

Design Digitaler Schaltkreise

Introduction to Register Transfer Level design with Verilog

Asic and Detector Lab - IPE

Prof. Ivan Peric ivan.peric@kit.edu

Richard Leys richard.leys@kit.edu

Ilias: https://ilias.studium.kit.edu/goto_produktiv_crs_430424.html

Goal

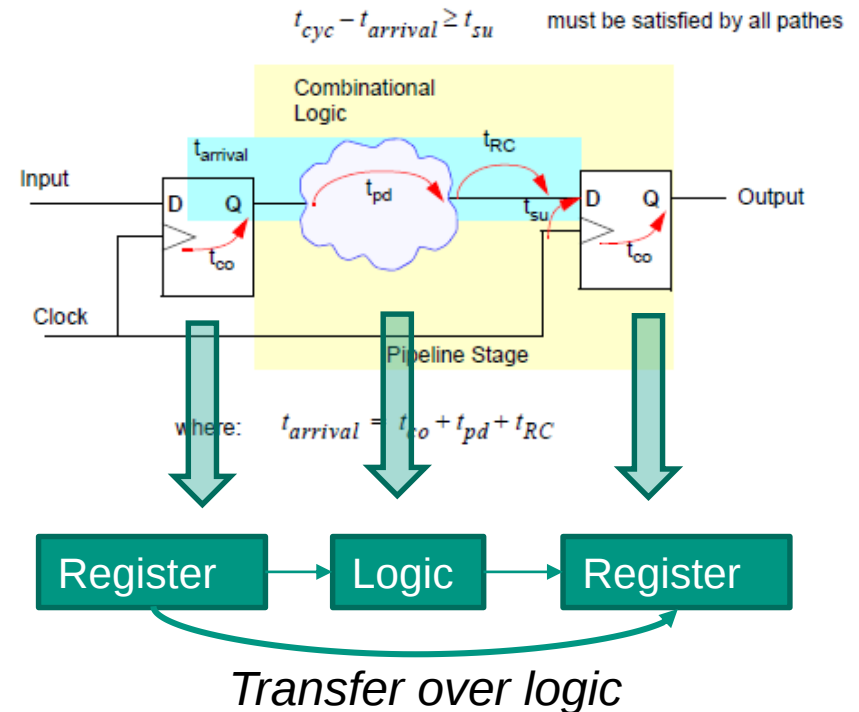
- Introduce Verilog/VHDL
 - Simulation/Synthesisable subsets
 - Note the basic semantic (HDL, RTL, DUT etc...)
- Understand how basic circuit elements map into Verilog
 - We will keep it very simple
- Introduce Simulation testbench basics
 - Keep it simple as well
- Note: Not all syntax elements are presented
- Note: You are free to use any languages/tools
- Analyse an example together
 - Recapitulate what has been seen before
 - Play around with the real tools
 - Do it yourself during first Übung

Links

- SystemVerilog/Verilog Reference
 - http://www.eda.org/sv/SystemVerilog_3.1a.pdf
- Some Tutorials on ASIC world
 - <http://asic-world.com>

Register Transfer Level (RTL)

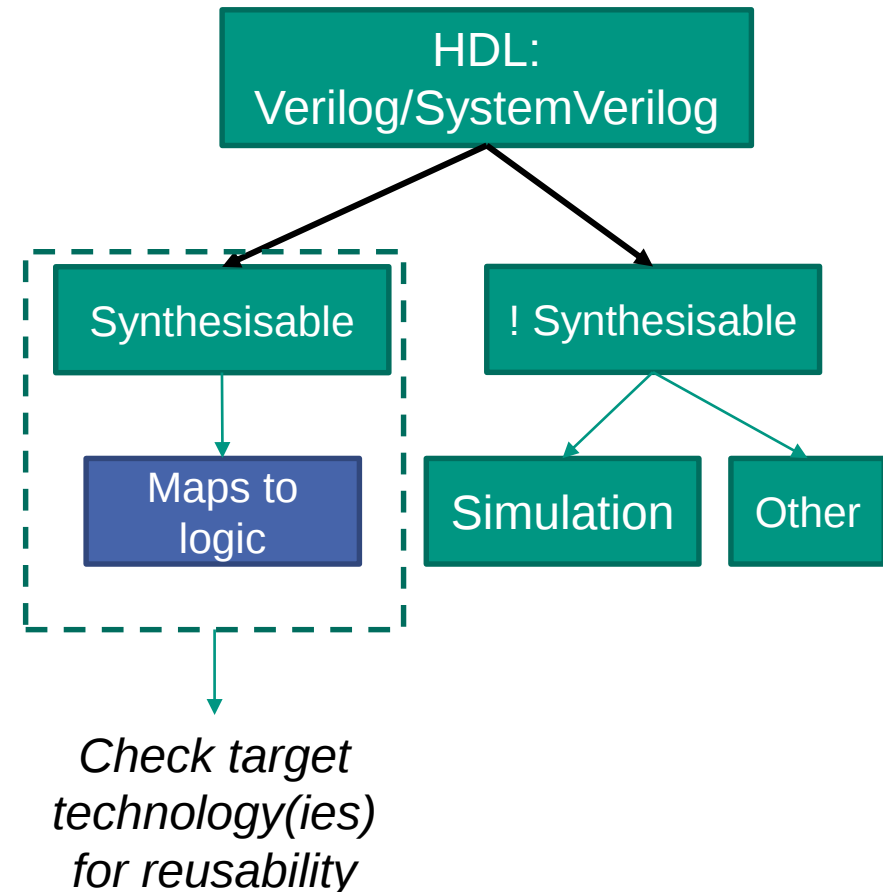
- Hardware Description Language (HDL)
- Verilog or VHDL description model circuits through known operators:
 - $\&$, $+$, $<<$, $>>$
 - Assignments for combinational logic or register
- They have some extra features to model complex designs:
 - “Modules” for hierarchies
 - Specific technology mapping
 - Simulation subset etc...



We will use Verilog in this lecture

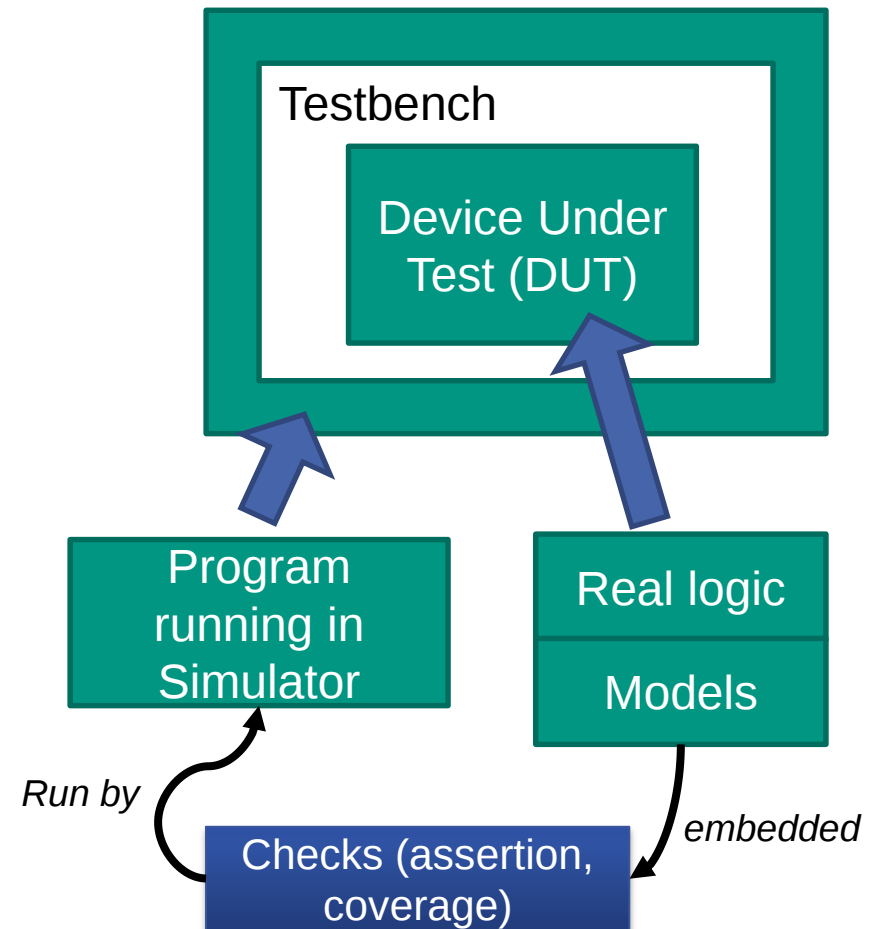
Synthesisable subset

- Verilog includes more features than just circuit modelling
- Ex: Simulation
 - Simulator time synchronisation
 - System Verilog: Support for object oriented programming
- **Careful:** Correct simulation does not mean the circuit is synthesisable or feasible
- **Careful:** Synthesisable subset can vary depending on the tools:
 - Keep the circuit description simple



Simulation Basics

- Synthesisable code: **Device**
- Simulation: Test the device
 - Drive its inputs
 - Check the outputs
 - This is a testbench
- Simulation code runs like a classical program: Standard programming
- Consequence: The simulation is a behavioural model of the device
 - Respects Input/Outputs
 - Only runs in simulator as “code”
 - No logic simulation: Faster
- Some models can be in the DUT as well:
 - Faster simulation
 - IP-Blocks whose logic content is not available
- Some Simulation checks are embedded in the DUT:
 - Coverage
 - Assertions



CIRCUIT MODELING BASICS

Value Representation

4-State Values:

- 0 } Logic levels
- 1 }
- Z → High Impedance
- X → Unknown

Register and wires

- Use wire for interconnections

Bus Size

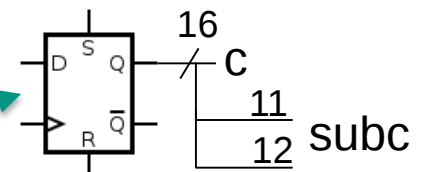
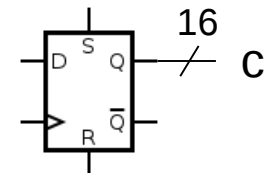
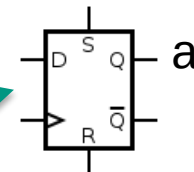
- Like C array
- Use Sublices

```
reg a;
```

```
wire b;
```

```
reg [15:0] c;
```

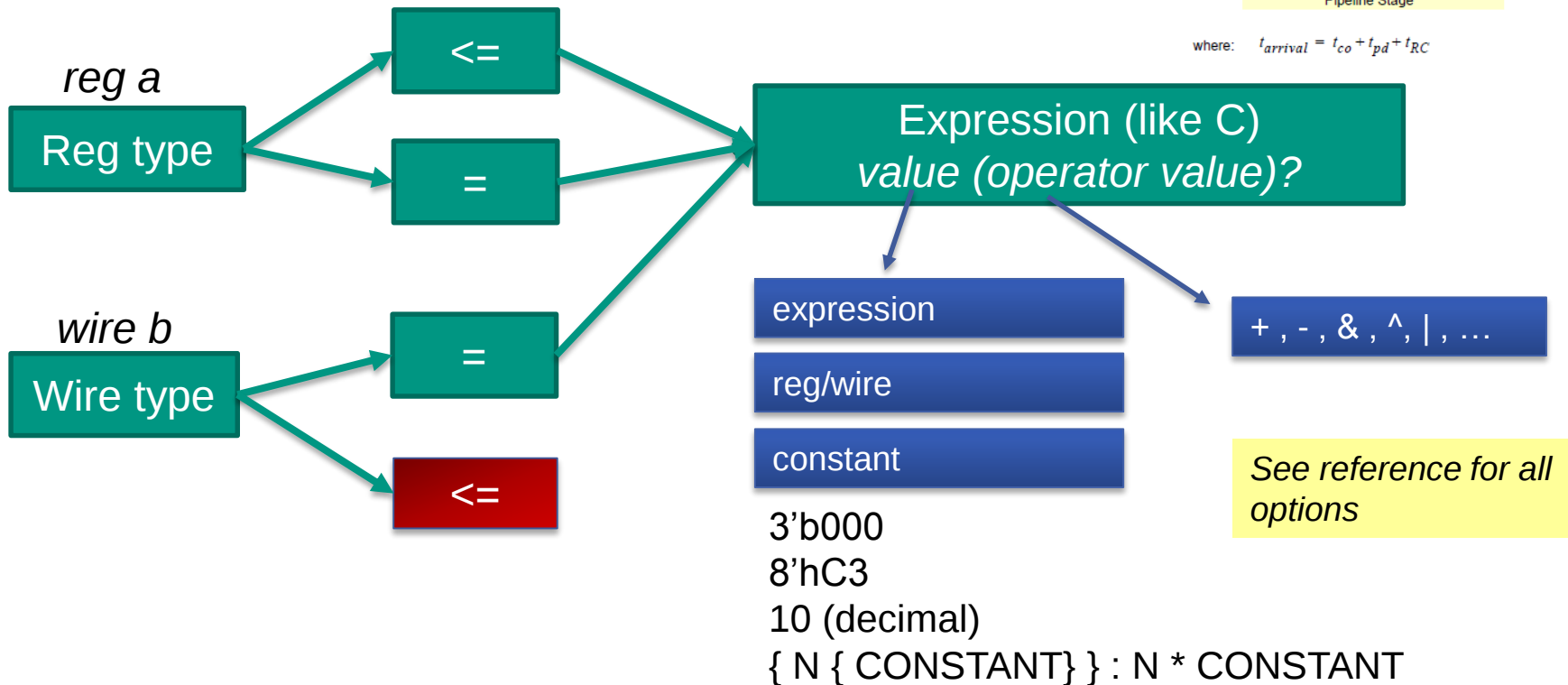
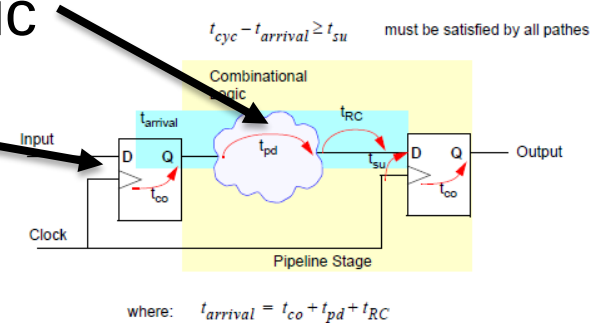
```
wire [1:0] subc = c [12:11];
```



Value assignments

■ = : Synchronous -> Combinational logic

■ <= : Asynchronous -> Register



Control Structures and syntax elements

- If-else
- Cases (like switch in C)
- assign
 - Simple Variable assignment
- Loops for multiple modules instantiation
- Blocks are delimited by “begin” ... “end”

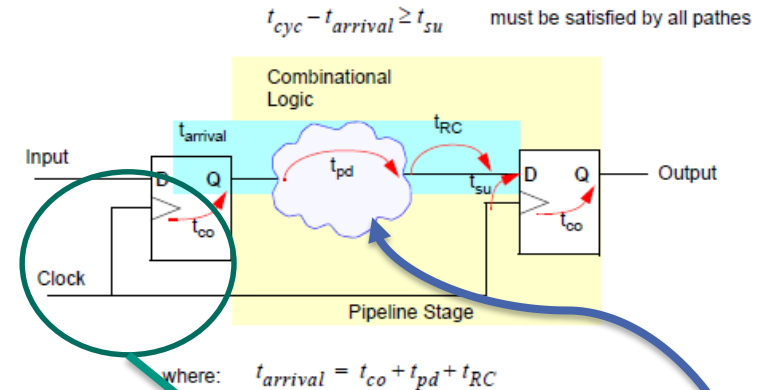
```
1 reg a ;  
2 reg b ;  
3 wire c ;  
4  
5 assign c = a ^ b ;  
6  
7 if (c == 1)  
8 begin  
9 ...  
10 end
```

Annoying!

See reference for all options

Asynchronous Process definition

- Use the always construct
 - Define signals triggering the block
 - Define edge sensitivity per signal
 - Posedge
 - Negedge
- Process on clock defines clock mapping for all the asynchronous assignments in the block
- Multiple Triggers per process:
 - One for clock
 - One for reset
 - More than that: Dangerous for implementation
- In Simulation: Does not matter



```

1 always @(posedge clk or posedge rst) begin
2   if (rst) begin
3     // reset
4   }
5   end
6   else begin
7     // Main Logic
8   }
9 end

```

Reset definition

Main Logic

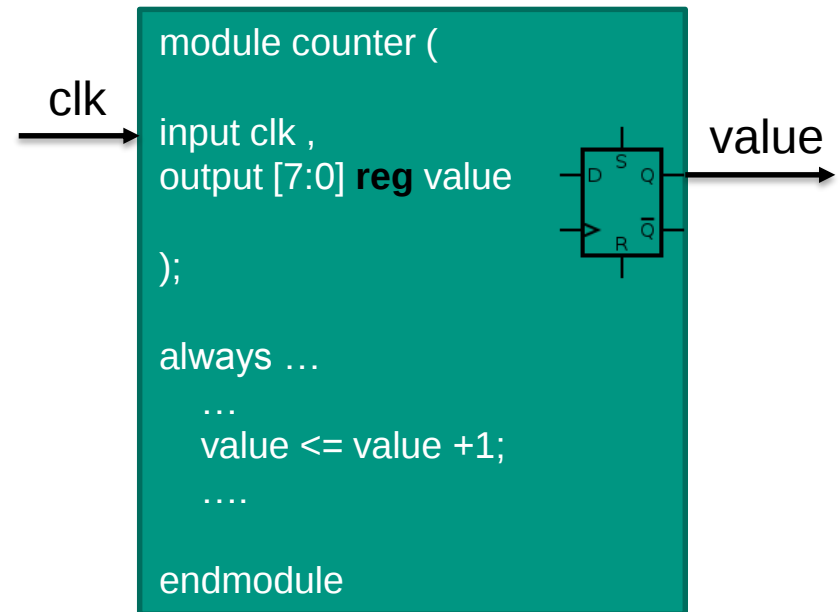
Reset Mapping

- Skip this subject for now...
- ...but...
- ...Design Rule: **All signals ALWAYS have a reset value**

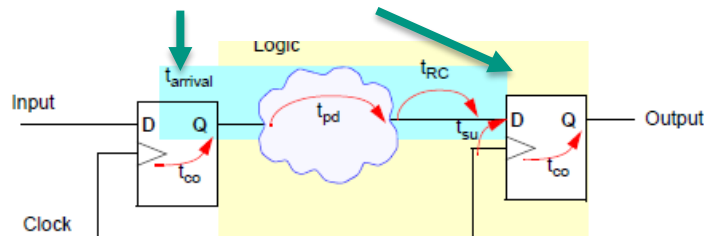
Hierarchy: Module Definition

- A Module is a “box” with
 - Input and Outputs
 - Logic inside
- The modules are instantiated where they are needed
- 1st Design Rule: All outputs are “reg”
 - If not, you don’t know how much combinational logic you end up with

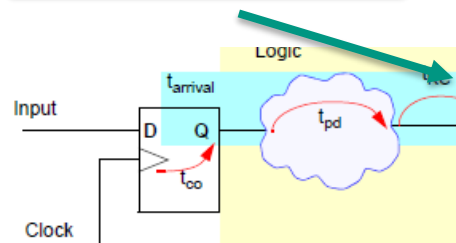
8bit counter



Reg output



Not Reg output



?

What will happen when performing timing analysis ?

Parameterised Modules

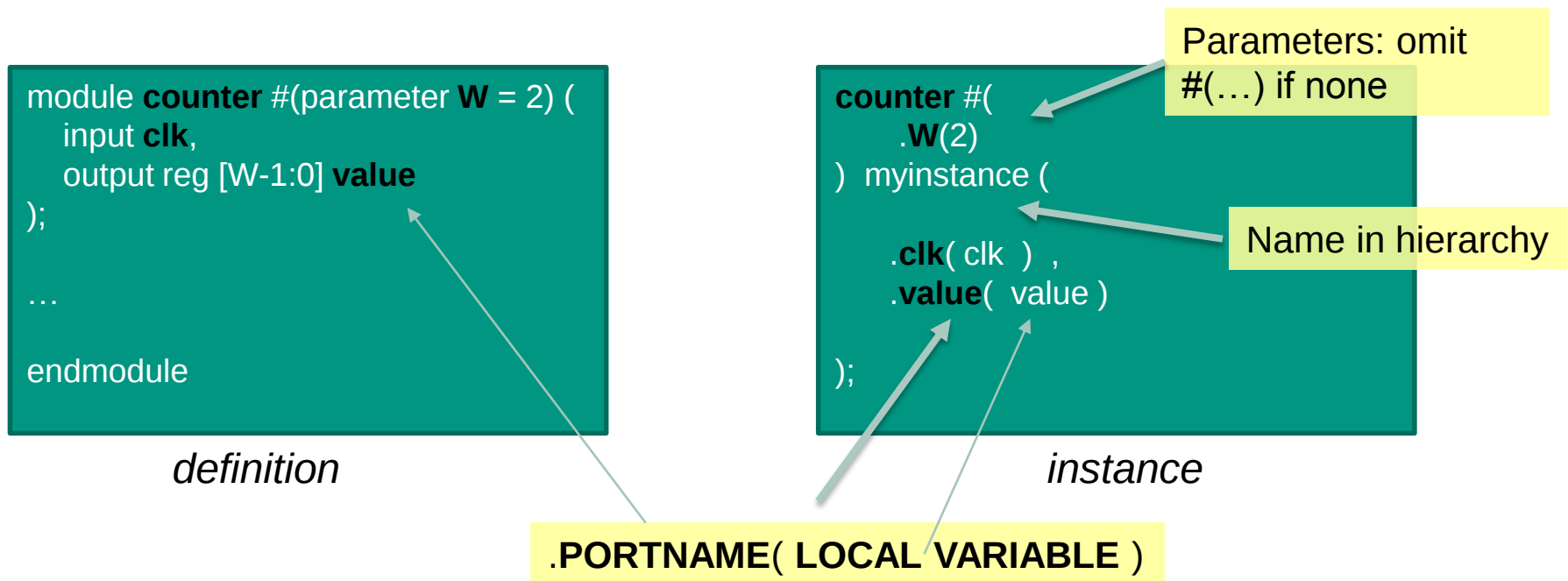
- Modules can have variables in their definition
- These variables are overridden when creating the module
- Counter is a good example:
 - The count register width can be made variable
 - It matters how wide when actually instantiating (i.e using) the counter

```
module counter #(parameter W = 8) (  
    input clk ,  
    output reg [ W -1 : 0 ] value  
);  
....  
endmodule
```

Default value if not
specified when
instantiated

Hierarchy: Module Instantiation

- Like a TypeDef in C
- There are multiple connection syntax, here is just the detailed safe one



Mind the parenthesis, commas, semicolons and stuff -> Waste of time

Simulation Setup and basic rules

- Testbench is a module
- It is like a software that checks the design
 - Software written in Verilog/SystemVerilog
- By convention called: `tb_top`
- Environment signals:
 - Use *always* without sensitivity list: runs continuously
 - -> Clock generator
 - Use *initial* for a sequence executed at the beginning
 - -> Reset
- First Golden Rule:
 - All signals must have a reset value
 - Never keep X in the waveforms after reset
 - Exception made for models of analog blocks like SRAMS (undefined until first read)

Simulation: Time primitives

- Synchronisation possible on:

- Pure simulator time

- Signals (wait, @)

- Use these features to write your tests

- Simulator time usage

- Write Clock generator

- Write initial sequence

- Static delays in Models

- Signal Synchronisation usage

- Easily drive inputs and test outputs

- Example: Wait on clock to change inputs

- Example: React on some output signals to call a task check

```
1 // Wait and release reset
2 #200ns res_n = 1;
```

```
1 // Wait for Reset to be released
2 wait (res_n==1);
```

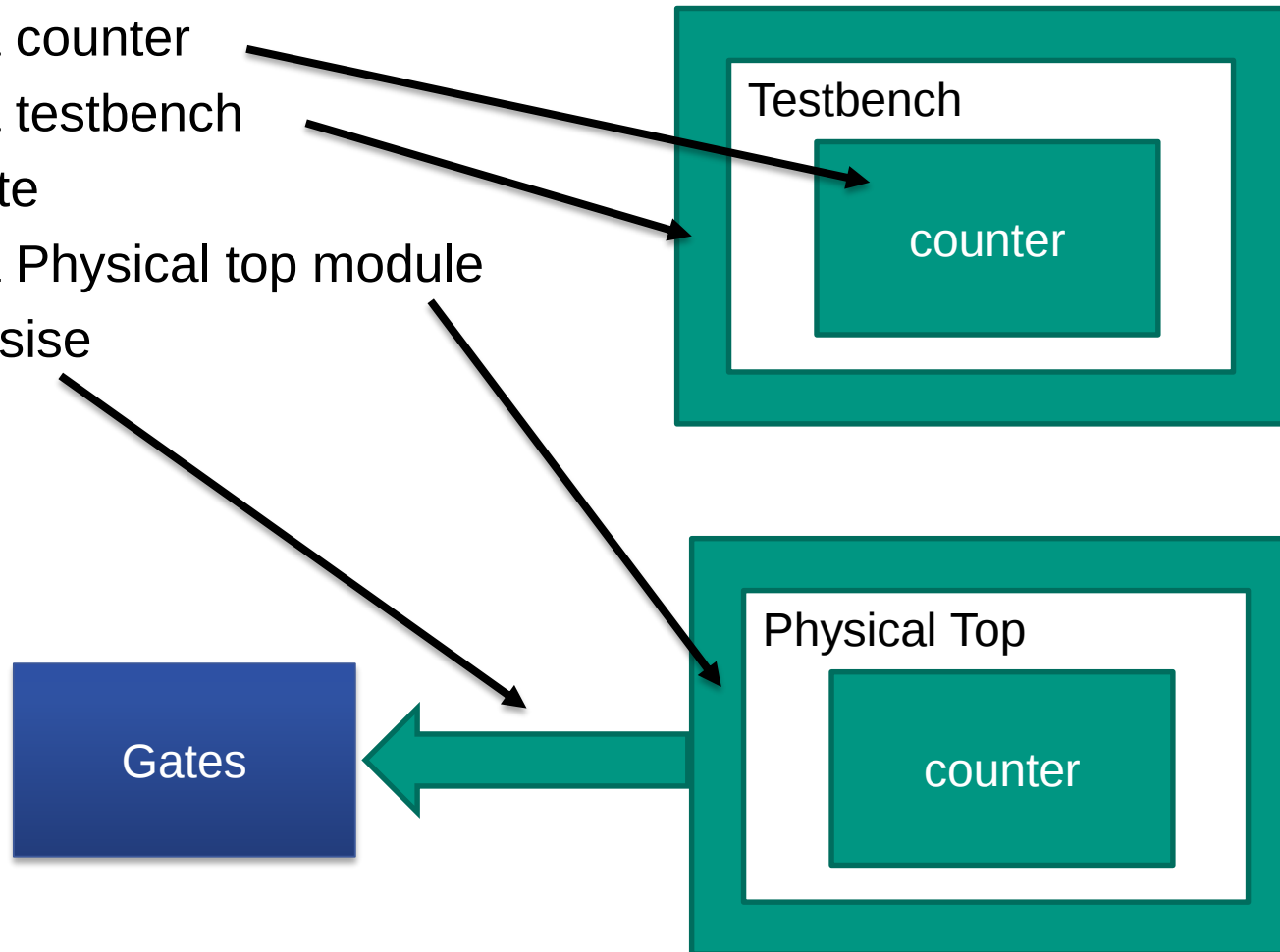
Other features

- Tasks, Functions and so on
- Events for complex model description
- External Software connection
- Complex Verification Environments like UVM (Functional Verification)
- Look at the SystemVerilog Reference
- Not the focus of the lecture

COUNTER EXAMPLE

Counter Example: Work tasks

- Write a counter
- Write a testbench
- Simulate
- Write a Physical top module
- Synthesise



Counter Specification

- +1 increment at each clock cycle
- Inputs:
 - Clock, Reset
 - Hold
 - Clear
- Outputs:
 - Value
- Parameter:
 - Value width
- Conditions:
 - Output is reg
 - All 0 at reset
- Function:
 - Clear sets all to 0 (like reset here)
 - Increment if not hold

```
1 module counter #(parameter SIZE = 8) (  
2  
3     // System  
4     input clk,  
5     input res_n,  
6  
7     // Control  
8     input hold,  
9     input clear,  
10  
11    // Output  
12    output reg [SIZE-1:0] value  
13  
14 );  
15  
16 always @(posedge clk) begin  
17  
18     // Reset  
19     if (!res_n || clear) begin  
20  
21         value <= {SIZE {1'b0}};  
22  
23     end  
24     // Main  
25     else begin  
26  
27  
28         if (!hold) begin  
29  
30             value <= value + 1;  
31  
32         end  
33  
34     end  
35 end  
36  
37 endmodule
```

Testbench Specification

- Module called “top”
- No Input/Output (highest level)
- Instantiate the counter
 - Set counter width
- Setup:
 - Clock generator
 - Reset Sequence
- Test Example:
 - Assert clear
 - Wait 2 cycles
 - Value Must be 0
 - Assert Hold
 - Wait 2 cycles
 - Value must be 2
 - Release Hold
 - Wait two cycles
 - Value must be 4
 - \$finish

```

1 module tb_top;
2
3 // Parameter
4 localparam SIZE = 8;

```

```

1 // Wiring
2 //-----
3 logic clk;
4 logic res_n;
5
6 logic hold;
7 logic clear;
8
9 logic [SIZE-1:0] value;
10
11
12 // Instance: Device Under Test (dut)
13 //-----
14 counter #(.SIZE(SIZE)) dut (
15     .clk(clk),
16     .res_n(res_n),
17
18     .hold(hold),
19     .clear(clear),
20
21     .value(value)
22
23 );

```

Connections
to local wiring

Clock and Reset

- Reset: Setup like in logic
- Clog gen: Clock of 100Mhz
 - 10ns period time
 - Start Simulator with a time unit of 1ns

```

1 // Reset
2 initial
3 begin
4   clk = 0;
5   res_n = 0;
6
7 // Functional initial values
8   hold = 0;
9   clear = 0;
10
11 // No reset for value, its reset is in the logic
12
13 // Wait and release reset
14   #50ns res_n = 1;
15
16 end
    
```

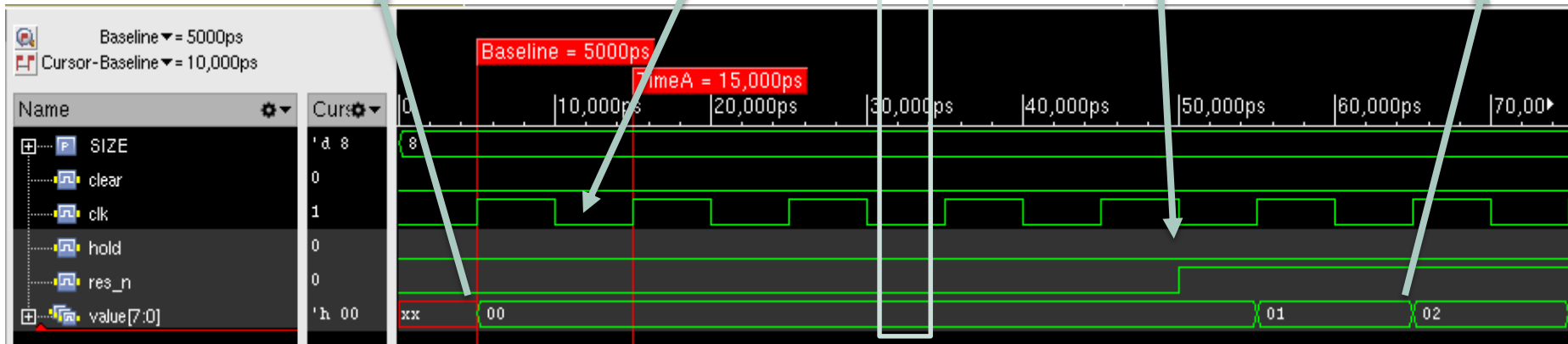
```

1 // Clock
2 always
3 begin
4   #5ns clk <= ~clk;
5 end
    
```

Reset in logic: good ☺

All set ☺

Running



Write a test

- Synchronise on clocks to wait and assert control signals
 - Use negedge for better readability
- Use normal control structures for tests
 - Skip this for now
- In case of error: use \$error
- Not very detailed here
- Try in the lab work
- Use \$finish to stop simulation after your tests

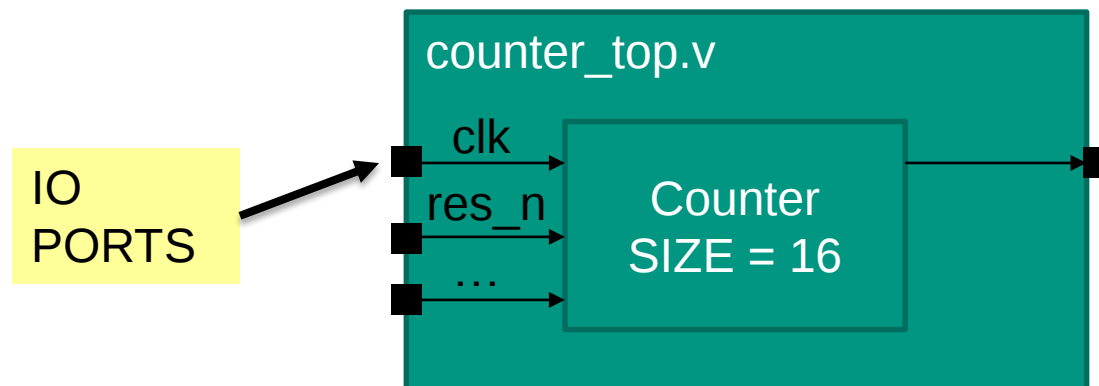
```

1 //
2 clear = 1;
3 @(negedge clk);
4 @(negedge clk);
5
6 clear = 0;
7 @(negedge clk);
8 @(negedge clk);
9 .....
10 hold = 1;
11 @(negedge clk);
12 @(negedge clk);
13
14 hold = 0;
15 @(negedge clk);
16 @(negedge clk);
17
18
19 #100ns $finish();

```

Very Simple Synthesis

- Create a Real Physical Design
 - Create a “top” file, with an instance of the counter
 - The size of the counter is then fixed
- Use Cadence RTL Compiler
- Write a TCL script to read the design data
- Write Constraints for Clock
- Synthesise



Standard Cells :
Slow corner

Constraints.sdc

Summary

- Verilog as HDL language:
 - Synthesisable subset
 - Use to describe the logic
 - Non Synthesisable subset
 - Mainly just standard code running in simulator
- Simulation
 - How to write a simple testbench
 - Took notice of embedded checks like assertion and coverage, maybe we will see this once in the Lab work
- Implementation:
 - Create a “Physical” Top with final design, IO and so on
 - First step is synthesis
- Next step: Do it yourself in the Übung.

End Slide