstraction
  - consolidate simulation results such as to render interpretation by humans as convenient as possible.

- The difficulty is that
  - hardware contains many (timing) aspects that are not easy to model on a high abstraction level where functional responses are typically generated (e.g., MATLAB, C, C++)
  - Latency and other timing parameters of a circuit may change in the design cycle
  - Hardware interacts with other components that are also not part of functional/behavioral/algorithmic stimuli generation

Alternative ways to build a testbench

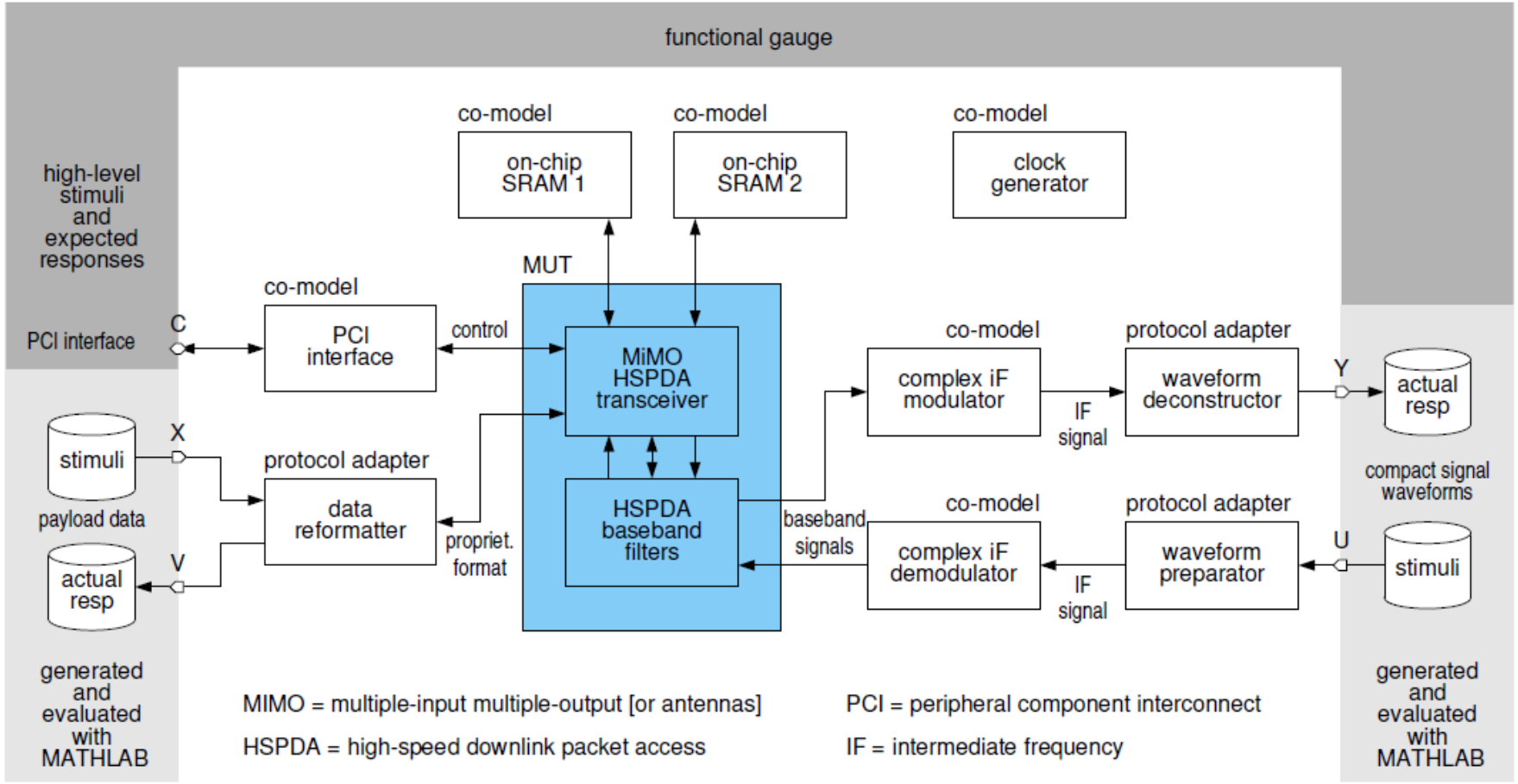
- **Abstracting to higher-level transactions on higher-level data**
  - Abstracting timing issues is best delegated to protocol adapters.



  - Emulation of interfaces to peripheral components (e.g., external SRAMs) is best dediated to functional models of other system components

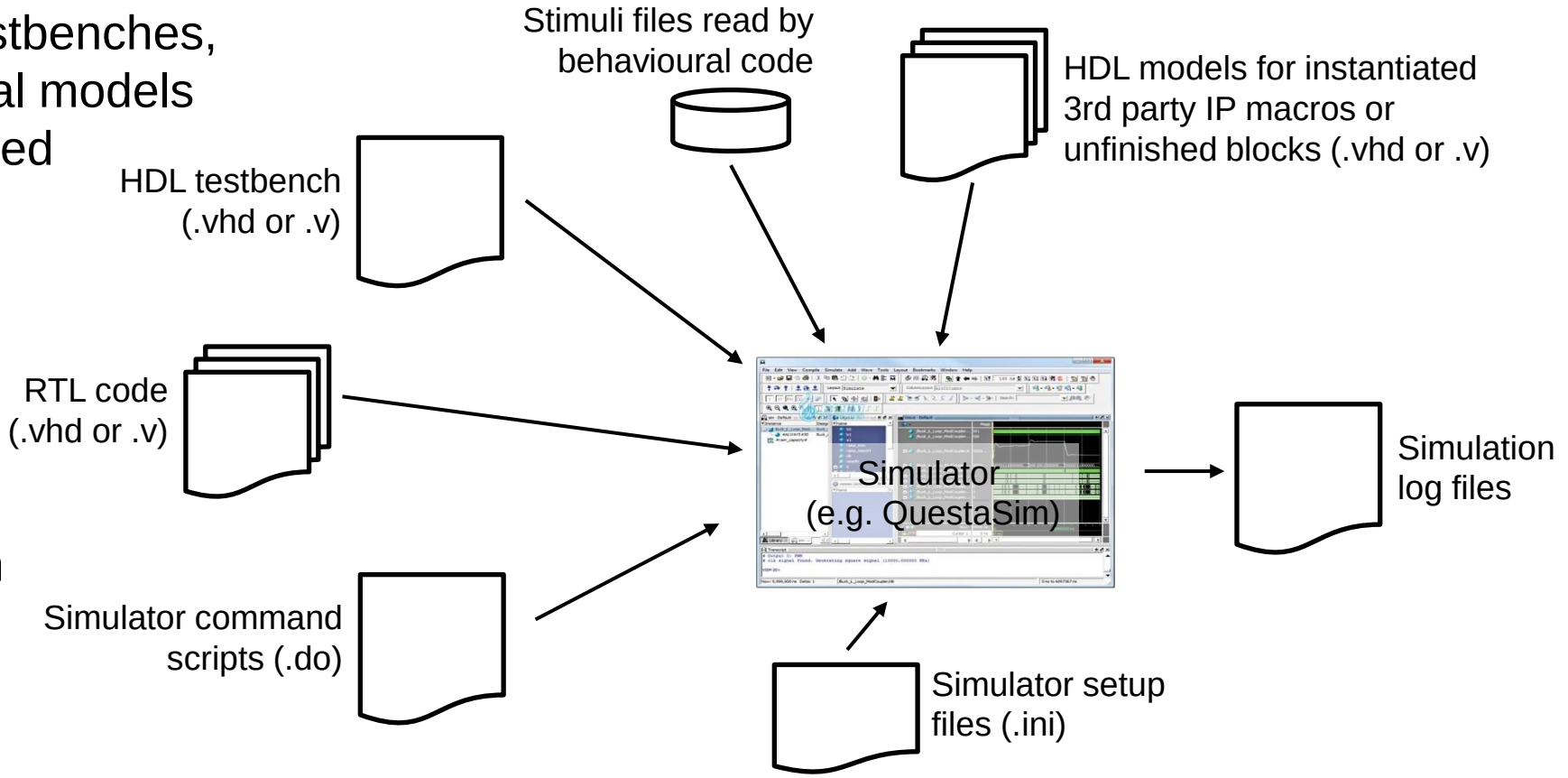
Alternative ways to build a testbench

- **Example: Simulation set-up for a wireless transceiver**



RTL Simulation in the Design Flow

- **RTL simulations are performed with an HDL simulator**
  - Event driven simulation of the HDL code
- **Various inputs are required for HDL simulation of RTL code**
  - HDL models including testbenches, your RTL code, behevioral models for IP blocks and unfinished RTL blocks
  - Data files with stimuli or expected responses that are read by a testbench or by a behavioural model of an IP macro or a model in a testbench component
  - Simulator setup (.ini) and command scripts (.do)



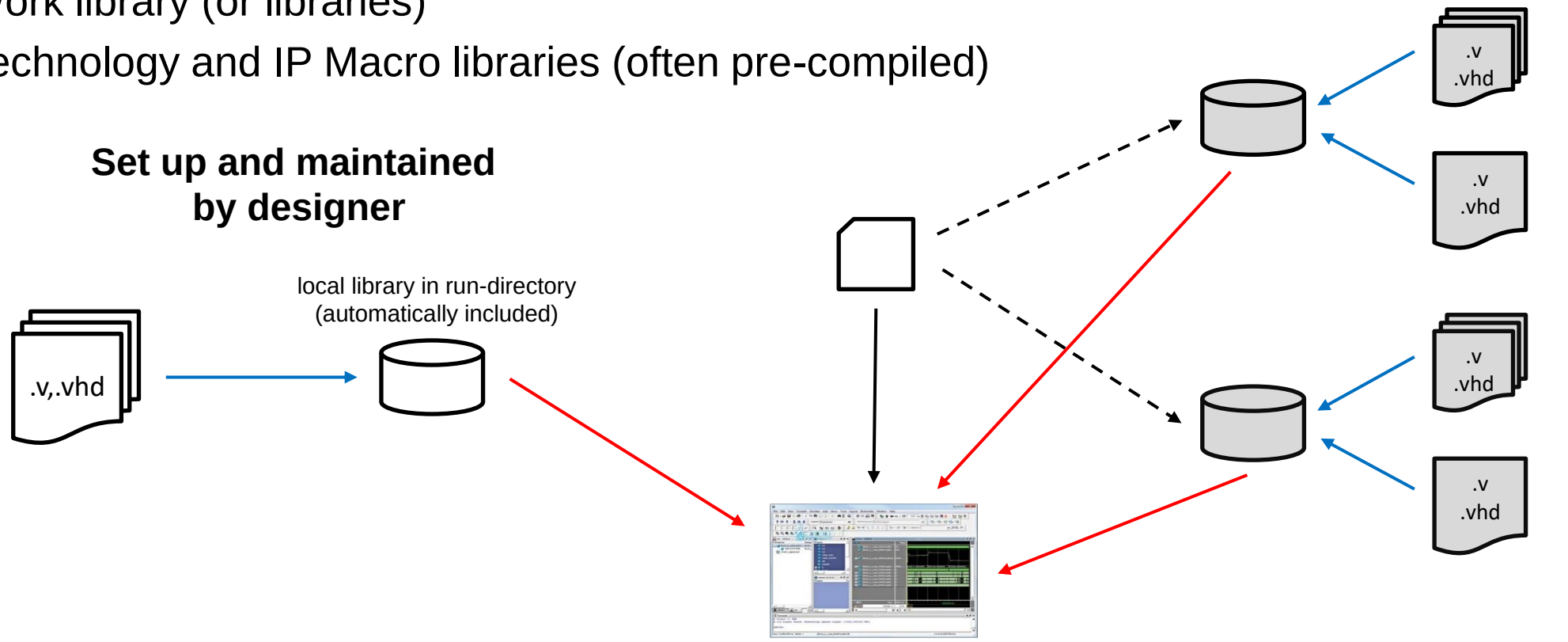
The RTL Simulation Environment

- **RTL simulation runs in a dedicated sub-directory of your project**
- **In this directory, you find typically**
  - Setup/configuration files for the simulator (e.g., modelsim.ini)
  - Sub-directories for scripts
  - Libraries (often as directories) containing the compiled HDL code
  - Links to the source file location (e.g., to your HDL directory)

```
QUESTASIM/
├── ACTIVITY                      - - - folder for VCD activity files
├── edadk.conf -> ../edadk.conf   - - - configuration file used in EPFL for EDA tools
├── IPS -> ../IPS/                - - - link to IPS folder
├── LIBS                          - - - folder for compiled stdcells
├── HDL -> ../HDL/                - - - link to VHDL/Verilog source files
└── modelsim.ini                  - - - configuration file for questasim
```

The Simulation Environment & Libraries

- **HDL simulators work with libraries**
  - Work library (or libraries)
  - Technology and IP Macro libraries (often pre-compiled)



Setting up Libraries with QuestaSim (Modelsim)

- **Start by setting up a work environment/directory** (often given by EDA team)

```
QUESTASIM/
├── ACTIVITY                        - - - folder for VCD activity files
├── edadk.conf -> ../edadk.conf     - - - configuration file used in EPFL for EDA tools
├── IPS -> ../IPS/                  - - - link to IPS folder
├── LIBS                            - - - folder for compiled stdcells
├── HDL -> ../HDL/                  - - - link to VHDL/Verilog source files
└── modelsim.ini                    - - - configuration file for questasim
```

- Set up reference libraries (for standard cells and IP macros)
  - Often created by EDA team or project management
  - Point to those libraries in the simulator init file (e.g., modelsim.ini)
  - Defines the relationship between <library name> and <library folder>

Modelsim.ini from EDA-Labs showing IEEE libraries and UMC 65nm std.cell libs

```
[Library]
std = $MODEL_TECH/../std
ieee = $MODEL_TECH/../ieee
vital2000 = $MODEL_TECH/../vital2000

;uk65lscllmvbbr_sdf21 = ./DLIB/uk65lscllmvbbr_sfd21_vlog_local/
uk65lscllmvbbr_sdf21 = ./DLIB/umc_65nm_ll_uk65lscllmvbbr_sdf21_vlog
;uk65lscllmvbbr_sdf21 = ./umc_lib
```

Side note: many other libraries are often defined here as well

Compiling Work Libraries with QuestaSim

- **Work libraries are created in the run-directory of the simulator**
  - Often, there is only one work library, called “work”
- Creating a work library

```
> vlib <library name>
```

  - This creates a directory (storage place of the work library) in the simulator run-directory
- Libraries in the simulator run directory are automatically read by the simulator
  - No need to include or define these in the modesim.ini (only needed if directory name should be different from the library name)
- Compiling your HDL code into the work library
  - Compiling VHDL

```
> vcom -work <library name> <VHDL file>
```

  - Compiling Verilog

```
> vlog -work <library name> <Verilog file>
```

- HDL files need to be compiled/recompiled bottom-up after every change
- Best practice: create a script to compile your code