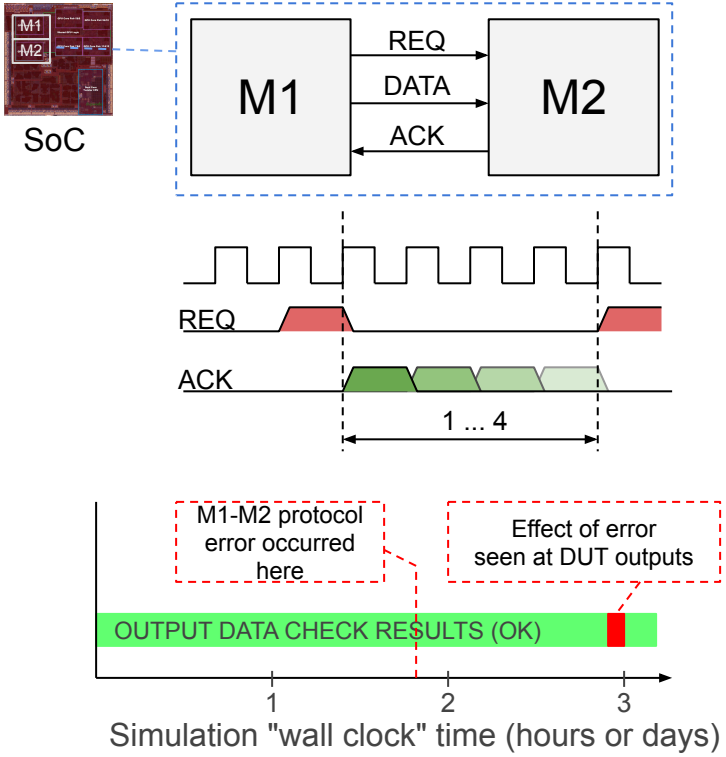
Digital Techniques 2

L10: Verification Techniques

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

Assertion-Based Verification: The Problem

- Assume M1 and M2 are modules in a large design that communicate using request-acknowledge protocol: M2 must respond with ACK in 1-4 cycles after M1 has written out DATA and set REQ
- A system-on-chip design can contain a large number of such interfaces
- It is difficult to observe protocol failures in intra-design interfaces from simulation results
- Wrong data will eventually propagate to DUT outputs, but it difficult to find out the exact location and time of the error event
- You need watchdogs everywhere!



The Solution: SystemVerilog Concurrent Assertions ("SVAs")

- SVAs are used to declare design properties such as:
  "If REQ rises, then ACK must be set to '1 in 1 - 4 clock cycles"

- SystemVerilog **property** statements can be used to express this formally

```
property P1;
  @(posedge CLK disable iff (rst_n == '0)
     $rose(REQ) |=> (ACK == '0) [*0:3] ##1 (ACK == '1);
endproperty
```

- An **assert** statement checks the **property** on every clock edge **in simulation** if reset is not on, and it reports the time and location of an error immediately in simulation

```
assert property (P1)
   else $display("%d: ACK not seen within 1 to 4 cycles.", $time);
```

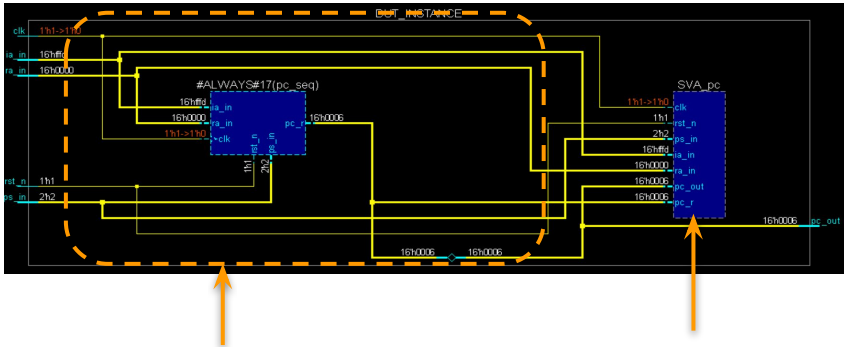
- Properties can also be **proved without simulation** using a **formal verification** tool

Concurrent Assertion Usage in Practice

- Concurrent assertions can be placed in many places, but usually a separate "**assertion module**" is created for every design module
- Concurrent assertions of a design module are declared in its assertion module
- The assertion module ("des_sva") is instantiated inside the design module "des" using a **bind** statement (SVA is the instance name):
  `bind des des_sva SVA (.*);`
- This way the code of the design module does not have to be changed and assertions can be easily removed from simulation when they are not needed
- All ports and variables of the design module must be declared as inputs in the assertion module

Example: Binding of module pc_svamod to module pc as instance SVA_pc:

```
bind pc pc_svamod SVA_pc (.*);
```



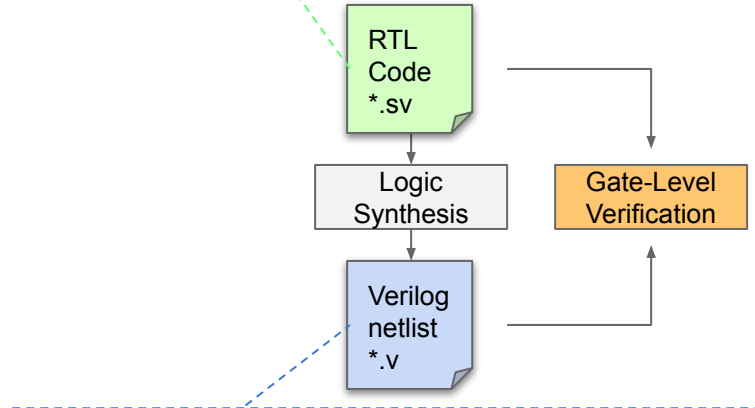
Design module's RTL code

Assertion module instance (sees all signals)

Gate-Level Verification

- Gate-level verification is done after logic synthesis to check that the synthesized model (Verilog netlist) is logically equivalent with RTL code
- In a synthesis-based flow, simulation-synthesis mismatches caused by bad RTL coding are a common source of RTL-vs-gates differences
- Verification methods:
  - **Formal logic equivalence check** (LEC)
  - **Gate-level simulation**
- LEC is the prevalent method for GL verification, but simulation has its uses, too

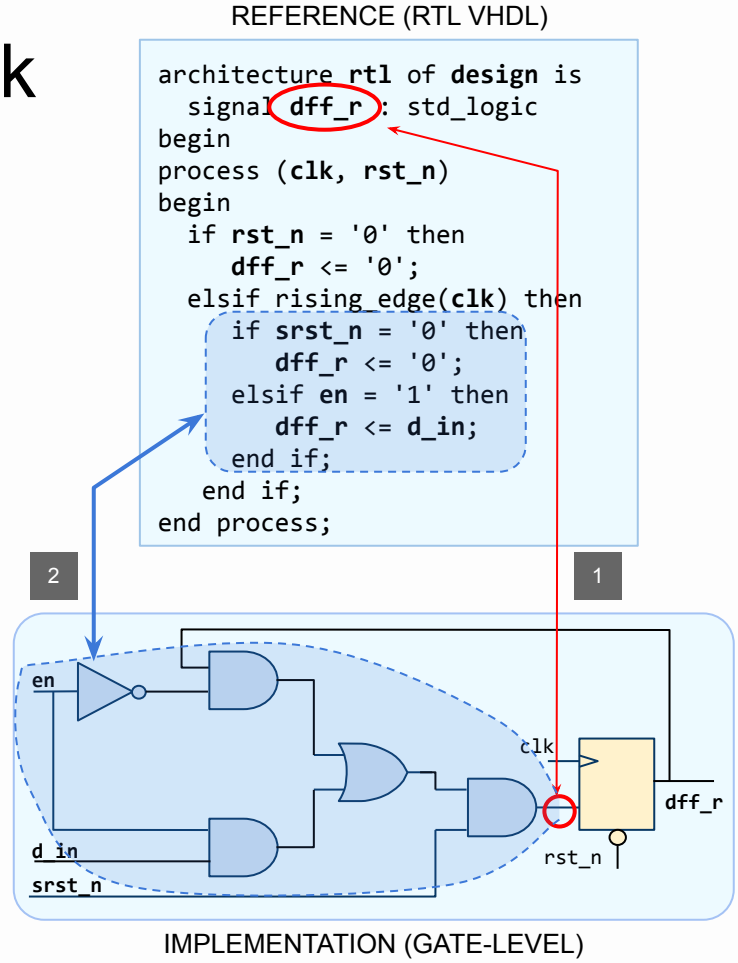
```
case (mode_in)
  2'b00: ctr_r <= ctr_r;
  2'b01: ctr_r <= BITS'(ctr_r + 1);
  2'b10: ctr_r <= BITS'(ctr_r - 1);
  2'b11: ctr_r <= data_in;
```




```
DFFARX1_HVT ctr_r_reg_0_ ( .D(n17), .CLK(clk), .RSTB(rst_n),
.Q(data_out[0]), .QN(n43) );
AO21X1_HVT U42 ( .A1(n35), .A2(mode_in[0]), .A3(n34), .Y(n17) );
MUX21X1_HVT U41 ( .A1(data_in[0]), .A2(n43), .S0(n31), .Y(n35) );
NAND2X0_HVT U32 ( .A1(n22), .A2(n21), .Y(n34) );
```

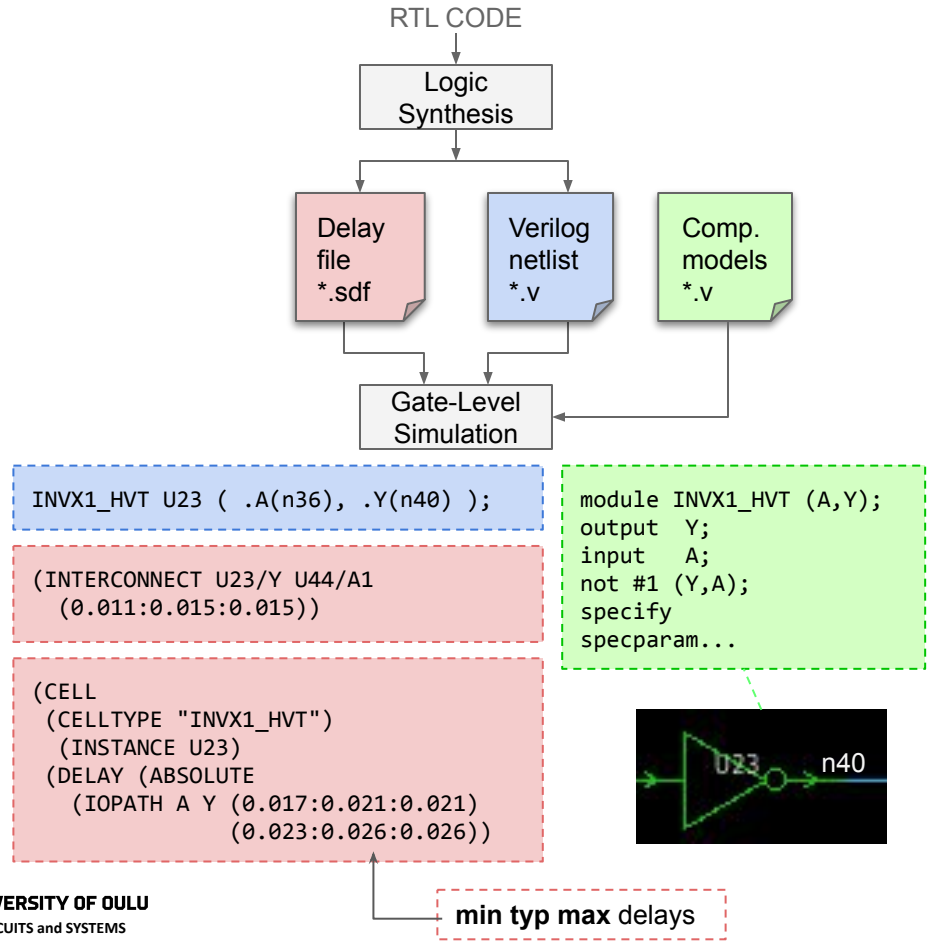
Formal Logic Equivalence Check

- LEC checks that Boolean logic functions of **compare points** are equal in the reference design (RTL) and its implementation (GL design)
- Compare points are flip-flop data inputs and design output ports
- LEC procedure:
  - **Match** all design outputs and flip-flop inputs with the respective variables/signals in the RTL code [1]
  - **Verify** that the logic cones that drive these compare points are functionally equivalent [2]
- Advantages over simulation: Not dependent on test stimulus coverage
- May require some setup effort if changes are made to the design in synthesis by adding test logic etc. (more in DT3)



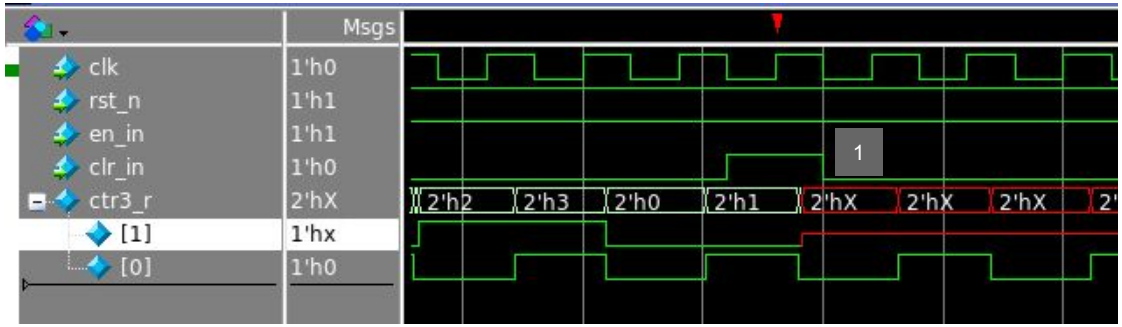
Gate-Level Simulation

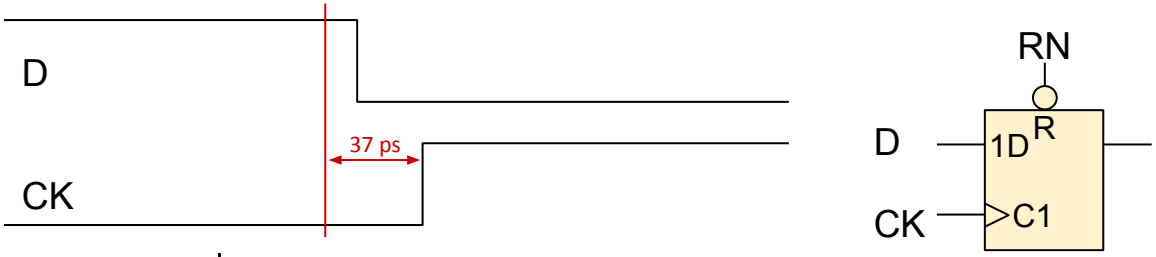
- Verilog **netlist** is instantiated in testbench
- Netlist instantiates **library components** whose functional and timing models are presented in a Verilog file
- Component and wire delays are obtained from a **standard delay format (SDF) file** and "back-annotated" to "slots" in the component models using simulator or SystemVerilog commands
- Gate-level simulation is useful for verifying
  - Asynchronous interfaces (synchronizers)
  - Resets
  - Detecting glitches in edge-sensitive signals
  - Structures added in synthesis (scan paths, clock gates, more in DT3)
- GL simulation is slow compared to RTL!



Debugging of Timing Violations in GL Simulation

- Symptom: Signal goes to unknown state (1)
- Simulator reports a setup time violation (2)
- The D input fell at time 2505 ps, the CK input rose 25 ps later at 2530 ps, while the setup constraint was 37 ps
- The component instance that got hurt was ctr3_tb/DUT/ctr3_r_reg_0_ (3)
- Using simulator GUI, figure out why this happened! (why did D arrive so late?)
- Notice: If slack ≥ 0 after optimization, timing violations should not occur. Sometimes improperly timed test data can cause them (e.g. wrong input delay)





Summary: Logic Design and Verification Flow