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

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
- <u>Note: Not all syntax elements are presented</u>
- <u>Note: You are free to use any languages/tools</u>
- Analyse an example together
    - Recapitulate what has been seen before
    - Play around with the real tools
    - Do it yourself during first Übung

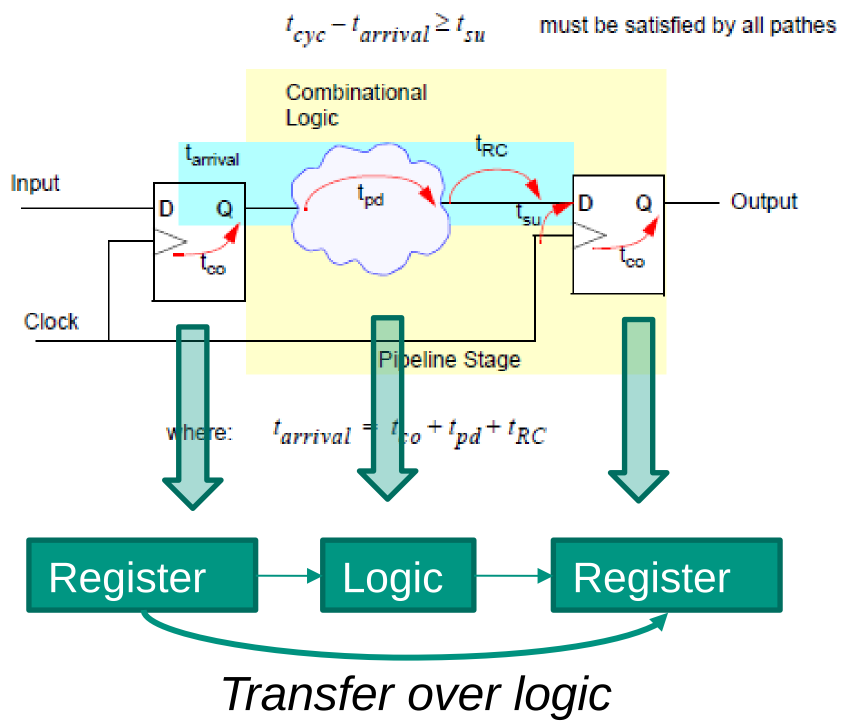
Links

- SystemVerilog/Verilog Reference
  - http://www.eda.org/sv/SystemVerilog_3.1a.pdf
- Some Tutorials on ASIC world
  - http://asic-world.com
- Linux Command Line:
  - http://linuxcommand.org
- Open Source Toolchain:
  - https://www.idyria.com/access/osi/files/builds/adl/

INTRODUCTION

Register Transfer Level (RTL)

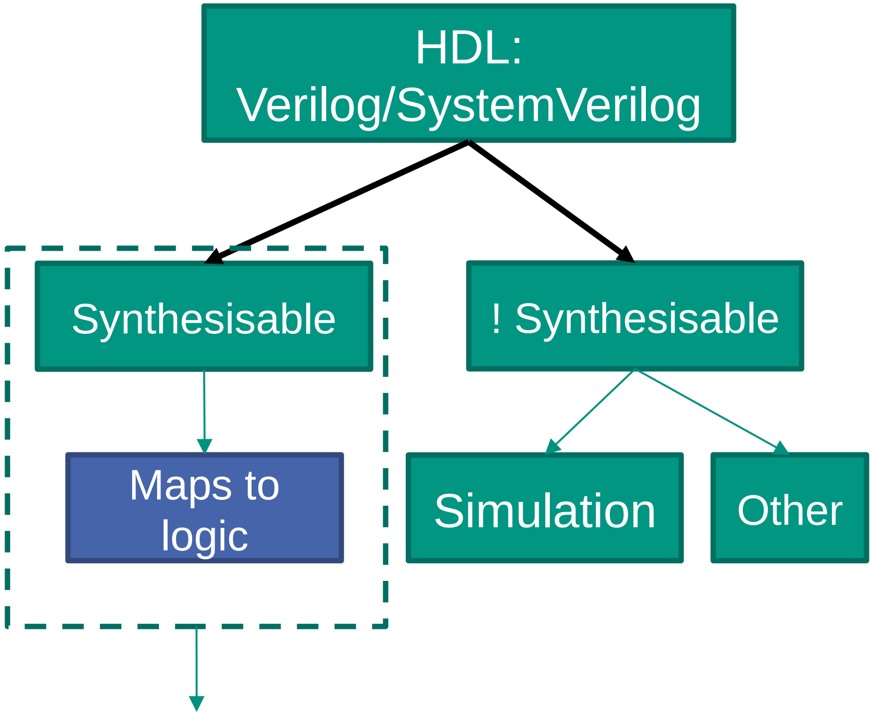
- Hardware Description Language (HDL)
- Verilog or VHDL description model circuits through known operators:
  - &, + , << ,>>
  - Assignments for combinational logic or register
- They have some extra features to model complex designs:
  - “Modules” for hierarchies
  - Specific technology mapping
  -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



We will use Verilog in this lecture

Synthesisable subset

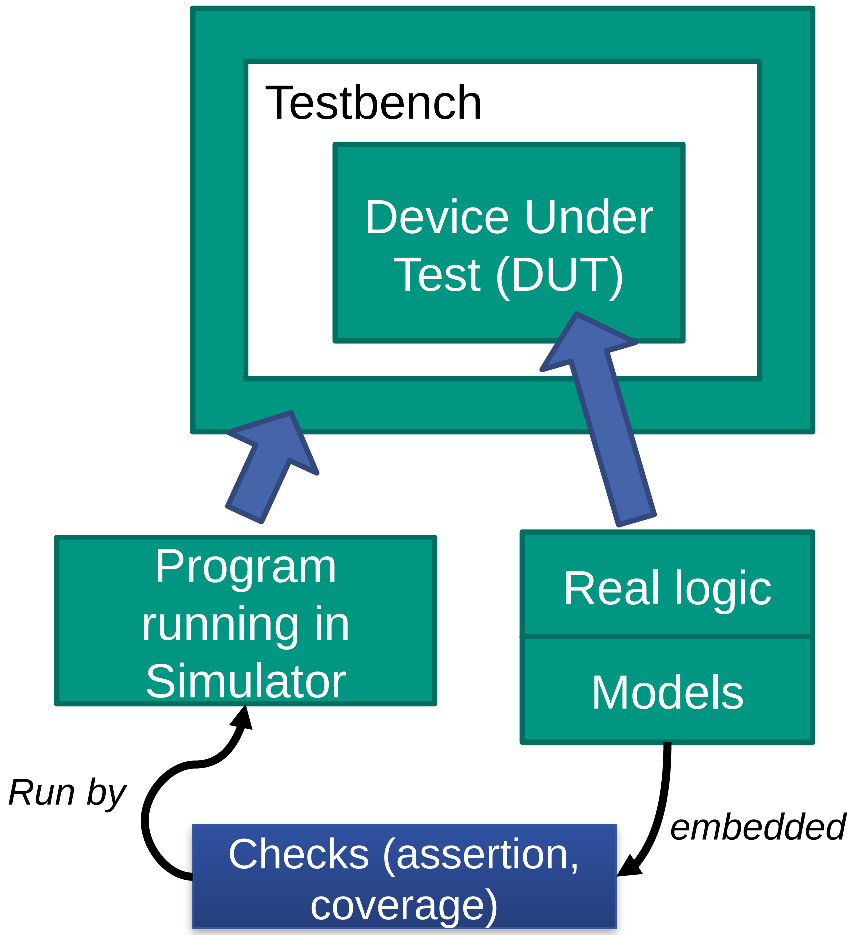
- Verilog includes more features than just circuit modelling
- Ex: Simulation
  - Simulator time synchronisation
  - System Verilog: Support for object oriented programming
- **Careful:** Correct simulation does not mean the circuit is synthesisable or feasible
- **Careful:** Synthesisable subset can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



Check target technology(ies) for reusability

Simulation Basics

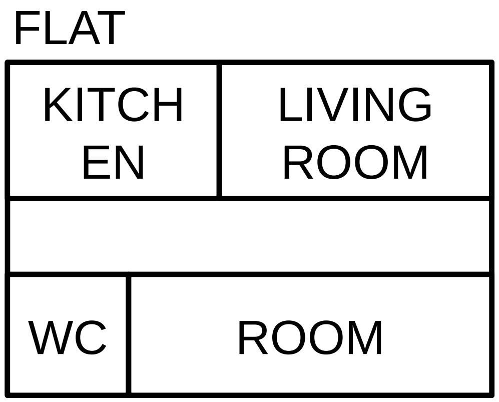
- Synthesisable code: **Device**
- Simulation: Test the device
  - Drive its inputs
  - Check the outputs
  - This is a testbench
- Simulation code runs like a classical program: Standard programming
- Consequence: The simulation is a behavioural model of the device
  - Respects Input/Outputs
  - Only 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

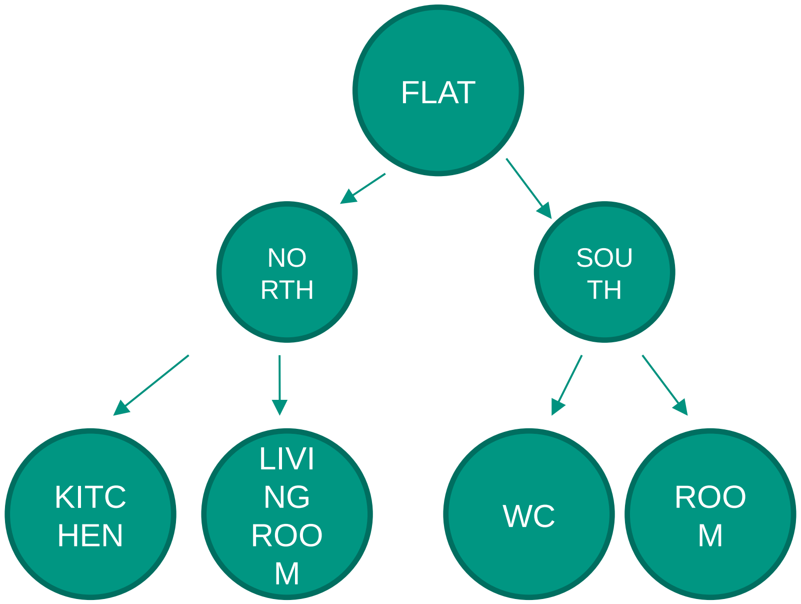


HIERARCHY ELEMENTS

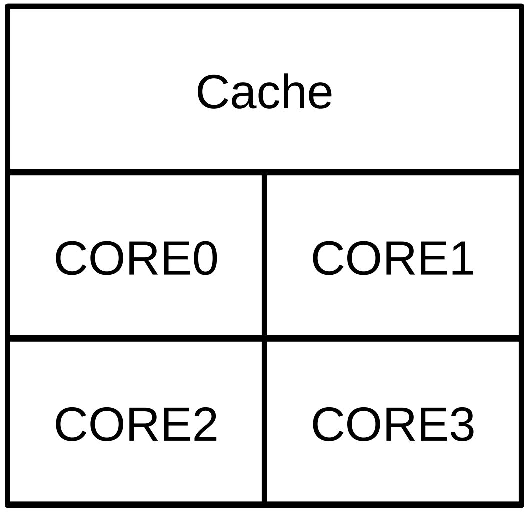
Hierarchy Modeling?

- Circuits have to main components:
  - Logic Modeling
  - Structure
- Think about software:
  - Instructions: + , / , -
  - Structure: Functions, Classes etc…
- Simple Analogy: Flat
  - Organisation of the rooms: Hierarchy
  - Content of the rooms: primitives
- In Verilog:
  - Modules
- In VHDL:
  - Entities

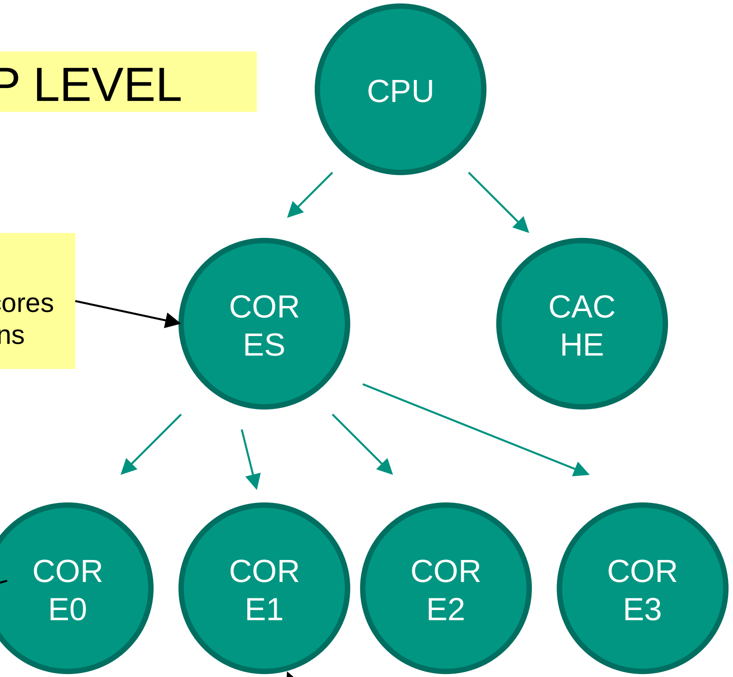




Hierarchy Example for a CPU

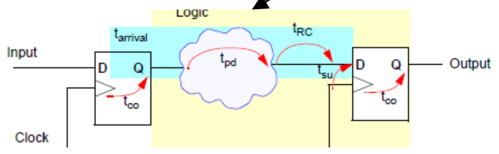


TOP LEVEL



Virtual:
Logical Container for the cores and their interconnections

Inside Core:
Actual Logic + More Instances (like ALU)



Instances:
CORE defined only once.
0…3 are copies/instances

Hierarchy: Module Definition

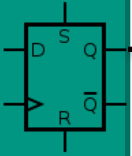
- A Module is a “box” with
  - Input and Outputs
  - Logic inside
- The modules are instantiated where they are needed
- 1st Design Rule: All outputs are “reg”
  - If not, you don’t know how much combinational logic you end up with

8bit counter specification

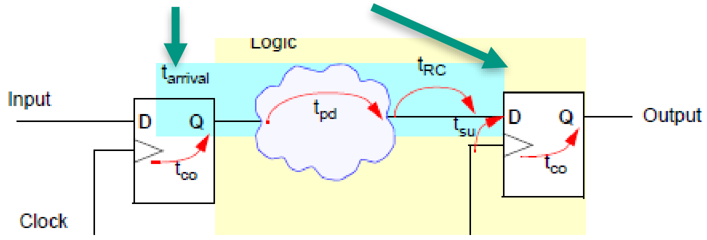
clk

```
module counter (

input clk ,
output [7:0] reg value

);

always …
    …
    value <= value +1;
    ….

endmodule
```

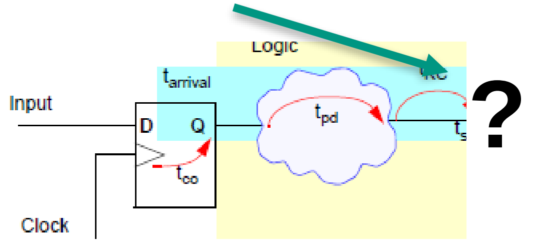
value



Reg output



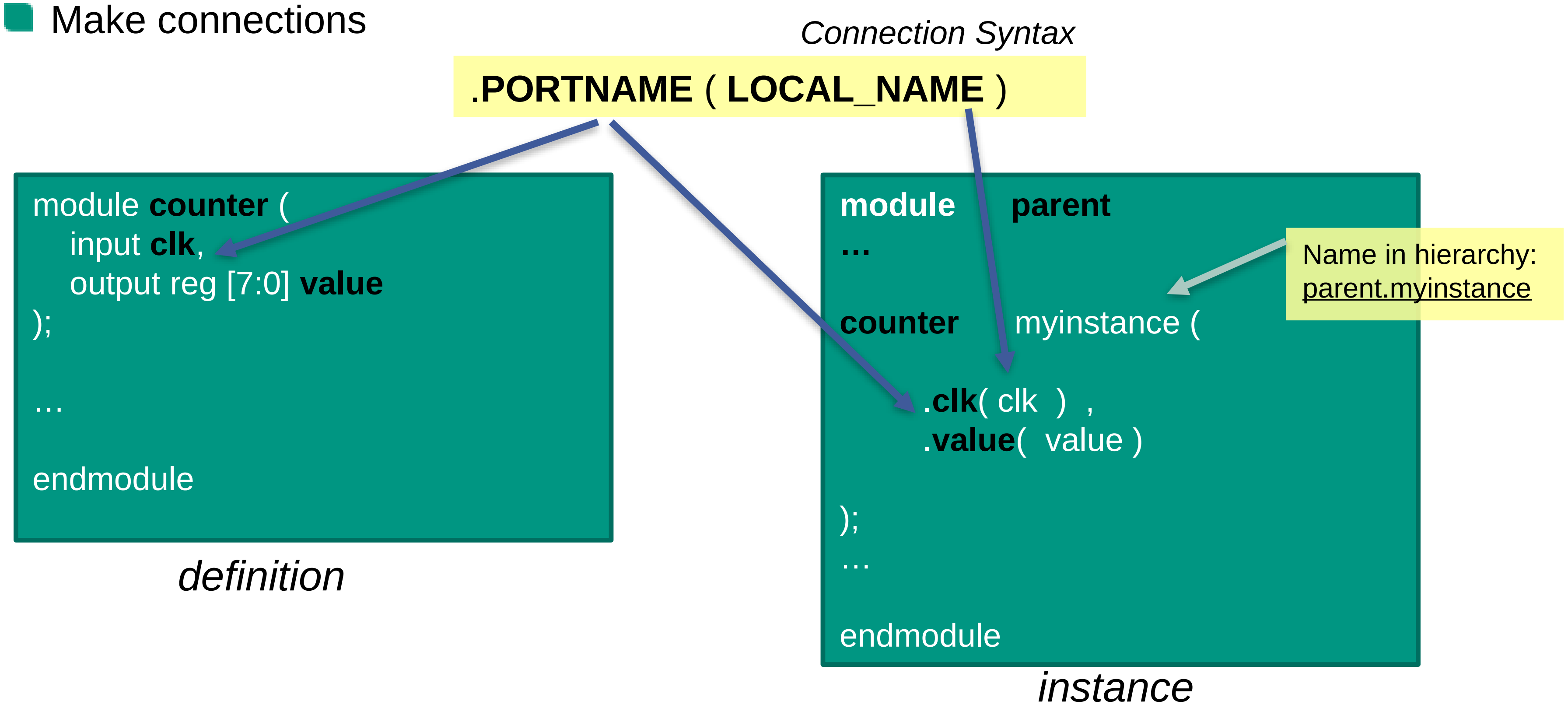
Not Reg output



What will happen when performing timing analysis ?

Hierarchy: Module Instantiation

- Select the name of the module to use
- Give the instance a local name
- Make connections



Connection Syntax

.PORTNAME (LOCAL_NAME)

```
module counter (
    input clk,
    output reg [7:0] value
);

…

endmodule
```

definition

```
module    parent
…

counter    myinstance (

        .clk( clk  )  ,
        .value(  value )

);
…

endmodule
```

instance

Name in hierarchy:
parent.myinstance

Mind the parenthesis, commas, semicolons and stuff !!

Parameterised Modules

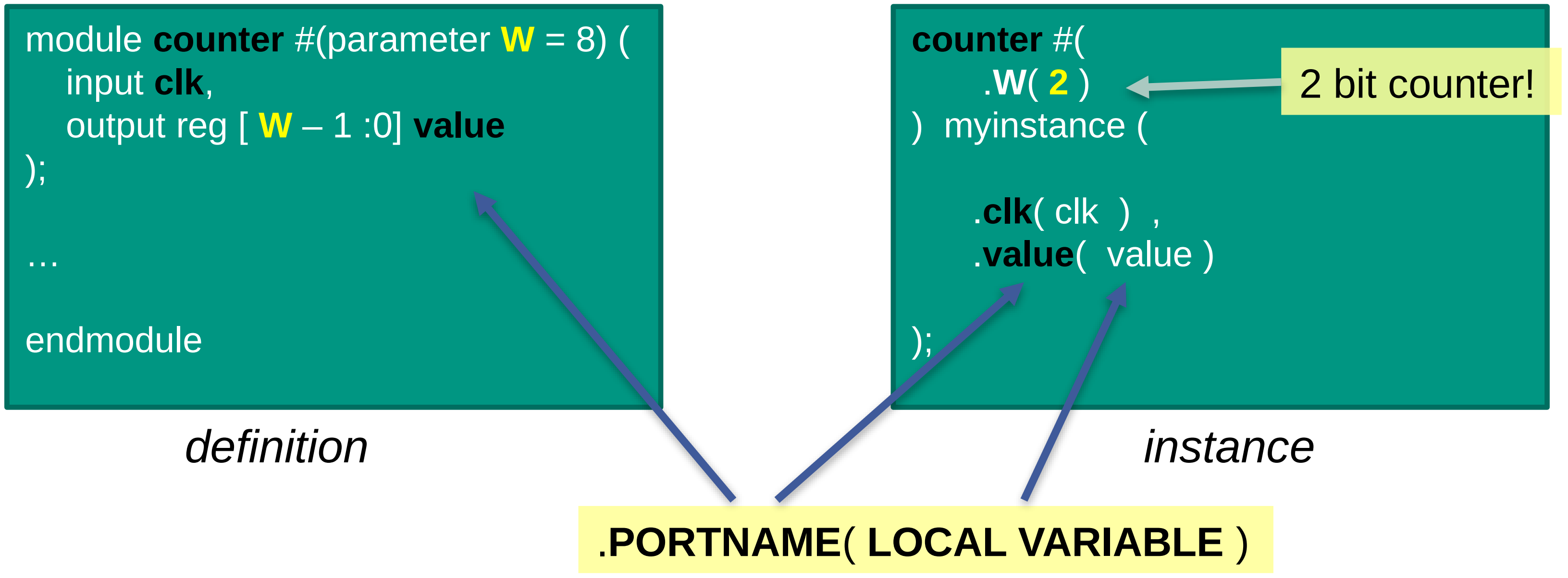
- Modules can have variables in their definition
- These variables are overridden when instantiating the module
- Counter is a good example:
  - The count register width can be made variable
  - It matters how wide when actually instantiating (i.e using) the counter

```
module counter #(parameter W = 8)  (

input clk ,

output reg [  W -1 : 0 ]  value

);

….

endmodule
```

8 is the default value if not specified when instantiated

Hierarchy: Parameterised Module Instantiation

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



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

…

endmodule
```

definition

```
counter #(
      .W( 2 )
)  myinstance (

     .clk( clk  )  ,
     .value(  value )

);
```

instance

Mind the parenthesis, commas, semicolons and stuff!!

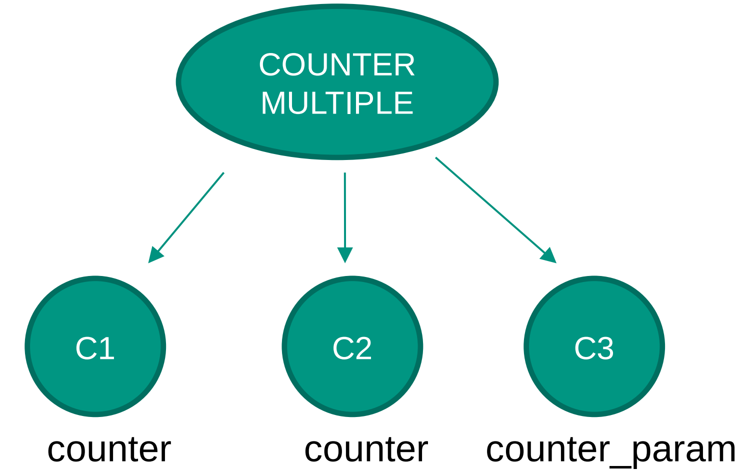
Hierachy: Multiple Instanciations

- Of course, the same syntax can be used repeatedly inside a module definition

```
module counter_multiple
…

counter c1 (
    .clk(clk),
     .value(value0)
);

counter c2 (
    .clk(clk),
    .value(value1)
);

counter_param #(.W(16)) c3  (
    .clk(clk),
    .value(value2)
);

endmodule
```

Same 8bit counter 2 times

Counter with parameter, 16bit

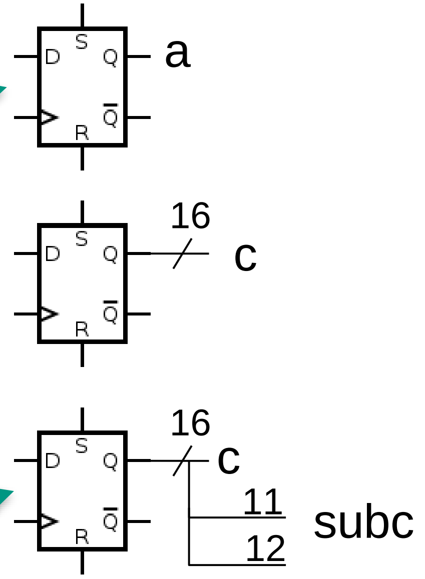


CIRCUIT MODELING

Value Representation

- 4-State Values:
  - 0
  - 1 } Logic levels
  - Z → High Impedance
  - X → Unknown
- Register and wires
  - Use wire for interconnections
- Bus Size
  - Like C array
  - Use Sublices

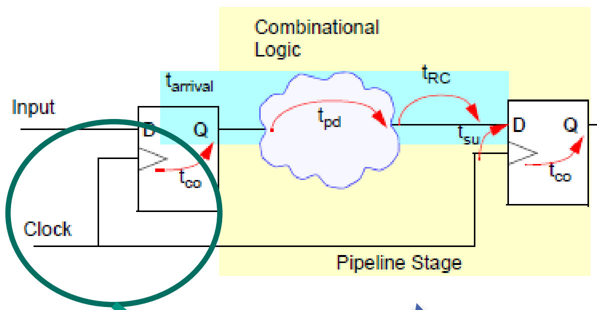
```
reg  a;

wire b;

reg [15:0] c;

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



Clock Synchronous 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

$t_{cyc} - t_{arrival} \ge t_{su}$ must be satisfied by all pathes



where: $t_{arrival} = t_{co} + t_{pd} + t_{RC}$

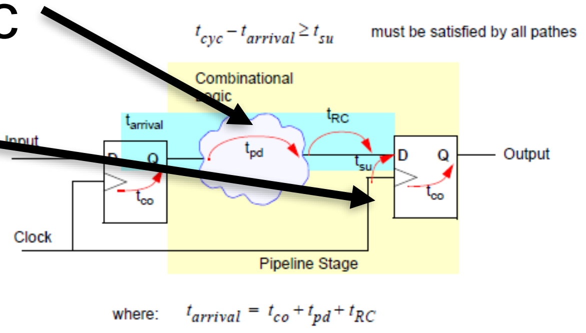
```
always @(posedge clk or posedge rst) begin
    if (rst) begin
    // reset

    end
    else begin

    end
end
```

Reset definition

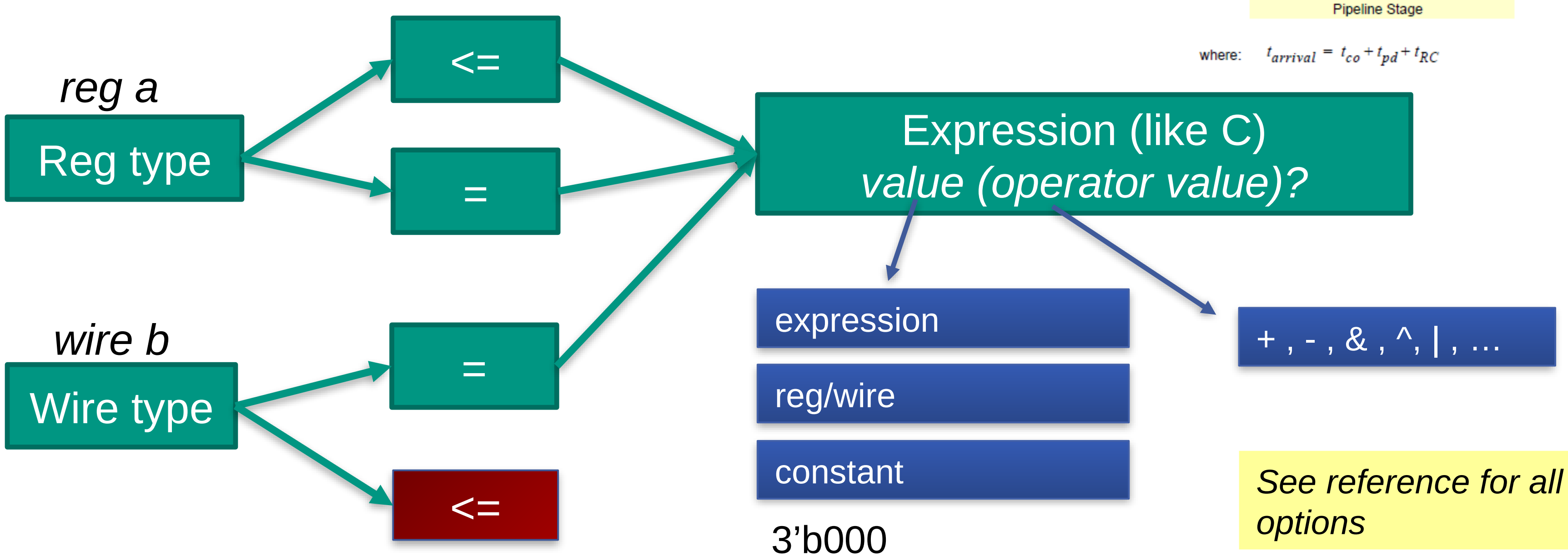
Main Logic

Value assignments in synchronous block

- **=** : Synchronous -> Combinational logic
- **<=** : Asynchronous -> Register



$t_{cyc} - t_{arrival} \geq t_{su}$ must be satisfied by all pathes
Combinational Logic
$t_{arrival}$
t_{RC}
t_{pd}
t_{su}
t_{co}
Input
D
Q
Output
Clock
Pipeline Stage
where: $t_{arrival} = t_{co} + t_{pd} + t_{RC}$



3’b000
8’hC3
10 (decimal)
{ N { CONSTANT} } : N * CONSTANT

See reference for all options

Control Structures and syntax elements

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

```
reg a ;
reg b ;
wire c ;

assign c = a ^ b ;

if (c == 1)
begin
  ...
end
```

Annoying syntax requirements

See reference for all options

Reset Mapping

- Skip this subject for now…
- …but…
- …Design Rule: <u>**All signals ALWAYS have a reset value**</u>

```
always @(posedge clk or posedge rst) begin
    if (rst) begin
    // reset

    end
    else begin

    end
end
```

Reset assignments here!!

SIMULATION BASICS

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 , or something with “tb” in name
- 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

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

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

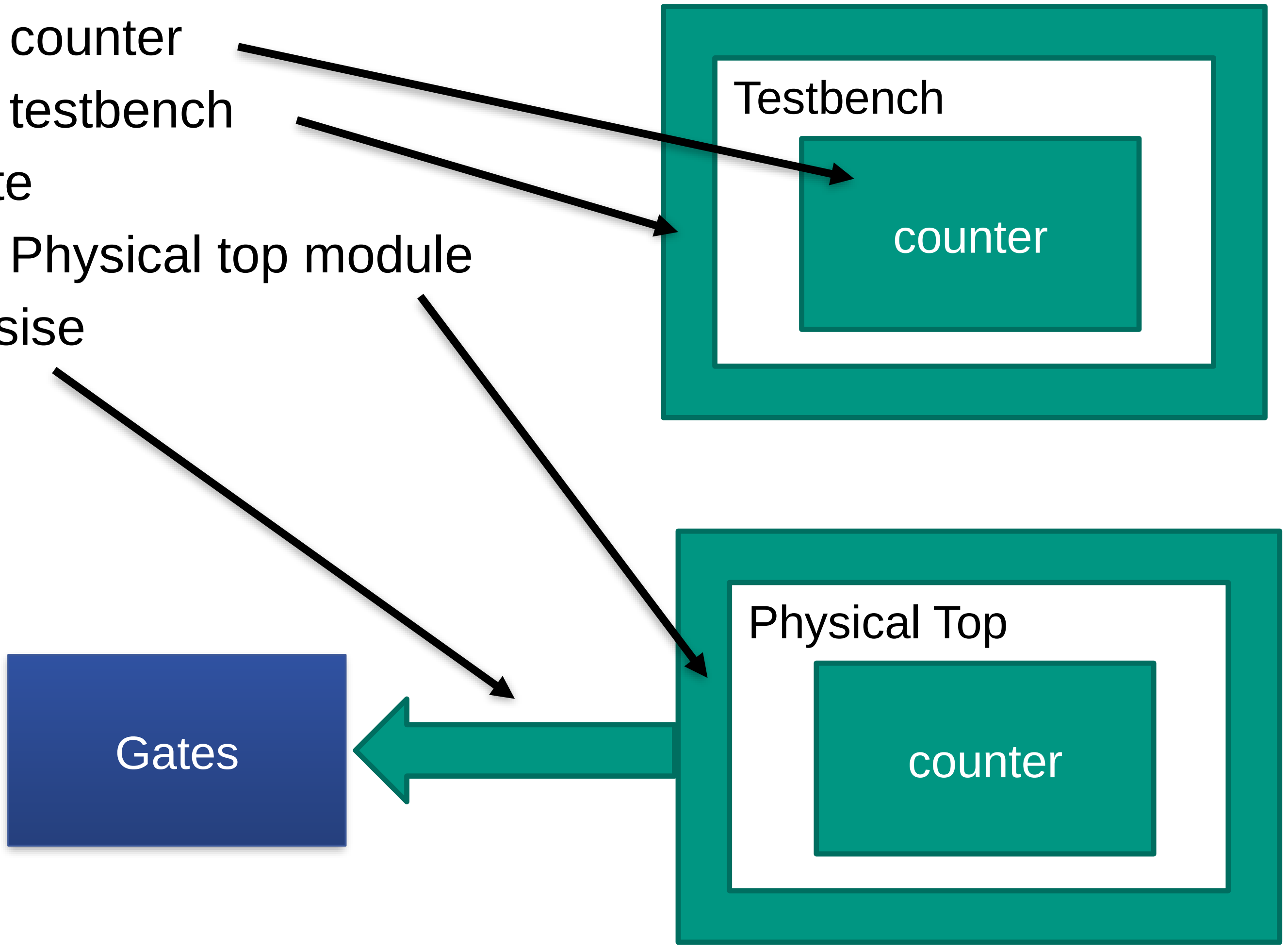
Other features

- 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 if 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

```verilog
module counter #(parameter SIZE = 8) (

  // System
  input clk,
  input res_n,

  // Control
  input hold,
  input clear,

  // Output
  output reg [SIZE-1:0] value

);

  always @(posedge clk) begin

    // Reset
    if (!res_n || clear) begin

      value <= {SIZE {1'b0} };

    end
    // Main
    else begin


    if (!hold) begin

      value <= value +1;

    end

  end
end

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

```
module tb_top;

// Parameter
localparam SIZE = 8;
```

```
// Wiring
//--------------------
logic clk;
logic res_n;

logic hold;
logic clear;

logic [SIZE-1:0] value;


// Instance: Device Under Test (dut)
//---------------
counter #(.SIZE(SIZE)) dut (
    .clk(clk),
    .res_n(res_n),

    .hold(hold),
    .clear(clear),

    .value(value)

);
```

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

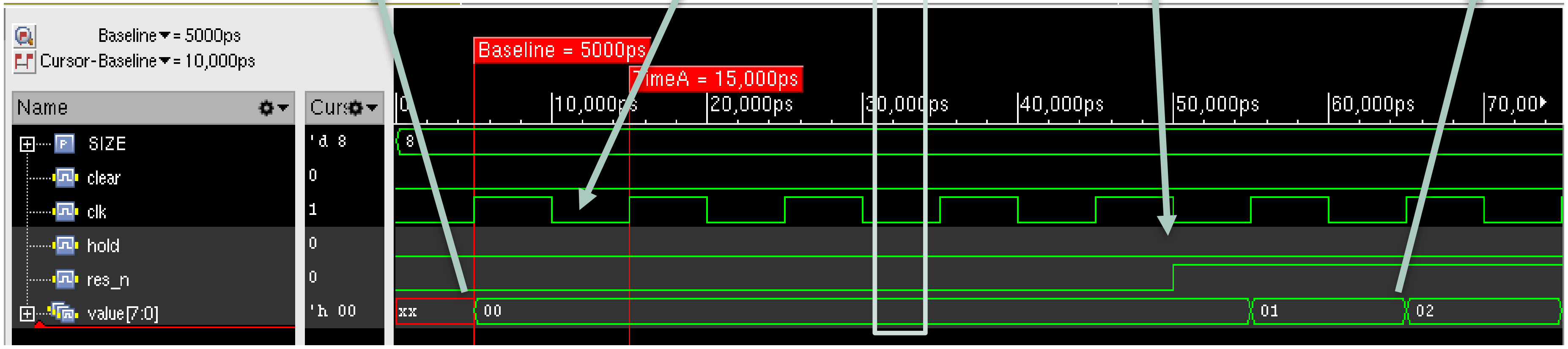
```
// Reset
initial
begin
  clk = 0;
  res_n = 0;

  // Functional inital values
  hold = 0;
  clear = 0;

   // No reset for value, its reset is in the logic

   // Wait and release reset
   #50ns res_n = 1;

end
```

```
// Clock
always
begin
     #5ns clk <= ~clk;
end
```

Reset in logic: good ^^

All set ^^

Running



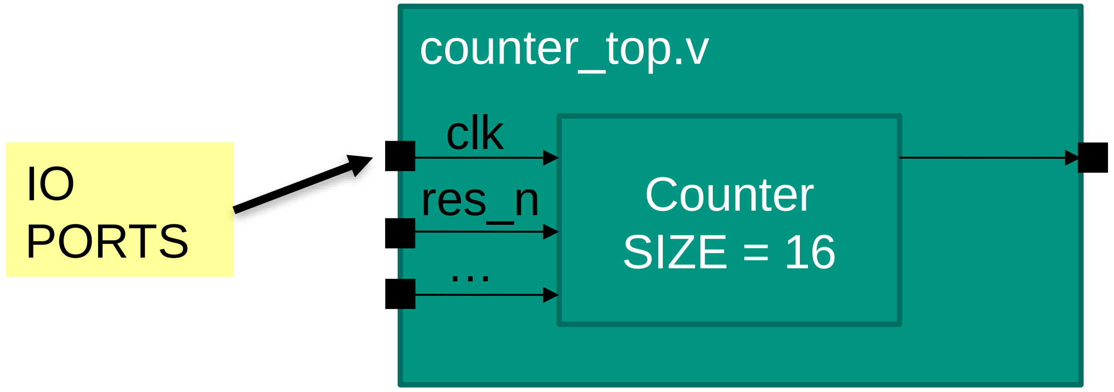
Write a test

- 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
- <u>Not very detailed here</u>
- <u>Try in the lab work</u>
- Use $finish to stop simulation after your tests

```
//
clear = 1;
@(negedge clk);
@(negedge clk);

clear = 0;
@(negedge clk);
@(negedge clk);
    -------------
hold = 1;
@(negedge clk);
@(negedge clk);

hold = 0;
@(negedge clk);
@(negedge clk);


#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