Circuit and System Design

Verilog Hardware Description Language

FAKULTÄT **ELEKTROTECHNIK** UND **INFORMATIONSTECHNIK**

Content

- Verilog HDL
  - Overview
  - Basics
  - Procedures
  - Hierarchical Elements (modules, functions, tasks)
  - Gate-Level Modeling
  - Behavioral Modeling
  - System Functions
- Circuit Simulation and Verification

Lecture Goals

- This lecture will:
  - Present basic concepts of Verilog HDL
  - Give methods for describing and modeling digital circuit blocks
  - Give verification strategies
  - → Fundamentals for the use of verilog during lab excercies
  - Give motivation for self-study

- This lecture will **NOT:**
  - Provide a complete language reference of the Verilog HDL
  - Replace the exercises in lab excercies

Verilog - Overview

Motivation

- Hardware Description Language HDL
- Description of digital circuits in machine-readable text form
- Application for
  - Modeling
  - Simulation
  - Verification
  - Synthesis
- Ability to describe Hardware properties, e.g.
  - Sequential processes
  - Parallel processes

Verilog Hardware Description Language

- 1983/84 developed by Phil Moorby (Gateway Design Automation) as **Simulation language** „Verilog"
- 1985 extended language scope and simulator „Verilog-XL"
- 1988 available synthesis tools based on Verilog (Synopsys Design Compiler)
- 1990 Acquisition by Cadence Design Systems
- 1993 85% of all ASIC chip designs in Verilog
- since 1995 open standard → IEEE Standard 1364-1995 (Verilog-95)
- 2001 extension IEEE Standard 1364-2001, „Verilog2001"

Literature

- Standard
  - IEEE Standard Verilog® Hardware Description Language; IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)
    - *Available on IEEE Explore*
- Books
  - HDL Chip Design; Douglas J. Smith; Doone Publications; 1996
  - The Verilog hardware description language; Thomas, Donald E. ; Moorby, Philip R.; Springer; 2002

- Online Tutorials
  - http://www.asic-world.com/verilog/veritut.html
  - http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf

Abstraction Levels of Verilog Descriptions

Abstraction

- Algorithmic/Behavioral
  - High-level design construct

```
…
real a,b,c;
always c = #1 a*b;
…
```

- Register-Transfer-Level (RTL)
  - Data flow between registers
  - Basis for synthesis

```
…
reg [7:0] a, b;
always @(posedge clk) begin
    a<=b+1;
end
…
```

- Gate-Level
  - Description of logic gates and their connections

```
…
wire out, in1, in2;
nor(out,in1,in2);
…
```

- Switch-Level
  - Description of circuits (Transistors) as well as storage nodes

```
…
wire g, s, d;
nmos(d,s,g);
…
```

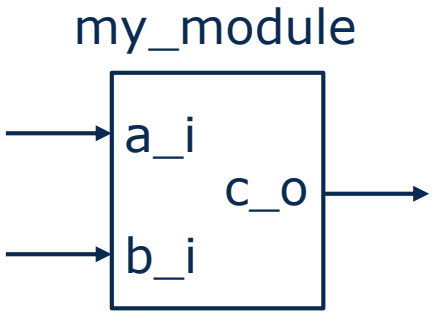
Overall simulation of all abstraction levels is possible

Verilog Extensions

- VerilogA
  - A language for modelling analog components (time and value continuous)
  - No pure digital Verilog construct is possible
  - Simulation requires pure analog simulator
  - Applicaitons:
    - Building blocks modeling
    - Modeling of analog components (e.g. OPAmp)
- VerilogAMS
  - Extension of Verilog language scope for modeling analog signals
  - allowed Mixed-Signal Simulation using digital and analog solver
  - Applications:
    - Modelling of Mixed-Signal systems
- System Verilog
  - Hardware description and verification language (HDVL)
  - Contains more complex data types (similar to C/C++), object-oriented programming is possible as well as verification
  - Applications:
    - Modelling and verification of more complex systems
    - Test benches and Verification IP

Verilog - Basics

Design Unit Module

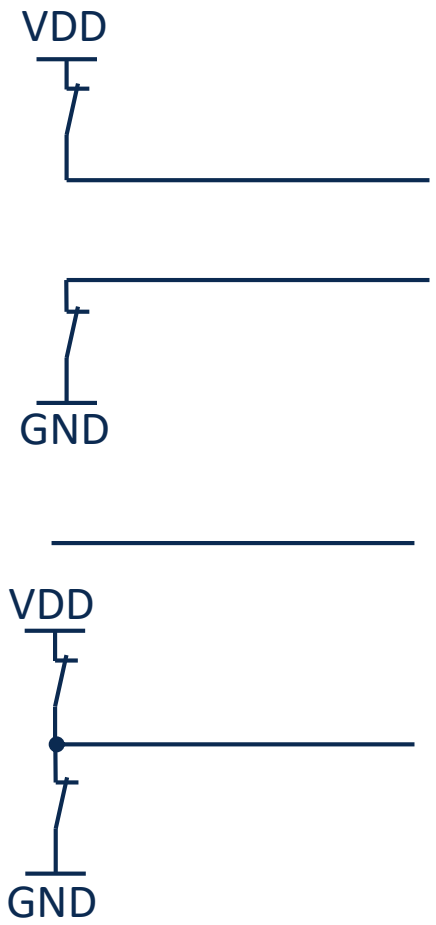



```
// Company  :   tud
// Author   :   hoeppner
// E-Mail   :   <email>
// Filename :   my_module.v
// Project Name       :   p_cool
// Subproject Name    :   s_cool28soc
// Description        :   <short description>
// Create Date        :   Mon Jan 30 14:10:45 2012
// Last Change        :   $Date: 2012-10-24 12:07:59 +0200$
// by                 :   $Author: scholze $
//-----------------------------------------------------------
module my_module (a_i, b_i, c_o);
 input a_i;
 input b_i;
 output c_o;
…

…
endmodule
```

- Coding Guideline:
  - for RTL 1 module per Verilog .v File
  - for libraries (e.g. standard cells) several modules per file

Logic Level

- Verilog describes 4 logic levels
  - logic 1 : 1‘b1
  - logic 0 : 1‘b0
  - High impedence/tri-state : 1‘bz
  - unknown : 1‘bx

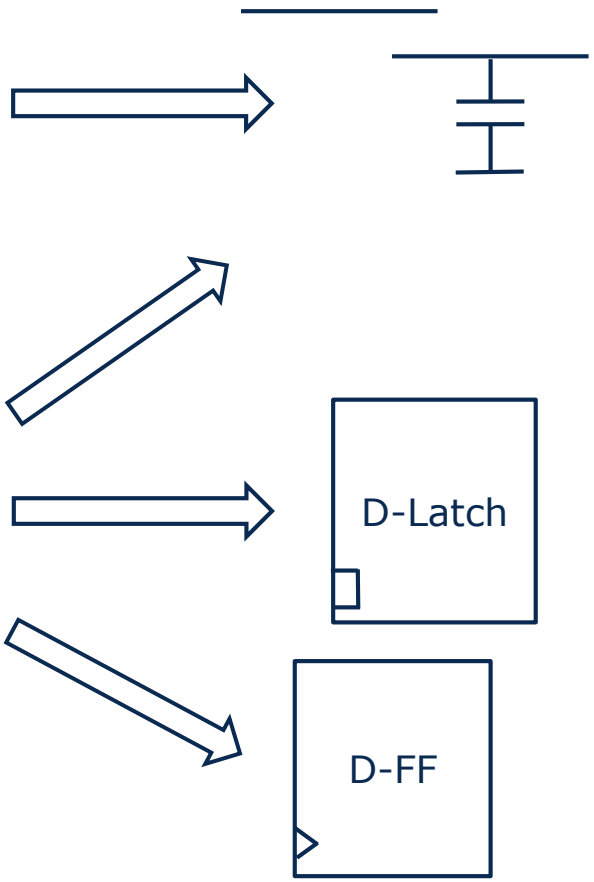


Data Types

- There is 2 different data types for describing electrical signals:
- „Net" data type
  - Stores no values
  - Must be continuously driven
  - Example:
    - **wire** → electrical conductor

- „Register" Data type
  - Abstraction of a storage node
  - Describes „Register" in simulator
  - Stores value between individual assignments
  - Examples:
    - **reg** → logic Signal
    - integer → 32 Bit „Register"
    - real → real number
    - time → point in time

  - **Be aware:** Register Data type can be used to describe:
    - Combinatinal signals
    - memories (Latch, FlipFlop, SRAM)

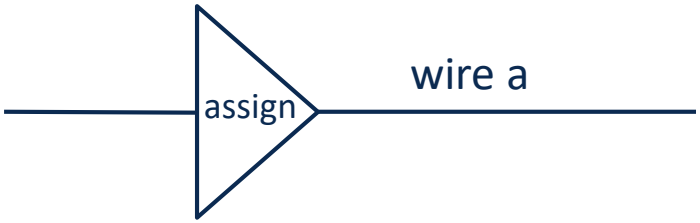


wire Data Type

- Describes electrical signals
- Continuous assignment for values through
  - **assign** statement
  - Gate or switch
  - Module instances
- Behaves in multiple assignments:

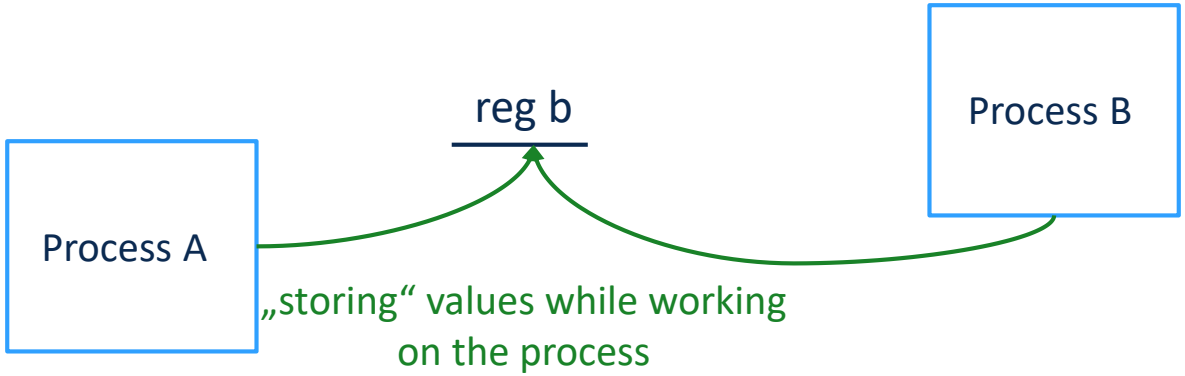
```
wire a,b,c,d;
assign a=b;
nor(b,c,d);
```

wire	0	1	x	z
0				
1				
x				
z				



reg Data Type

- Register Data type reg as Abstraction of a storage node
- Describes „Register" in simulator
- Assignments by processes
- Stores values between individual assignments



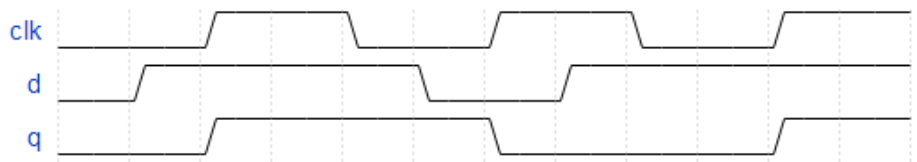
Reg Data Type Examples

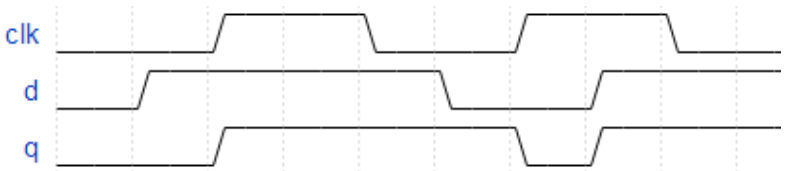
- Register Data type reg can describe:
  - Combinational signals
  - Latches and RAM cells (state-control)
  - Flip-Flops (edge-control)

```
//combinational
wire a;
reg b;
always @(a) begin
   b=a;
end
```

```
// D-FlipFlop
wire d,clk;
reg q;
always @(posedge clk) begin
   q<=d;
end
```

```
// D-Latch
wire d,clk;
reg q;
always @(clk or d) begin
   if (clk==1'b1) q<=d;
end
```





The reg data type does not necessairly describe a physical register!

Assignment of Constant Values

- Clear assignment of constants is possible:

```
//Example of constant assignment

//sized
reg [7:0] a =8‘d23;          //decimal value
reg [7:0] b =8‘b00010111;    //binary value
reg [7:0] c =8‘h17;          //hexadecimal value

//unsized
reg [7:0] d =23;             //leads to 8‘b00010111
reg [7:0] e =‘h3;            //leads to 8‘b00000011
reg [7:0] f =‘hf3;           //leads to 8‘b11110011

// ‘_‘ for better readability
reg [7:0] f =8‘b0001_0111; //binary value
```

Parameter

- The usage of parameters can make Source Code clearer and easier to be reused (facilitates **Reusability**)
- Parameter acts as **a constant value**
- Parameters can be overwritten during module instatiation

```
//Example of module with parameters
module incrementer (in_i,out_o);
   parameter C_DWIDTH=4;
   parameter C_STEP=2;

   input  [C_DWIDTH-1:0] in_i;
   output [C_DWIDTH-1:0] out_o;

   assign out_o=in_i+C_STEP;
endmodule
```

Bus Definitions

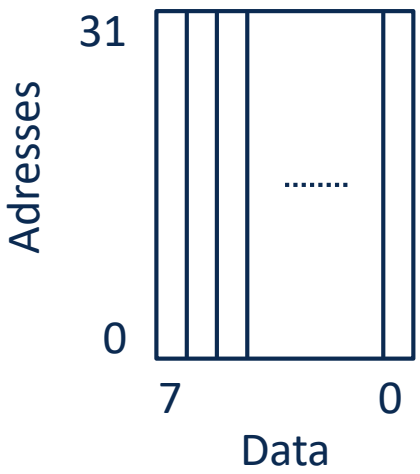
- Signals and module pins can be collected into Buses
- Examples:

```
//8-Bit Register without Reset
module reg_8 (clk, d_i, q_o)
   input clk;
   input [7:0] d_i;
   output [7:0] q_o;
   reg [7:0] q;
   always @(posedge clk) begin
      q<=d_i;
   end
   assign q_o=q;
endmodule
```

```
//signal assignments with Buses
wire a;
wire [7:0] b,c,d,e; //8-Bit Bus
reg  [3:0] r;    //4-Bit Register

assign a=b[7];          //1-Bit assignment
assign c=8‘b0011_11xz; //8-Bit constant

//Combined Signal (Concatenation)
assign d={b[6:4],1‘b0,c[3:0]};

//Zuweisung von 8‘b00000000
assign e={8{1‘b0}};

always @(b) begin
   r[3:0]=b[3:0]; //assigning part of a word
end
```

Memory Arrays

- Memory blocks can be defined as Arrays




```
wire we;
wire [4:0] addr;
wire [7:0] data_write;
wire [7:0] data_read;

// memory array
reg [7:0]   memory_32x8 [0:31];

//write logic
always @(data_write or we or addr) begin
  if (we==1‘b1) begin
      memory_32x8[addr]=data_write;
  end

//continuous read
assign data_read=memory_32x8[addr];
```

Driving Strengths

- Signal drivers can have different driving strengths

	Name	Level	Abbr.	
1	`supply1`	7	Su1	
	`strong1`	6	St1	← Default
	`pull1`	5	Pu1	
	`large1`	4	La1	
	`weak1`	3	We1	
	`medium1`	2	Me1	
	`small1`	1	Sm1	
	`highz1`	0	HiZ1	
0	`highz0`	0	HiZ0	
	`small0`	1	Sm0	
	`medium0`	2	Me0	
	`weak0`	3	We0	
	`large0`	4	La0	
	`pull0`	5	Pu0	
	`strong0`	6	St0	← Default
	`supply0`	7	Su0	

Assignment of Driving Strengths

- Continuous Assignment

```
//Examples of driving strengths of continuous assignment
assign (strong1, pull0)  a = b; //Unsymmetrischer Treiber
assign (highz1, strong0) c = d; //Open-Drain Treiber
```

- Gate Circuits

```
//Examples of driving strengths of gates
and (strong1, pull0) (out,in1,in2);
nor (strong1, highz0)(out,in1,in2);
```

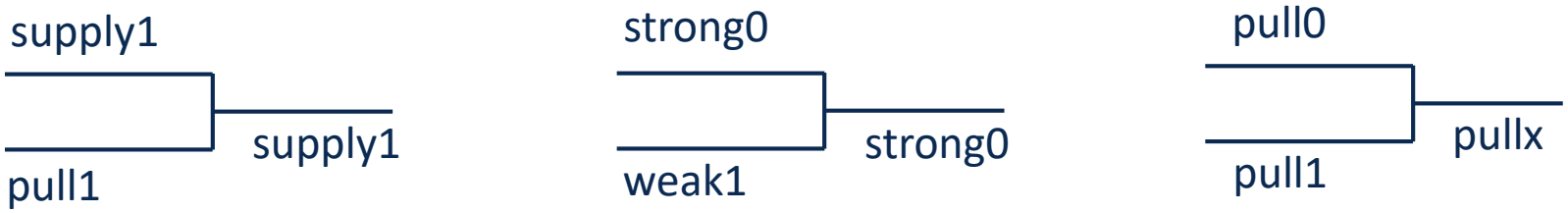
Combination of different driving strengths

- Nets can be driven through multiple assignments

```
//Example of driving strengths
wire a, b, c;
assign (strong1, pull0) a = b;
assign (pull1, strong0) a = c;
```

- Conflicts in interconnects can be resolved
- Most important cases:
  - Different driving strengths; different logic levels
    - → logic level for stronger signal
    - → driving strength for stronger signal
  - Same driving strengths; different logic levels
    - → logic level is 1‘bx with the same driving strength

- Examples:



Logical Operators in Verilog

- Operators combine and/or modify signals

{}	concatenation
+ - * /	arithmetic
%	modulus
> >= < <=	relational
!	logical negation
&&	logical and
||	logical or
==	logical equality
!=	logical inequality

~	bit-wise negation
&	bit-wise and
|	bit-wise or
^	bit-wise xor
^~ or ~^	bit-wise equivalence
&&	logical and
&	reduction and
|	reduction or
^	reduction xor
^~ or ~^	reduction xnor
<<	left shift
>>	right shift
?:	conditional

Priority of Operators

! ~	Highest priority
* / %	
+ -	
<< >>	
< <= > >=	
== != === !===	
&	
^ ^~	
|	
&&	
||	
?:	Lowest priority

```
//Example of priorities of operators
wire a,b,c,d;
assign a= d==c&b ? d|b&c : c;
```

d	c	b	a
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Arithmetic Operators

- Examples of arithmetic operators

```
wire [4:0] a,b,c,d,e,f;

//Arithmetic Operators
assign a= 27+2‘b01; // Result 28
assign b= 27+5;     // Result 0
assign c= 27-2‘b01; // Result 26
assign d= 3*2;      // Result 6
assign e= 5/2;      // Result 2
assign f= 10%3;     // Result 1
```

Logical Operators and Comparison Operators

- Examples of logical operators and comparison operators

```
wire a,b,c,d,e,f,g,h,i,j;

//Relational and Logic operators
assign a=(2‘b01==2‘b10); //Result 1‘b0
assign b=(2‘b01!=2‘b10); //Result 1‘b1
assign c=(1‘bx===1‘bx);  //Result 1‘b1
assign d=(1‘b1&&(2‘b10>=2‘b01)); //Result 1‘b1
```

Comparison Operators	
`a === b`	a equals b, including x and z, **not synthesizable!**
`a !== b`	a not equal b, including x and z, **not synthesizable!**
`a == b`	a equals b, result can be x
`a != b`	a not equal b, result can be x

Bit-wise and Reduction Operators

- Examples of Bit-wise and Reduction Operators

```
wire [3:0] a,b,c,d;

//Bitwise operators
assign a=4‘b0010&4‘b1110; //Result 4‘b0010
assign b=4‘b0010|4‘b1110; //Result 4‘b1110
assign c=4‘b0010^4‘b1110; //Result 4‘b1100
assign d=~4‘b0010;        //Result 4‘b1101
```

```
wire a,b,c,d;

//Reduction operators
assign a=&4‘b0010; //Result 1‘b0
assign b=|4‘b0010; //Result 1‘b1
assign c=^4‘b0010; //Result 1‘b1
assign d=^~4‘b0010; //Result 1‘b0
```

~	
0	1
1	0
x	x

&	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

|	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

^	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

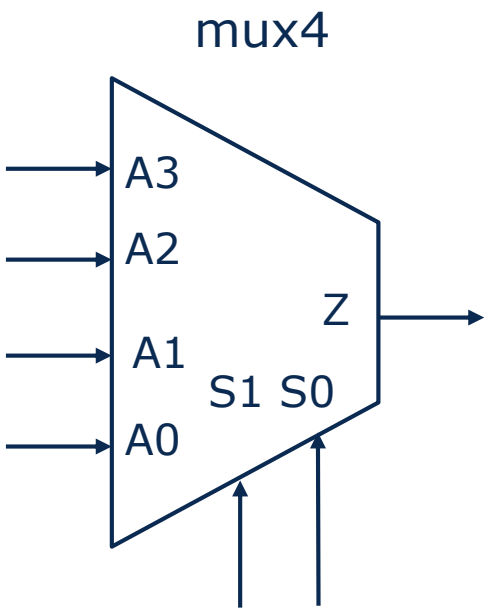
^~	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

Shift Operators

- Examples of Shift Operators

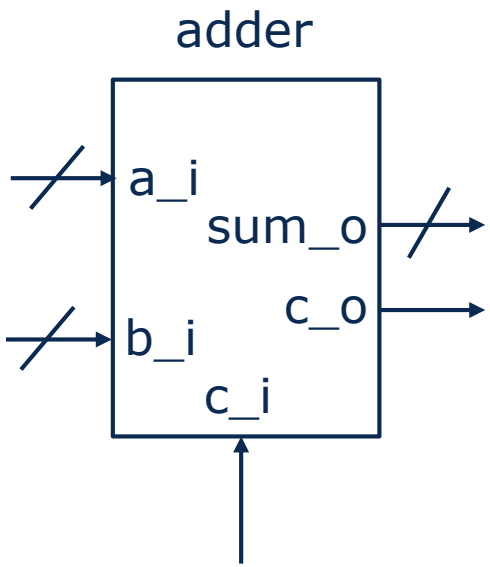
```
wire [3:0] a,b,c,d;

//Shift operators
assign a=4‘b0010 << 1; //Result 4‘b0100
assign b=4‘b0010 << 2; //Result 4‘b1000
assign c=4‘b0010 >> 1; //Result 4‘b0001
assign d=4‘b0010 >> 2; //Result 4‘b0000
```

Example: Multiplexer




```
//4-to-1 Multiplexer
module mux4 (A3,A2,A1,A0,Z,S1,S0);
   input A3,A2,A1,A0;
   input S1,S0;
   output Z;

   wire s32,s10,s3210;

   //MUX logic
   assign s10   =(S0) ? A1  : A0 ;
   assign s32   =(S0) ? A3  : A2 ;
   assign s3210 =(S1) ? s32 : s10 ;

   //output assignment
   assign Z=s3210;

endmodule
```

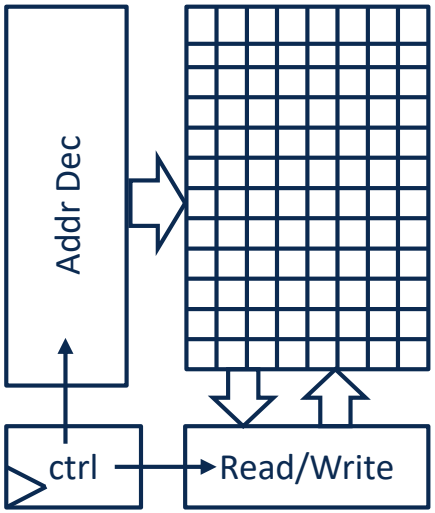
Example: Adder



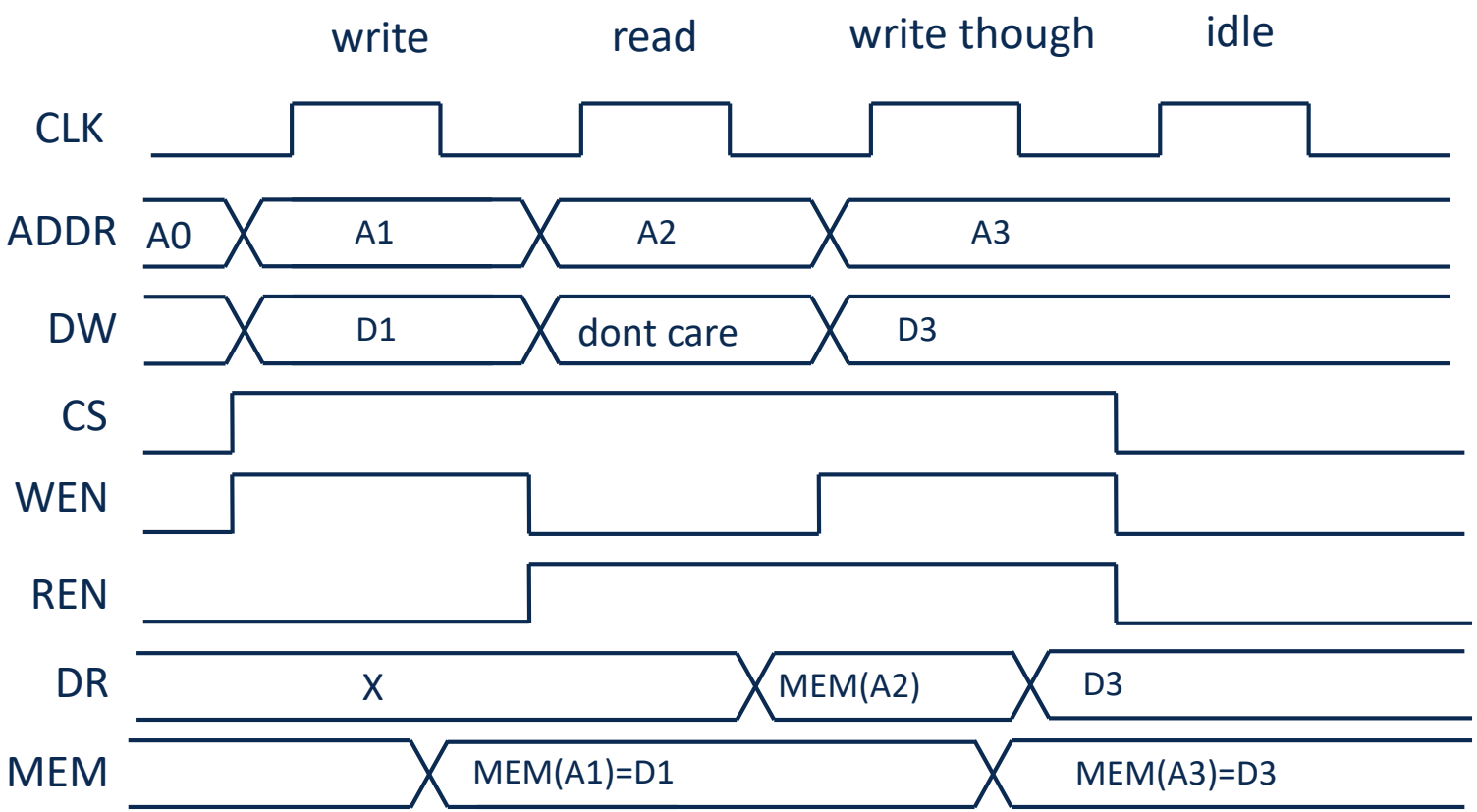

```
//Adder
module adder (sum_o, c_o, c_i, a_i, b_i) ;
   parameter C_DWIDTH=4;

   input [C_DWIDTH-1:0]  a_i, b_i;
   input c_i;
   output [C_DWIDTH-1:0] sum_o;
   output c_o;

   assign {c_o, sum_o} = a_i + b_i + c_i;
endmodule
```

Example: Single-Port SRAM Makro Block

```
module HM_1P_core_1cr (CLK_I,ADDR_I,DW_I,WE_I,RE_I,
CS_I,DR_O);

   parameter P_ADDR_WIDTH=8;
   parameter P_DATA_WIDTH=128;

   input CLK_I, WE_I, RE_I, CS_I;
   input [P_ADDR_WIDTH-1:0] ADDR_I;
   input [P_DATA_WIDTH-1:0] DW_I;
   output [P_DATA_WIDTH-1:0] DR_O;

   reg [P_DATA_WIDTH-1:0]    mem [0:2**P_ADDR_WIDTH-1];
   reg [P_DATA_WIDTH-1:0]    dr_r;

   always @(posedge CLK_I) begin
      if(CS_I==1'b1 && WE_I==1'b1) begin
         mem[ADDR_I] <= DW_I;    //write
      end
      if(CS_I==1'b1 && RE_I==1'b1) begin
        if (WE_I==1‘b1) dr_r<=DW_I;      //write through
        else            dr_r<=mem[ADDR_I];      //read
        end
   end
   assign DR_O=dr_r;
endmodule
```



Example: Single-Port SRAM Makro Block – Timing Diagram



Verilog – Procedures

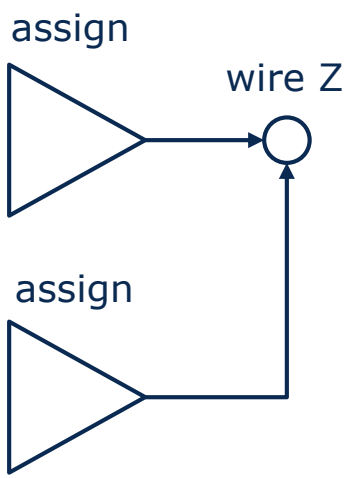
Procedural Assignment

- Continuous Assignment (assign)
  - → drives a net (wire) permanently
  - → Value changes as soon as the „inputs" of the assignment change

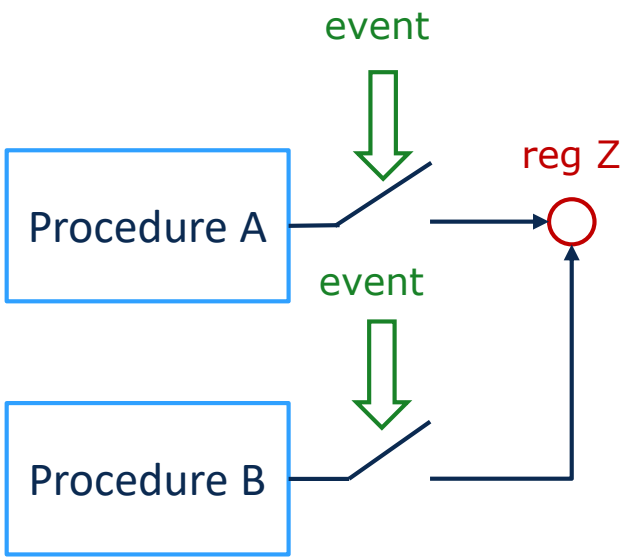
- Procedural Assignment
  - → writes a value in a Register data type (e.g. reg, integer)
  - → Register data type holds the value until the next procedural access
  - → Assignments are triggered by the control flow blocks (procedures)
    - initial
    - always
    - task
    - function
- Procedures can run in parallel and concurrently

Continuous and Procedural Assignments

Continuous Assignment

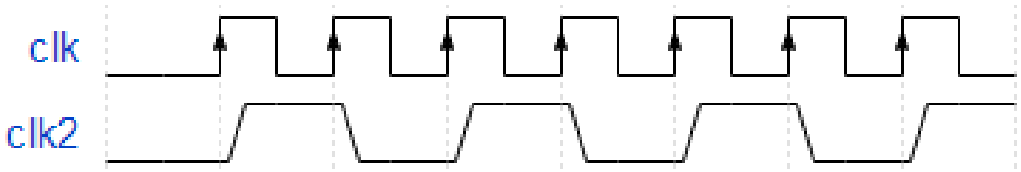


Procedural Assignment



Example: Procedural Assignments

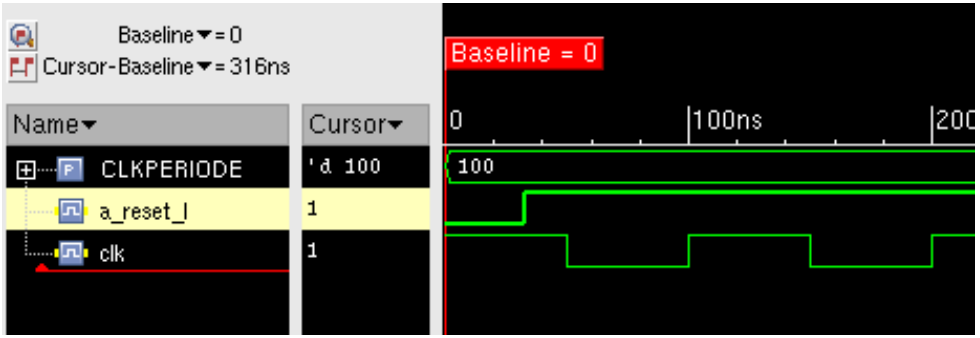
```
`timescale 1ns/1ps
module clock_div ();
   parameter PERIOD=10
   reg clk, clk2;
   initial begin
      clk=0;
      clk2=0;
   end

   always #PERIOD/2 clk=~clk;

   always @(posedge clk) begin
      clk2<=~clk2;
   end
endmodule
```



initial Procedure

- **initial**
  - Execution at the beginning of the simulation at time step 0.
  - Typical Application
    - Initializing registers
    - Start stimuli (waveforms) in testbench environments
  - **Not synthesizable!**

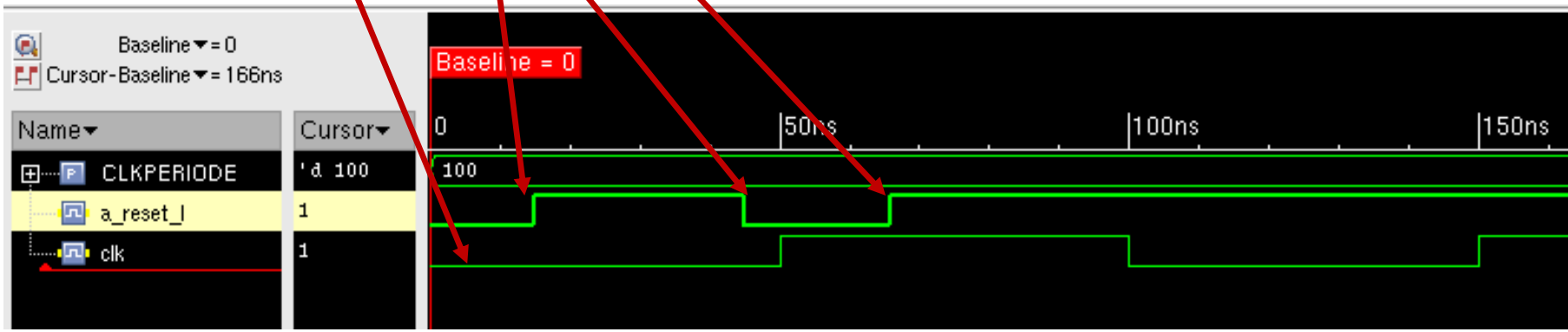
```
//Example Initial procedure
reg clk;
reg a_reset_l;
initial clk = 1'b1;

always #(CLKPERIODE/2) clk = !clk;

initial begin
    a_reset_l = 1'b0;
    #33
    a_reset_l = 1'b1;
end
```



Concurrent initial Procedures

```
reg clk;
reg a_reset_l;
initial clk = 1'b1;
initial clk = 1'b0;
always #(CLKPERIODE/2) clk = !clk;
initial begin
    #33 a_reset_l = 1'b0;
    a_reset_l = 1'b1;
    #33 a_reset_l =1‘b1;
end
initial begin
     a_reset_l = 1'b0;
     #15 a_reset_l = 1'b1;
     #30 a_reset_l = 1'b0;
end
```

- Concurrent execution of procedures is possible
- No x values as in continuous multiple assignment



Initialization of Register data type

- Initialization of Register data type is possible with declaration
- Easier for testbenches
- **Not synthesizable!**

```
//Example initialization of Register data type
reg clk=1‘b1;
real a=3.245;
integer i=1;
```

always Procedure

```
//Always Block
always @(…) begin
…
end
```

- Permanent Execution
- Useful application by means of „timing control" statements, e.g.
  - Delayed assignment
  - Event Control
  - Sensitivity Lists
- Suitable for modelling and describing :
  - Sequential Logic
  - Combinational Logic

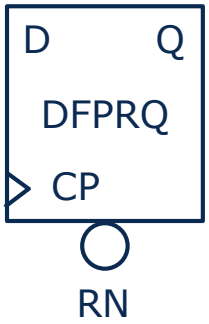
Event Control

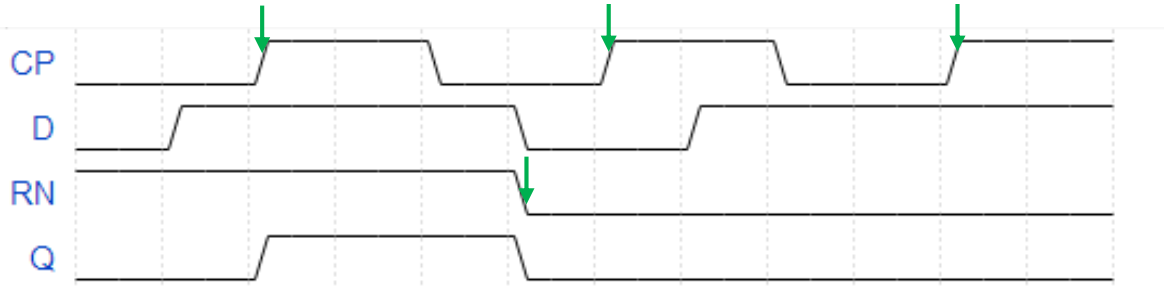
- Execution of always procedures controlled by „event control statements" @
  - @ <identifier> → signal change (rising or falling edge)
  - @ posedge → rising edge
  - @ negedge → falling edge
- Combination of events with ‚OR' is possible

```
// Example

//No Event Control
always #(CLKPERIODE/2) clk = !clk;  //permenant Execution

always @(b) a=b;            //Assignment triggered by b

always @(posedge clk) q <= d;       //rising Clk edge

always @(negedge clk) q <= d;       //falling Clk edge
```

Example: Flip-Flop with Asynchronous Reset

```
module DFPRQ (D,CP,Q,RN);

input D,CP,RN;
output Q;
reg q_reg;

always @(posedge CP or negedge RN) begin
   if (RN==1‘b0) begin
      q_reg<=1‘b0;
   end
   else begin
      q_reg<=D;
   end
end
assign Q=q_reg;

endmodule
```





Sensitivity List

- A combinational always block is triggered by a sensitivity list
- Coding Guidelines:
  - All signals on Right-Hand Side in assignments **must be included**!
  - Left-Hand Side should not be included → No „self-triggering"

```
// Example Sensitivity List (XOR)

//full Sensitivity List
always @(a or b) begin
   c1=a^b;
end
//missing Signal in Sensitivity List
always @( a ) begin
   c2=a^b;
end
```

Mismatch between RTL simulation and synthesis results

„Blocking Assignments“

- Blocking assignments (=) will be **immediately** executed before the next statement in the procedure
- Interrupt (blocks) the procedural process

```
// Example Blocking Assignment
initial begin
        a=0;
        b=0;
        a= #2 1;
        b= #4 1;
        a= #2 0;
        b=0;
end
```

- Blocking assignments are suitable for describing signal flow (Waveforms, Stimuli)

„Non-Blocking Assignments“

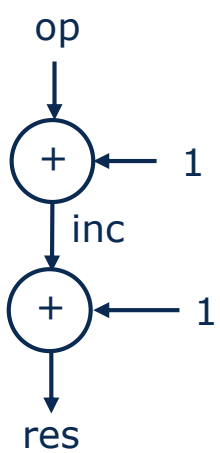
- Non-Blocking assignments (<=) do not interrupt the procedural process
- Execution by the simulator is done in 2 steps:
  1. Evaluation of the right side, **assigning** this result to the left side
  2. **Performing** the assignment at the time of Event Control Statements

```
// Example Non-Blocking Assignment
initial begin
        a <= 0;
        b <= 0;
        a <= #2 1;
        b <= #4 1;
        a <= #2 0;
        b <= 0;
end
```

- Non-Blocking assignments are suitable for describing sequential logic blocks whose timing is defined by Event Control Statements (e.g. @(posedge clk))

Example 1 of Blocking and Non-Blocking Assignments

Combinational Logic



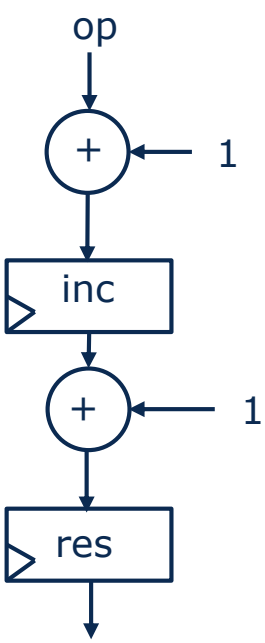

```
// Blocking Assignment
always @(op) begin
        inc1=op+1;
        res1=inc1+1;
end
```

```
// Non-Blocking Assignment
always @(op) begin
        inc2<=op+1;
        res2<=inc2+1;
end
```

- Using **Blocking Assignments** to describe **combinational logic**!

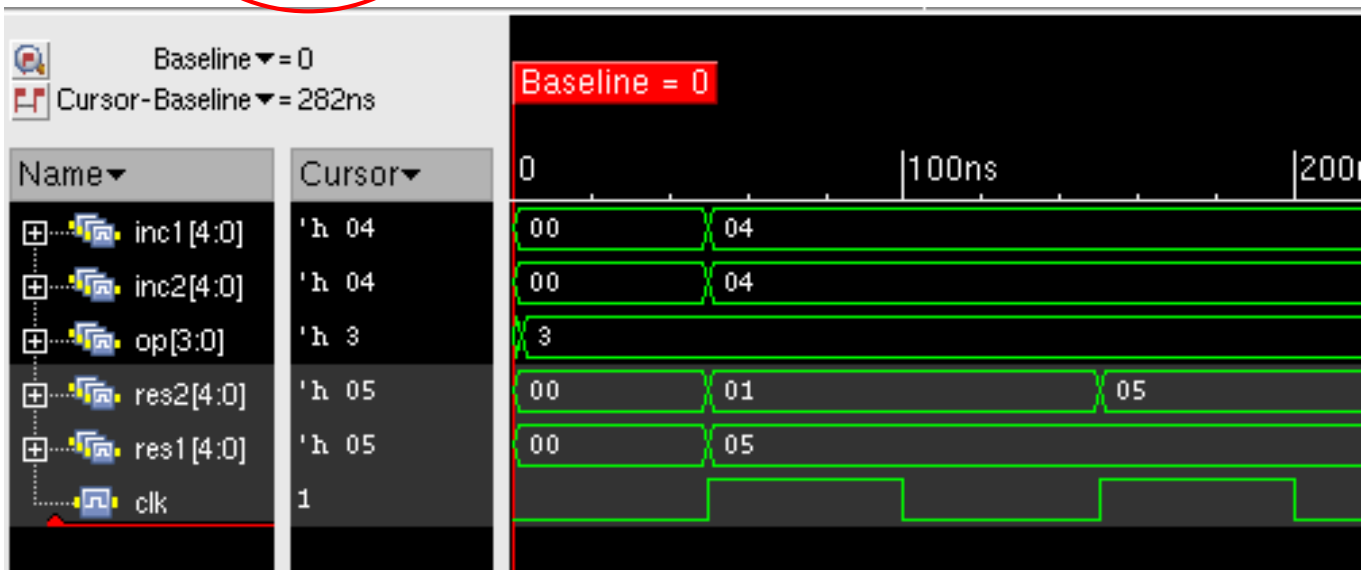
Example 2 of Blocking and Non-Blocking Assignments

Sequential Logic




```
// Blocking Assignment
always @(posedge clk) begin
    inc1=op+1;
    res1=inc1+1;
end
```

```
// Non-Blocking Assignment
always @(posedge clk) begin
    inc2<=op+1;
    res2<=inc2+1;
end
```

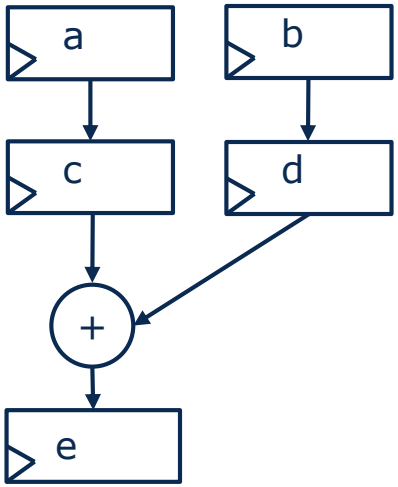


- Using **Non-Blocking Assignments** to describe **sequential logic**!

Example 3 of Blocking and Non-Blocking Assignments

- Only one assignment per always block
- Is using blocking assignments same as using non-blocking assignments?

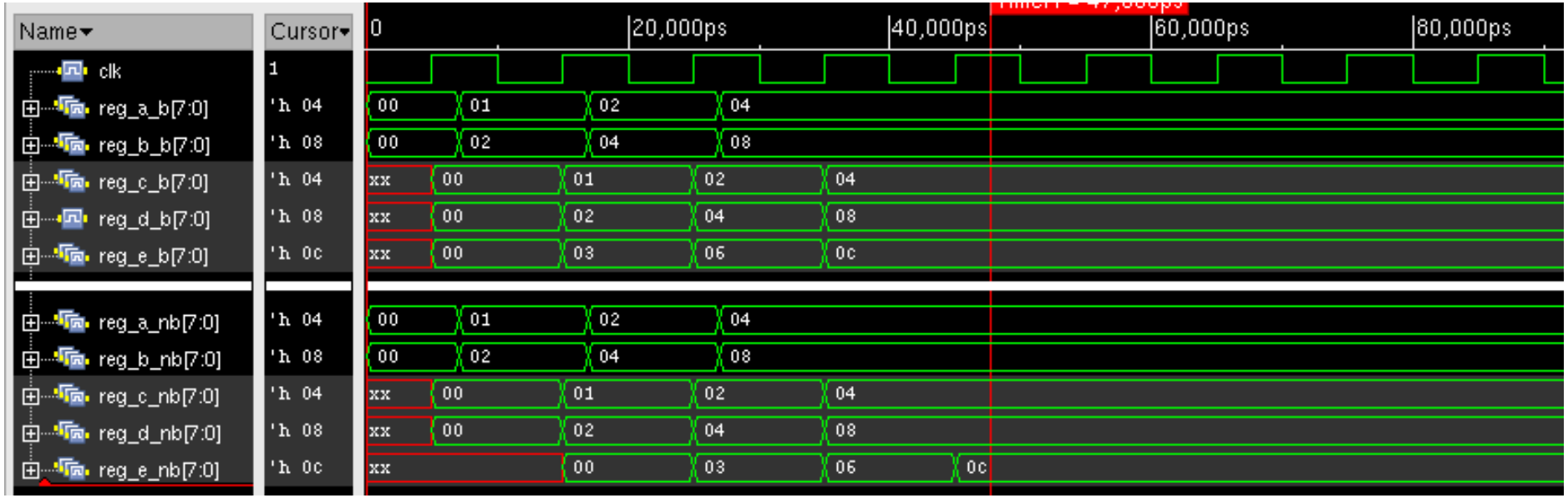
- Using many of these blocks (e.g. instances of a register)
- Example:




```
always @(posedge clk) begin
        reg_c_b=reg_a_b;
end
always @(posedge clk) begin
        reg_d_b=reg_b_b;
end
always @(posedge clk) begin
        reg_e_b=reg_c_b+reg_d_b;
end
```

```
always @(posedge clk) begin
        reg_c_nb<=reg_a_nb;
end
always @(posedge clk) begin
        reg_d_nb<=reg_b_nb;
end
always @(posedge clk) begin
        reg_e_nb<=reg_c_nb+reg_d_nb;
end
```

Example 3 of Blocking and Non-Blocking Assignments



- Using **Non-Blocking Assignments** to describe **sequential logic**!
- Only **one assignment** per always block as well

Simulation of Combinational Logic

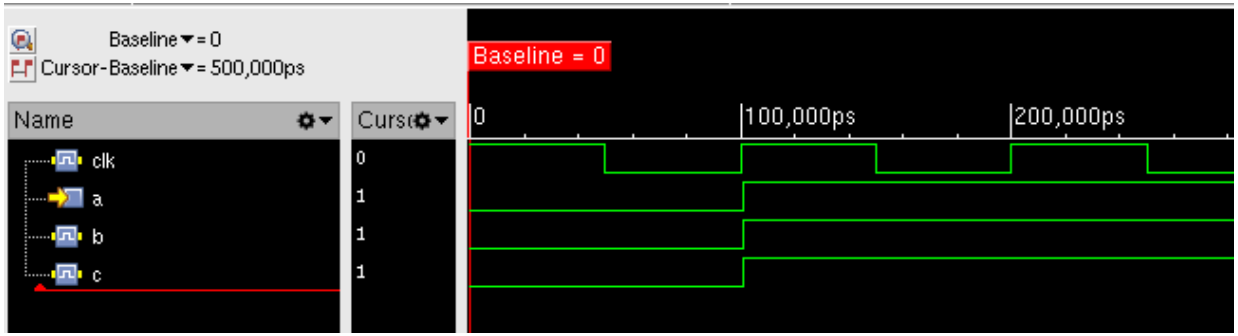
- **Delta Cycle:** concept in HDL simulations to arrange events, which event will take place with time interval of 0 (zero)
- **Time step:** advancing simulation time

```
input clk;
input a;
reg b,c;

always @(a)
begin
    b = a;
end

always @(b)
begin
    c = b;
end
```

Delta cycle:

```
b <- a
c <- b
```

Next time step: *No assignment*



→ **immediate** signal assignment

Simulation of Sequential Logic

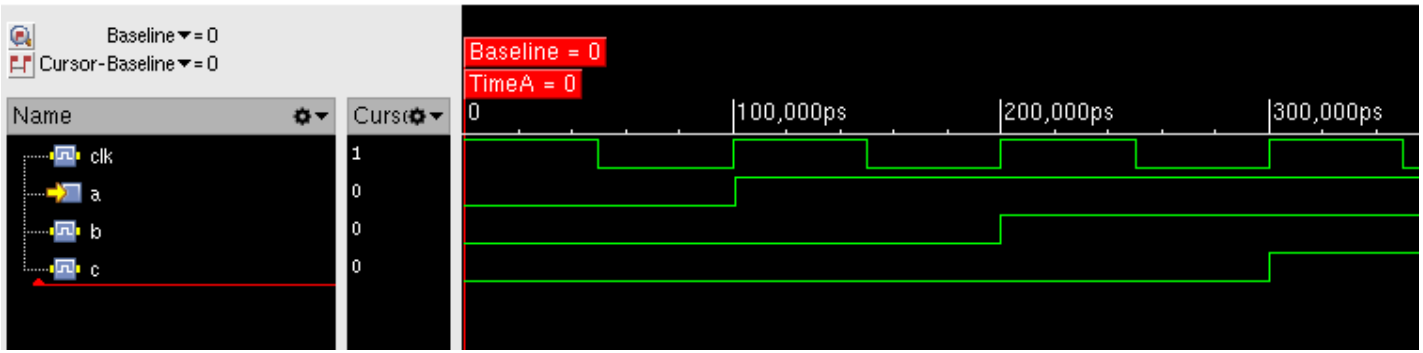
```
input clk;
input a;
reg b,c;

always @(posedge clk)
begin
    b <= a;
end

always @(posedge clk)
begin
    c <= b;
end
```

Delta cycle:

```
tmp_b <- a
tmp_c <- b
```

Next time step

```
b <- tmp_b
c <- tmp_c
```



→ Signal assignment when advancing time step

Control Statements

- Control statements can be used inside procedural assignments for conditional execution of blocks
  - if/else , case

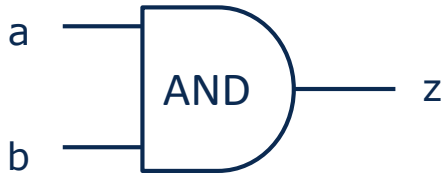
```
// Example IF/ELSE
wire [1:0] a;
always @(a) begin
  if (a==0) begin
      b=1;
  end
  else if (a==1) begin
      b=2;
  end
  else begin
      b=3;
  end
end
```

```
// Example case 1
wire [1:0] a;
always @(a) begin

case (a)
    2‘b00:  b=1;
    2‘b01:  b=2;
    2‘b10:  b=3;
    2‘b11:  b=3;
endcase
```

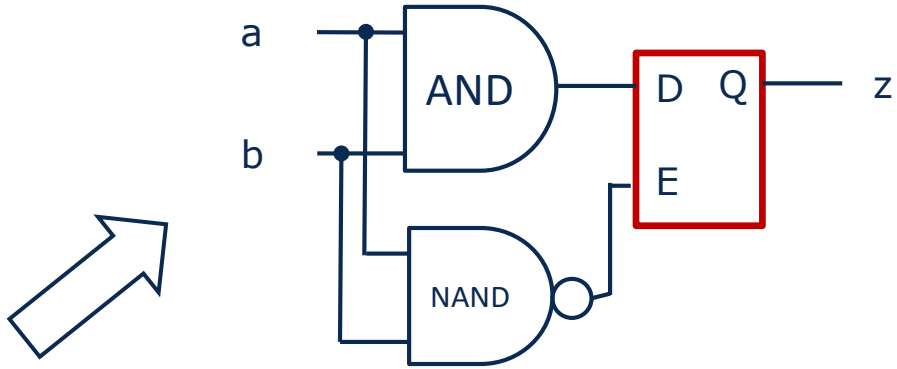
```
// Example case 2
wire [1:0] a;
always @(a) begin

case (a)
    2‘b00:  b=1;
    2‘b01:  b=2;
    default: b=3;
endcase
```

Unwanted Description of Latches in Combinational Logic

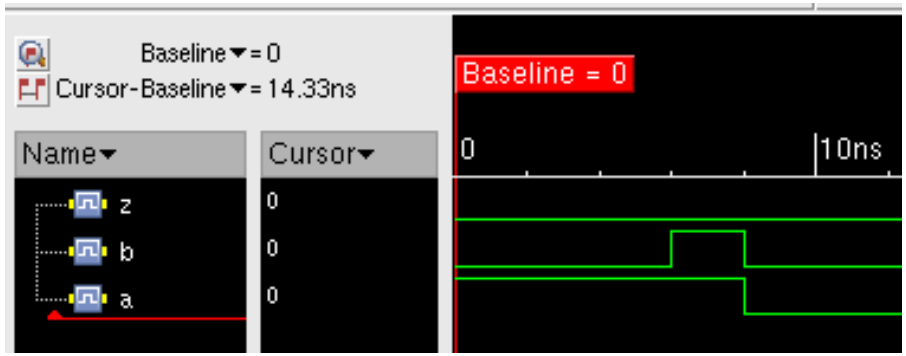
- For control statements describing combinational logic, Default assignments are mandatory!
- Otherwise, sequential elements (latches) are described




```
// Example IF/ELSE
// without default
always @(a or b) begin
  if ((a==0)||(b==0)) begin
      z=0;
  end
end
```

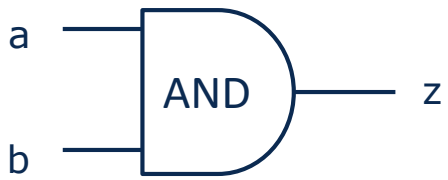


Latch is added!



Correct Default Assignments

- correct description versions:




```
// Example AND case 1
always @(a or b) begin
  case {a,b}
     2'b00:  z=0;
     2'b01:  z=0;
     2'b10:  z=0;
     2'b11:  z=1;
  endcase
 end
```

```
// Example AND case 2
always @(a or b) begin
  case {a,b}
     2'b11:     z=1;
     default:   z=0;
   endcase
 end
```

```
// Example AND IF/ELSE 1
always @(a or b) begin
  if ((a==0)||(b==0)) begin
      z=0;
  end
  else begin
     z=1;
  end
end
```

```
// Example AND IF/ELSE 2
always @(a or b) begin
  z=1;
  if ((a==0)||(b==0)) begin
      z=0;
  end
end
```

Loop Statements

- Blocks can be repeated in loops inside procedural assignments
  - forever → infinite loop
  - repeat → loop for a certain number of times
  - while → loop as long as condition is true
  - for → classic for loop

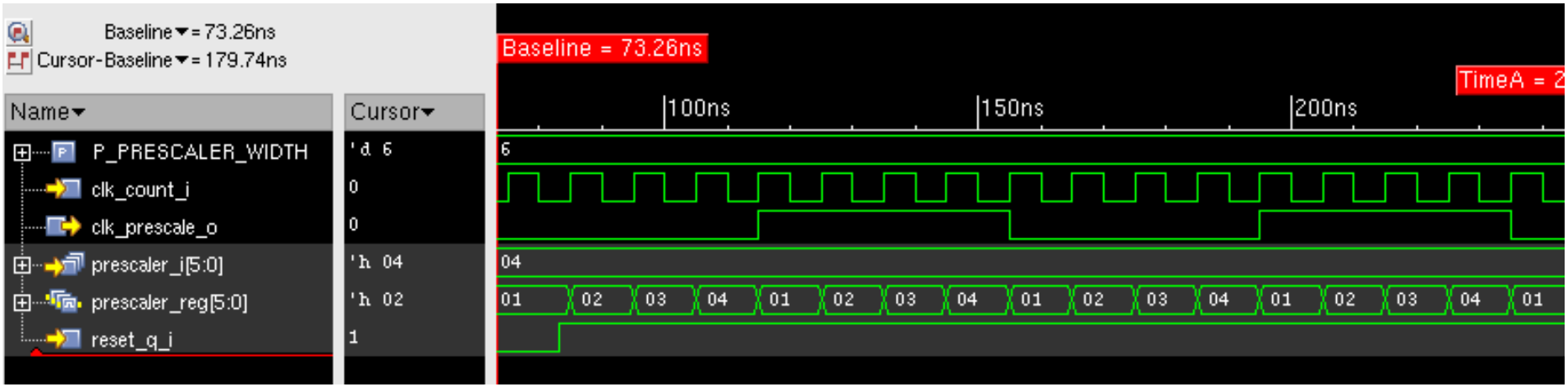
Example: Combinational Logic Thermometer Decoder

```
module thermo_decoder (bin_i, thermo_o);

   parameter BINBITS=5;
   parameter THERMOBITS=32;

   input   [BINBITS-1:0] bin_i;
   output  [THERMOBITS-1:0] thermo_o;

   reg [THERMOBITS-1:0] thermo_o;
   //thermo decoder block
   integer i;

   always @(bin_i) begin
      for (i=0;i<THERMOBITS;i=i+1) begin
         if (bin_i[BINBITS-1:0]>i) thermo_o[i]=1'b1;
         else thermo_o[i]=1'b0;
      end
   end
endmodule
```

Beispiel: Sequential Logic Frequency Divider

```
module clk_prescaler (reset_q_i, clk_count_i, prescaler_i, clk_prescale_o);

parameter P_PRESCALER_WIDTH=6;
input     reset_q_i;
input     clk_count_i;
input     [P_PRESCALER_WIDTH-1:0]       prescaler_i;
output    clk_prescale_o;
reg [P_PRESCALER_WIDTH-1:0] prescaler_reg;
reg       clk_prescale_reg;

always @(posedge clk_count_i or negedge reset_q_i) begin
   if (reset_q_i == 1'b0) begin
      prescaler_reg<=1;
      clk_prescale_reg<=0;
   end
   else begin
      if (prescaler_reg==prescaler_i) begin
          clk_prescale_reg<=~clk_prescale_reg;
          prescaler_reg<=1;
      end
      else begin
          prescaler_reg<=prescaler_reg+1;
      end
   end
end
assign clk_prescale_o=clk_prescale_reg;
endmodule
```

Example: Sequential Logic Frequency Divider Waveform



Summary: Modelling of Combinational Logic

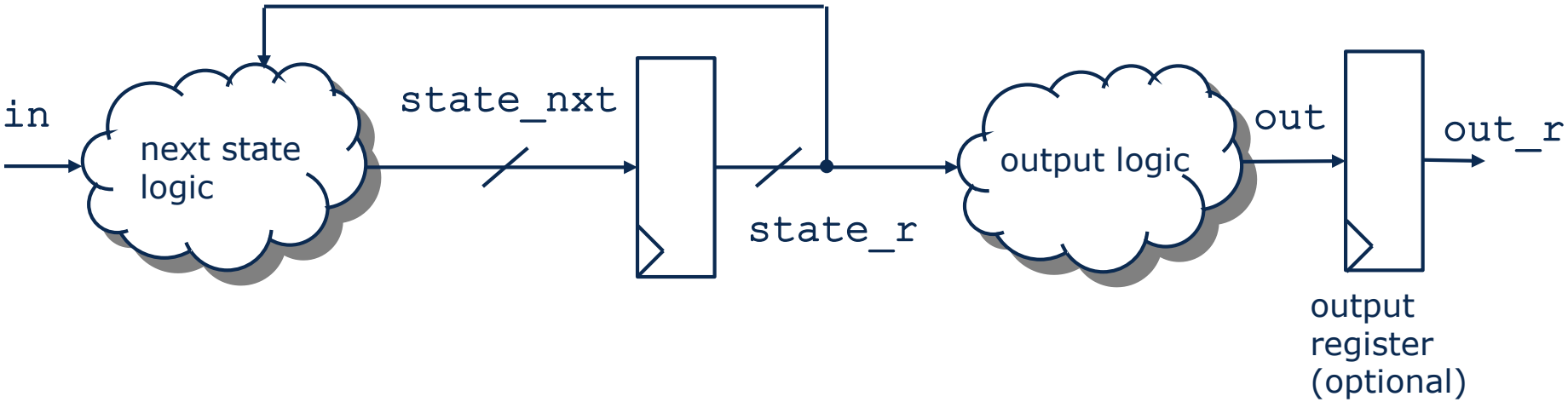
- Using always blocks is possible
- Full Sensitivity List is necessary
- Using Blocking Assignments (=)
- Default assignments or full coding of Control-Statements (if/else, case) to avoid unwanted latches
- assigning combinational signals only **in always block** (no concurrent assignments)

Summary: Modelling of Sequential Logic

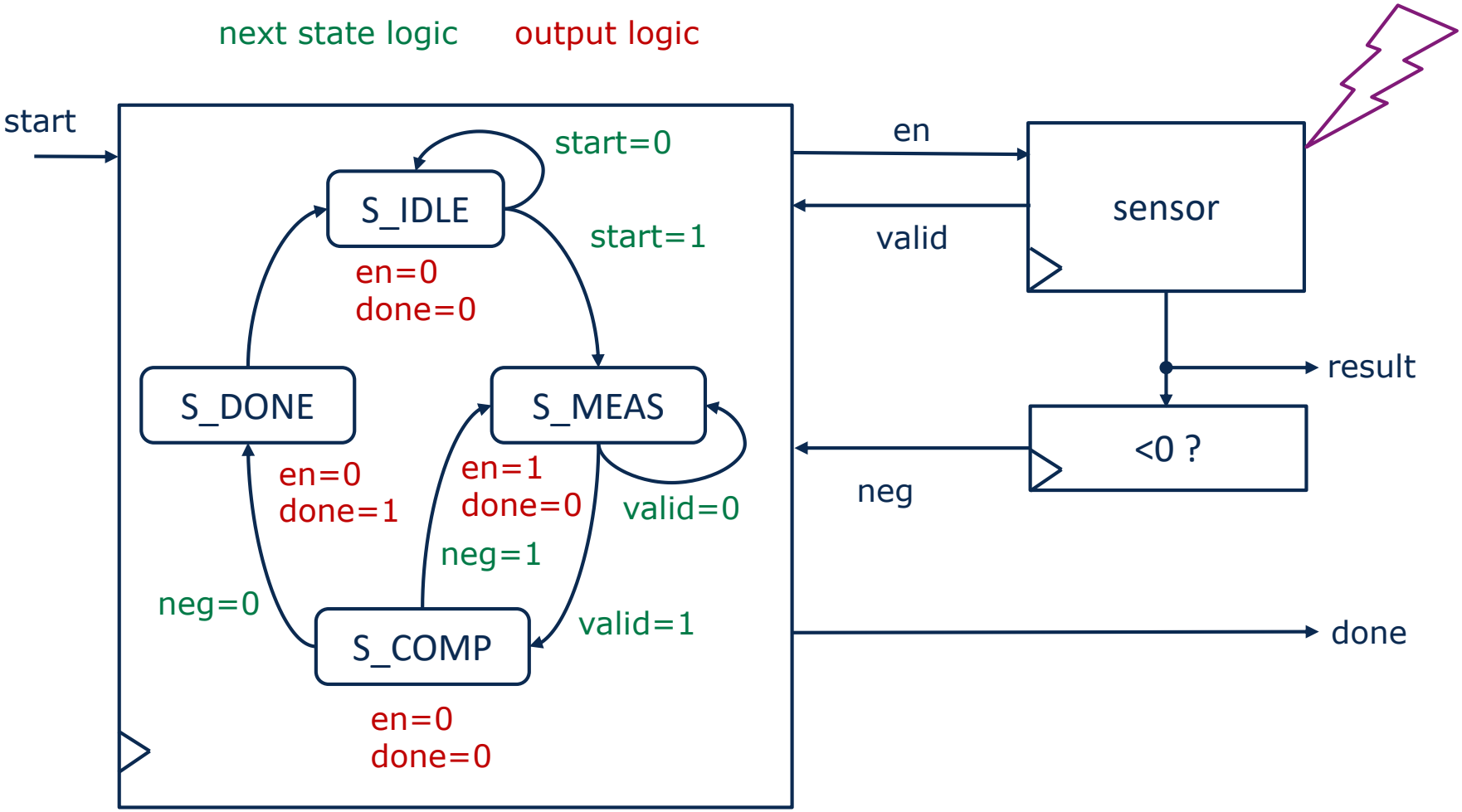
- Using always blocks is mandatory
- Edge-sensitive Event Control Statement as a trigger (e.g. @(posedge clk))
- Asynchronous set/reset is possible
- Always blocks with asynchronous set/reset must contain all register signals in reset block and functional block
- Using non-blocking assignments (<=)
- Writing signals only in always block (No concurrent assignments)

Description of Finite State Machine (FSM)

- FSM: Finite State Machine
- Modelled as Moore machine
  - State transition function: state_nxt=G(state_r,in)
  - Output function: out=F(state_r)



Example: FSM



Example: FSM Sourcecode Header

```
// Company            :   tud
// Author             :   hoeppner
// E-Mail             :   <email>
//
// Filename           :   fsm.v
// Project Name       :   p_ice
// Subproject Name    :   s_verilog
// Description        :   <short description>
//
// Create Date        :   Wed Jan 22 10:16:59 2014
// Last Change        :   $Date$
// by                 :   $Author$

//-----------------------------------------------------------------
module fsm (reset_q_i,clk_i,start_i,valid_i,neg_i,en_o,done_o);

input     reset_q_i;
input     clk_i;
input     start_i;
input     valid_i;
input     neg_i;
output    en_o;
output    done_o;
…
```

Example: FSM Sourcecode Parameters and Signals

```
…
parameter S_IDLE=0;
parameter S_MEAS=1;
parameter S_COMP=2;
parameter S_DONE=3;

//registers
reg [1:0] state_r;

//comb signals
reg [1:0]        state_nxt;
reg              en;
reg              done;
…
```

Example: FSM Sourcecode State Transition Logic

```
//next state logic
always @(state_r or start_i or valid_i or neg_i)
begin
case (state_r)
          S_IDLE: begin
                    if (start_i==1'b1) begin
                              state_nxt=S_MEAS;
                    end
                    else begin
                              state_nxt=S_IDLE;
                    end
          end
          S_MEAS: begin
                    if (valid_i==1'b1) begin
                              state_nxt=S_COMP;
                    end
                    else begin
                              state_nxt=S_MEAS;
                    end
          end
          …
```

```
…
S_COMP: begin
          if (neg_i==1'b1) begin
                    state_nxt=S_MEAS;
          end
          else begin
                    state_nxt=S_DONE;
          end
end
S_DONE: begin
          state_nxt=S_IDLE;
end
default: begin
          state_nxt=S_IDLE;
end
endcase
end
…
```

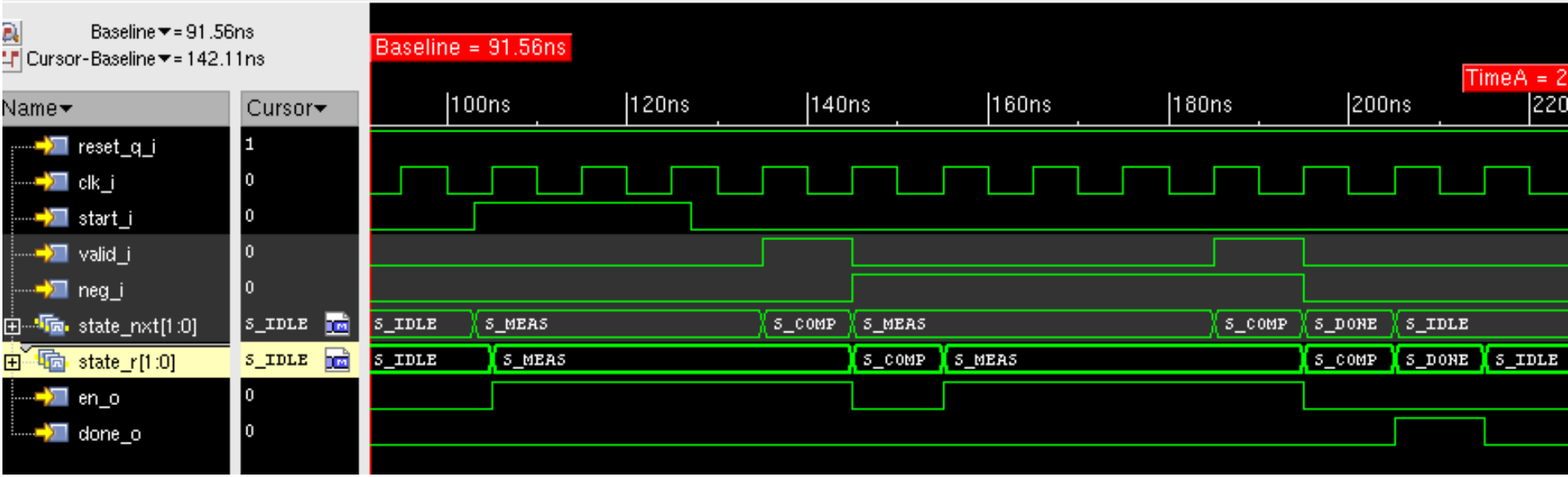
Example: FSM Sourcecode State Register

```
…
//state register
always @(posedge clk_i or negedge reset_q_i)
begin
        if (reset_q_i==1'b0) begin
                state_r<=S_IDLE;
        end
        else begin
                state_r<=state_nxt;
        end
end
…
```

Example: FSM Sourcecode Output Logic

```
…
//output logic
always @(state_r) begin
        //default assignment
        en=0;
        done=0;
        if (state_r==S_MEAS) begin
                en=1;
        end
        if (state_r==S_DONE) begin
                done=1;
        end
end

//output assignment
assign en_o       = en;
assign done_o     = done;

endmodule
```

Example: FSM Waveform



Guidelines to describe FSMs

- Use parameters to define the states
- Seperate combinational from sequential circuit blocks
  - Register
  - State transition function
  - Output function
- Do not describe any state transition functions in the sequential part
  - exception: synchronous Reset
- Separate assignment of output signals to the„outputs“

Verilog – Hierarchical Design

Hierarchical Design

- Verilog provides the ability to partition the design and reuse design elements
  - Module
  - Tasks
  - Functions
- Integration of Code Blocks with File Includes (´include)

Instantiation of Modules

Declaration

```
module my_module (a, b, c);
 input a,b;
 output c;
…
endmodule
```

Instantiation

```
…
reg a0;
wire b0,c0,c1;

my_module my_module_i0 (
        .a(a0),
        .b(b0),
        .c(c0)
);

my_module my_module_i1 (
        .a(a0),
        .b(c0),
        .c(c1)
);
```

- Modules can be instantiated
- Inputs can be driven as „reg" and „wire" data types
- Outputs are assigned as „wire" data type (Continuous Assignment)

Parameterization during Instantiation

Declaration

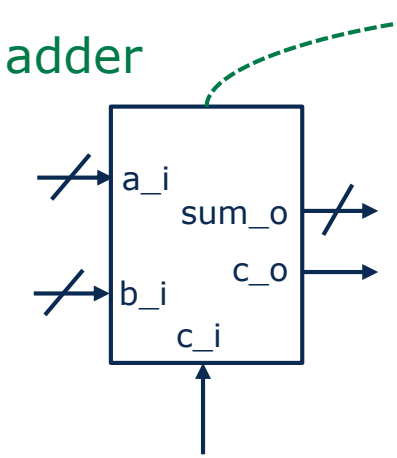
```
module my_module (a, b, c);

 parameter P0=1;
 parameter P1=2;

 input a,b;
 output c;
…
endmodule
```

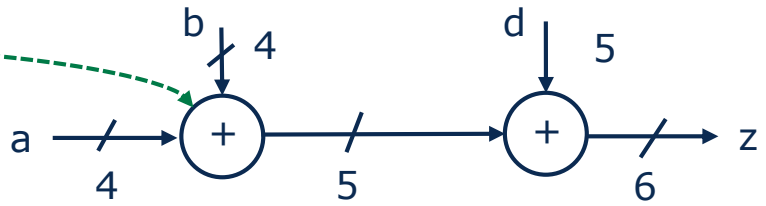
Instantiation

```
…
reg a0;
wire b0,c0,c1;

my_module #(.P0(2),.P1(3))
 my_module_i0 (
        .a(a0),
        .b(b0),
        .c(c0)
);
```

- Parameters of modules can be overwritten during instantiation
- Different instances can be parameterized individually

Example: Instantiation and Parameterization Adder Data Path



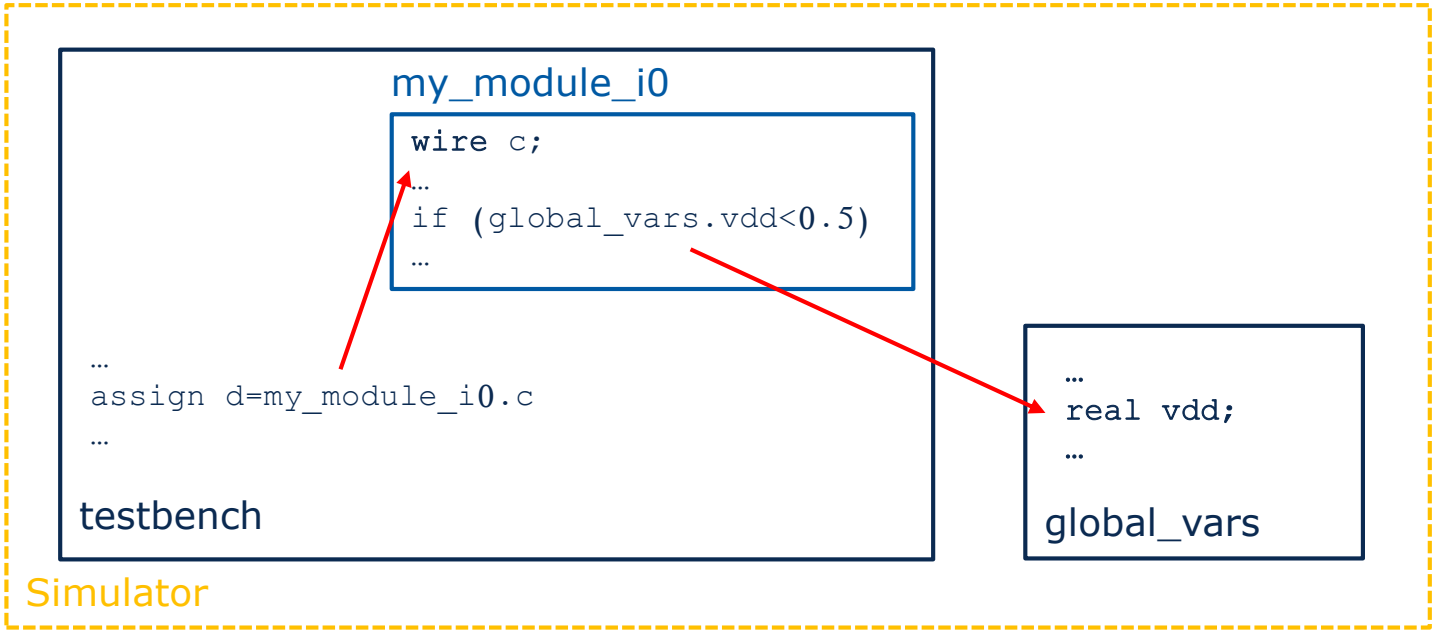



```
//Adder
module adder (sum_o, c_o, c_i, a_i, b_i) ;
   parameter C_DWIDTH=4;

   input [C_DWIDTH-1:0]  a_i, b_i;
   input c_i;
   output [C_DWIDTH-1:0] sum_o;
   output c_o;

   assign {c_o, sum_o} = a_i + b_i + c_i;
endmodule
```

```
…
wire [3:0] a,b,s0;
wire c0,c1;
wire [4:0] d,s1;
wire [5:0] z;

adder #(.C_DWIDTH(4)) adder_i0 (
   .sum_o(s0),
   .c_o(c0),
   .c_i(1'b0),
   .a_i(a),
   .b_i(b)
) ;
adder #(.C_DWIDTH(5)) adder_i1 (
   .sum_o(s1),
   .c_o(c1),
   .c_i(1'b0),
   .a_i({c0,s0}),
   .b_i(d)
) ;
assign z={c1,s1};
```

Hierarchies and Global Signals

- Signals in Verilog are globally available
- Instance hierarchies are defined and seperated with „ . „ from the signal name.
- **Not synthesizable!**
- Application:
  - Monitoring of Signals in testbenches for debugging and verification
  - For global nets which are not signal pins (e.g. supply voltage)
- Be aware: Signals which are not runnung over the ports are not writable / readable from the outside, even in the implemented circuit!



Example: FlipFlop Model with Power-Shut-Off

Testbench

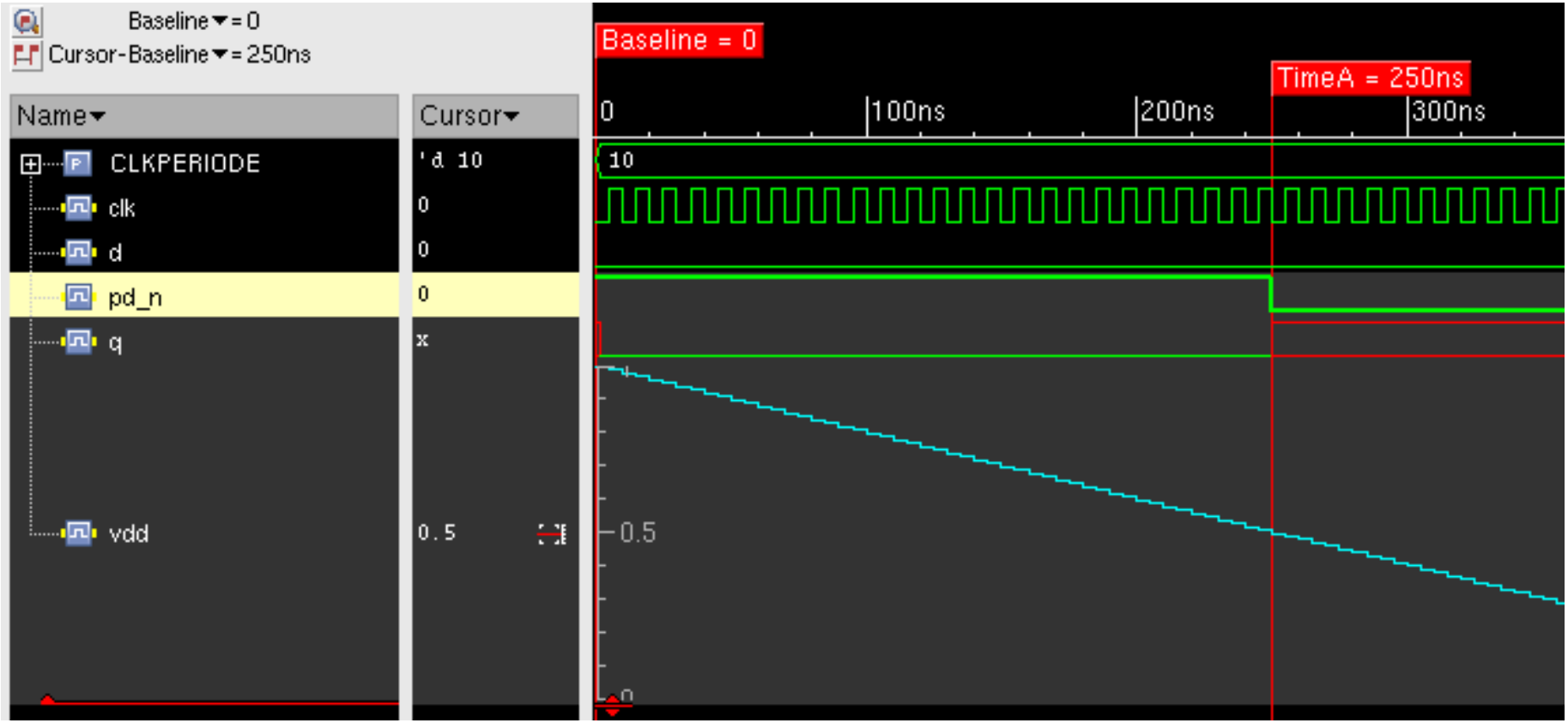
```
…
reg d,clk;
wire q;
wire pd_n;

reg_pd reg_i0 (
          .d(d),
          .cp(clk),
          .q(q)
);

//Observe internal power down signal
assign pd_n=reg_i0.pd_n;

//intialize FF without reset
initial #2 reg_i0.q_reg=$random;
```

Debug Probe

Full Instantiation

```
module reg_pd (d, cp, q);

input d, cp;
output q;

wire pd_n;

reg q_reg;

assign pd_n=(global_vars.vdd>0.4);

always @(posedge cp or negedge pd_n) begin
   if (pd_n==1‘b0) q_reg <=1‘bx;
   else q_reg<=d;
end

assign q=q_reg;
endmodule
```

Example: FlipFlop Model with Power-Shut-Off



Functions

- Functions can encapsulate complex assignments:
  - → Clear Code
  - → Reusability
- Functions have **a return value**.
- A function definition must include **an assignment** to the name of the function.
- Functions are executed in **one** time step.
- A function definition **cannot** contain **any timed execution** (Delays, Event Control Statements).
- Functions cannot call tasks.
- A function must have **at least one** input argument.
- A function definition **cannot** have any argument of type **output** or **inout**.

- → functions are suitable for describing combinational logic

Example Function

```
//Example Functions
…
//Definition
function [5:0] adder_func;
   input [3:0] a,b;
   input [4:0] d;
   begin
      adder_func=a+b+d;
   end
endfunction
…
//function call
assign result_1=adder_func(e,f,g);
…
always @(opa or opb or opd) begin
   result_2=adder_func(opa,opb,opd);
end
…
```

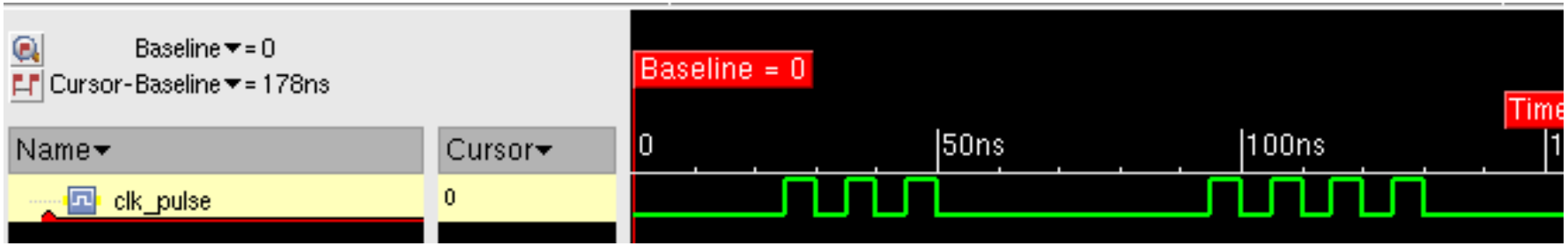
Tasks

- Tasks can encapsulate complex assignments, including time sequences:
  - → Clear Code
  - → Reusability
- Tasks can have **any number of inputs and outputs**.
- Tasks can contain **timed execution** (Delays, Event Control Statements).
- Tasks can call functions and other tasks.
- Tasks can access global variables.

- Tasks are suitable for describing combinatinal logic, sequential logic and stimuli.

Example: Task for Clock Pulse Generation

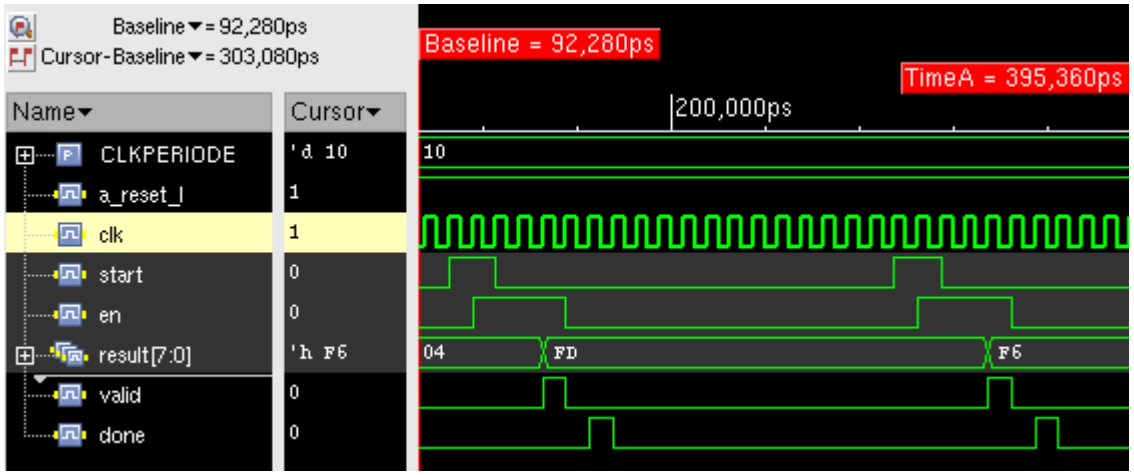
```
reg clk_pulse;
initial clk_pulse=1'b0;

task clk_pulses;
   input integer nr;
   integer i;
   begin
      for (i=0;i<nr;i=i+1) begin
         #(CLKPERIODE/2) clk_pulse=1'b1;
         #(CLKPERIODE/2) clk_pulse=1'b0;
      end
    end
endtask

initial begin
          #20 clk_pulses(3);
          #40 clk_pulses(4);
end
```



Example: Task to start a FSM

- Starting the FSM from example in slide 81

```
task run_meas;
   begin
      #(CLKPERIODE/2) start=1'b1;
      #(2*CLKPERIODE) start=1'b0;
      wait(done==1'b1);
      #(2.5*CLKPERIODE)
      $display("DONE");
   end
endtask

…

initial begin
         #100 run_meas;
         //do results processing
         #100 run_meas;
         //do results processing
end
```



File Includes

- Functions and tasks must be defined in the module where they will be used.
- File includes allow the use of predefined code fragments.

```
//Example File Include
module testbench ()

   `include "my_tasks.v"

   `include "my_functions.v"

   `include „stim_clk_gen.v"

   initial begin
   …
   //testcase code
   …

end
endmodule
```

```
//my_tasks.v
task my_task1;
…
endtask

task my_task2;
…
endtask
```

```
//my_functions.v
function my_func1;
…
endfunction

function my_func2;
…
endfunction
```

```
//stim_clk_gen.v
parameter CLKPERIODE = 10;
reg clk;
initial clk = 1'b0;
always #(CLKPERIODE/2) clk = !clk;
```

System Functions and Tasks

- Verilog have built-in system functions and tasks
- Here are some important ones:
  - $display
    - Formatted output to the command line
  - $fopen, $fwrite, $fflush, $fclose
    - Write to a file
  - $readmemh, $readmemb
    - Loading a memory from a text file
  - $random
    - Generating a random number
  - $dist_normal
    - Generation of a Gauss-distributed random number
  - $realtime
    - Return of the current simulation time as real
  - $finish
    - Stop the simulation

Example: Loading a memory in the simulation

mem_content.txt

```
12abc   34def   1dead 2bee1
```

```
module mem_init;

reg [19:0] memory [0:3];

initial $readmemh("mem_content.txt", memory);

integer i;
initial begin
   $display("rdata:");
   for (i=0; i < 4; i=i+1)   begin
     $display("%d:%h",i, memory[i]);
   end
end
endmodule
```

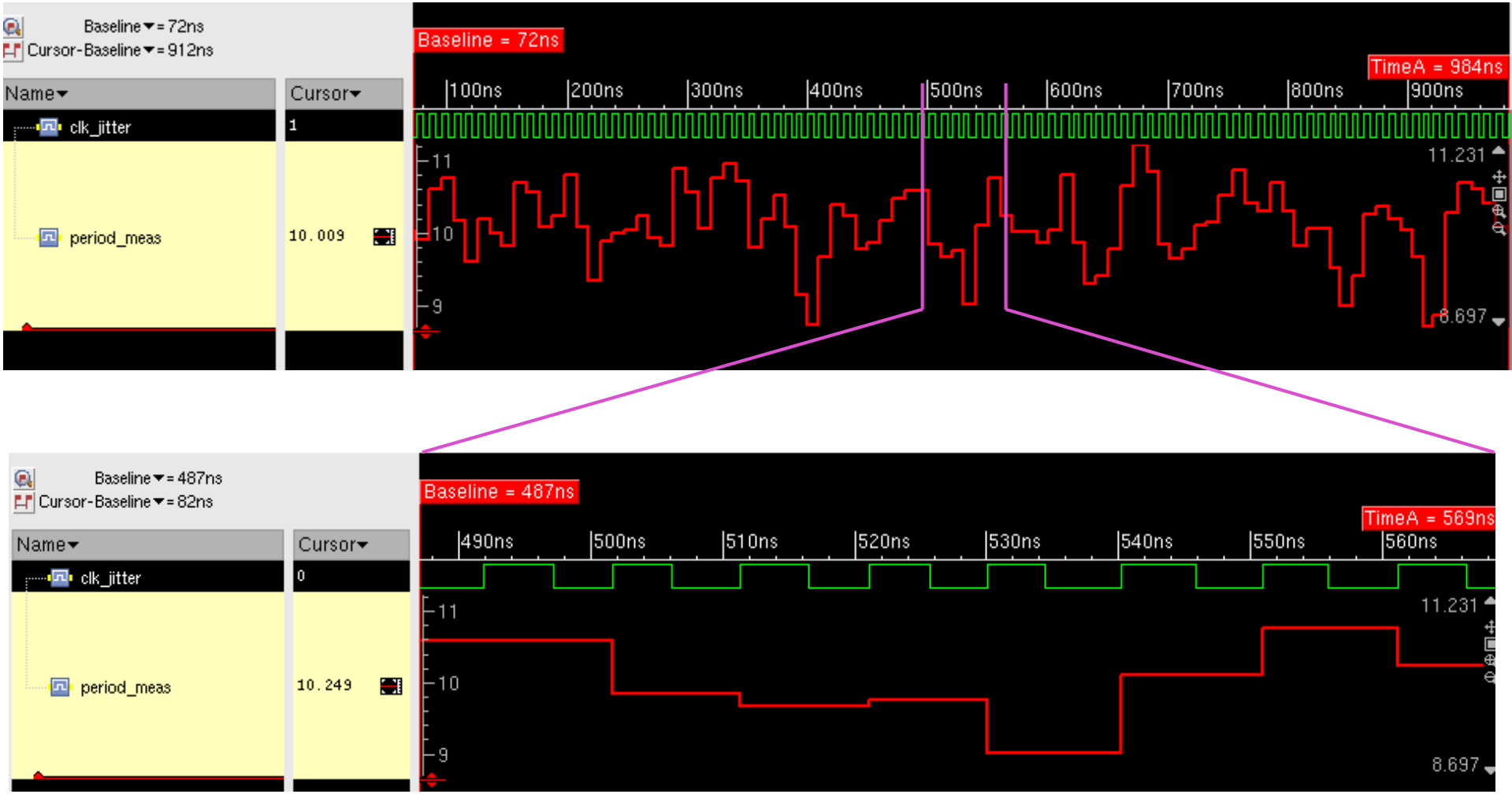
```
rdata:
0:12abc
1:34def
2:1dead
3:2bee1
```

simulation output

Example: Oscillator with Jitter

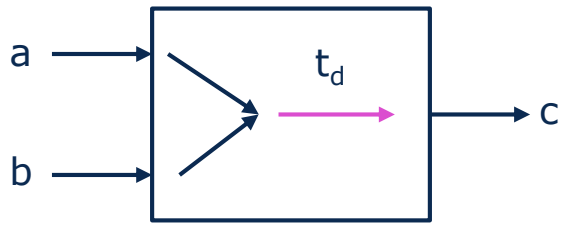
```
parameter NOISE_SEED=321;
parameter SIGMA_T=0.5;

reg clk_jitter=0;
real period_jitter=0;
real period=CLKPERIODE;
integer seed=NOISE_SEED, mean=0, stdev=1000000, random_value_int;

always @(period_jitter)  begin
           period=CLKPERIODE+period_jitter;
           if (period<=0) period=0.001;
  end
always @(posedge clk_jitter) begin
          random_value_int=$dist_normal(seed,mean,stdev);
          period_jitter=random_value_int/1000000.0*SIGMA_T*1.4142135;
end
always #(period/2)  clk_jitter=~clk_jitter;

real period_meas=0;
real prev_event_time=0;
real event_time=0;
always @(posedge clk_jitter) begin
          prev_event_time=event_time;
          event_time=$realtime;
          period_meas=event_time-prev_event_time;
end
```

Example: Oscillator with Jitter



Verilog – Modelling of Delays

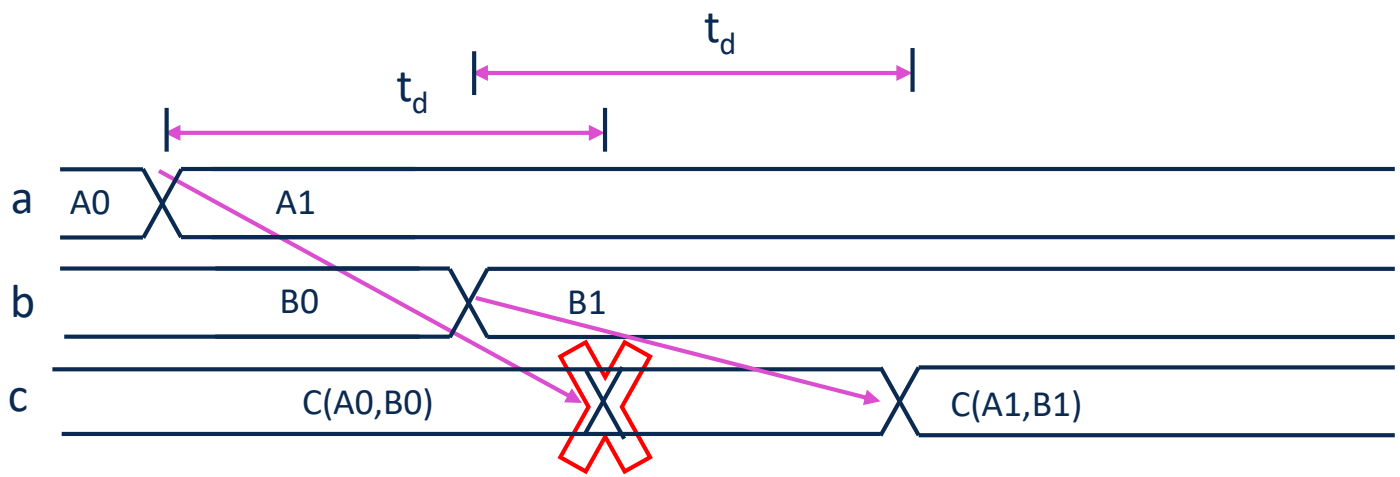
Modelling of Delays



- Abstraction of the delays of combinational logic blocks
- Assignment of delays t_d for timing arcs
- Consideration in digital circuit simulation
- Modelling in simulation as
  - **Inertial Delay** or **Transport Delay**

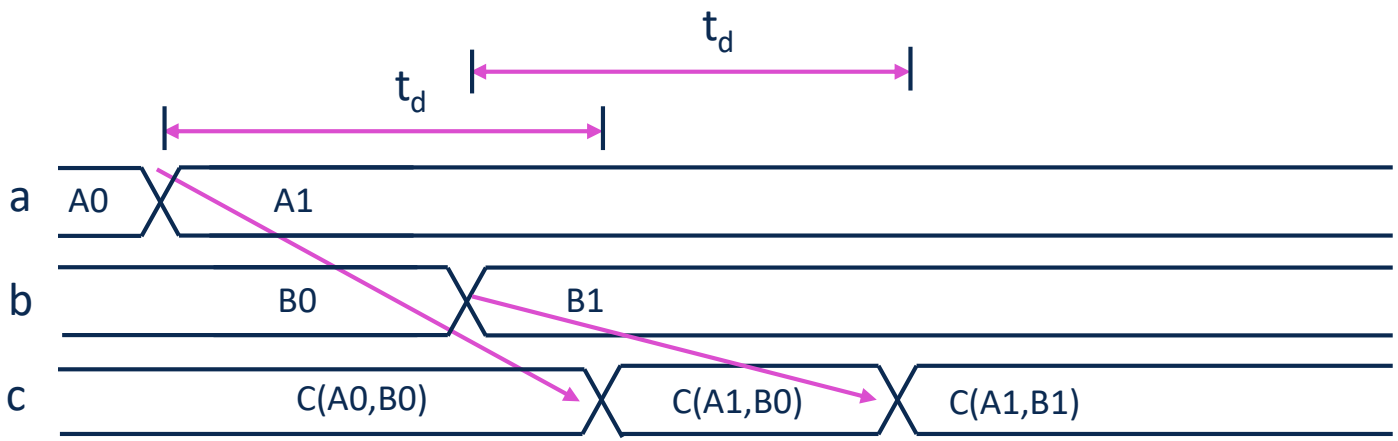
Inertial Delays

- **Inertial Delays:**
  - If the input signal changes after $t < t_d$ → Generation of a new event, discard the previous event
  - Signal propagation to an output signal **after** the input signals are stable



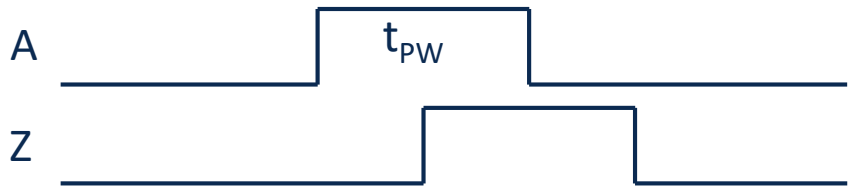
Transport Delays

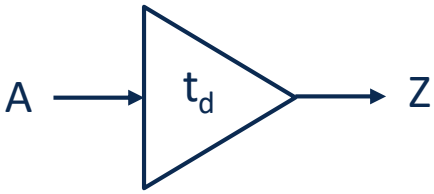
- **Transport Delays**:
  - Propagation of **all** input changes to the output
  - Generation of a new event with every input signal change
  - Disadvantages of modelling combinational logic:
    - Propagation of even shorter glitches
    - Generating many events → slower simulation



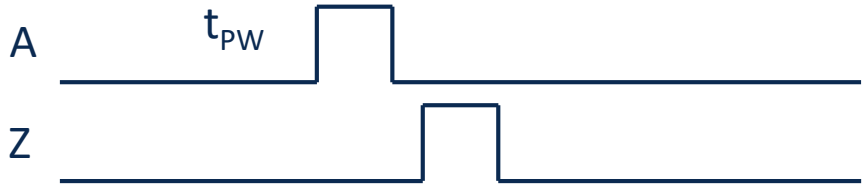
Example: Pulse

$t_{PW}>t_d$

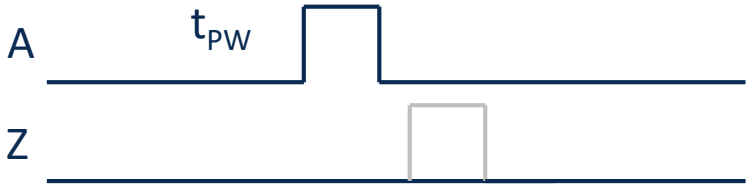




$t_{PW}<t_d$, Transport Delay Model



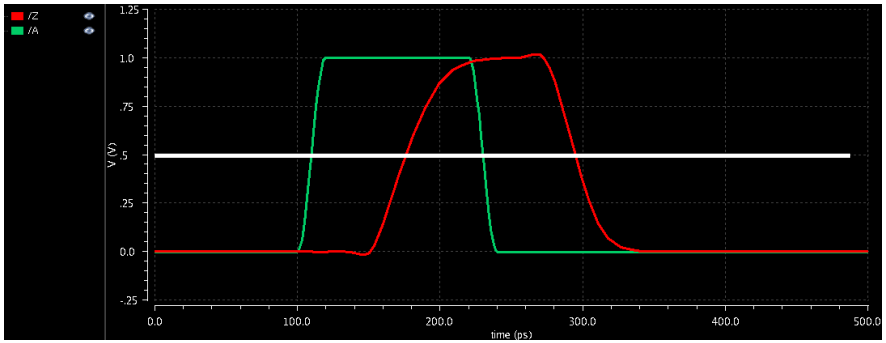
$t_{PW}<t_d$, Inertial Delay Model

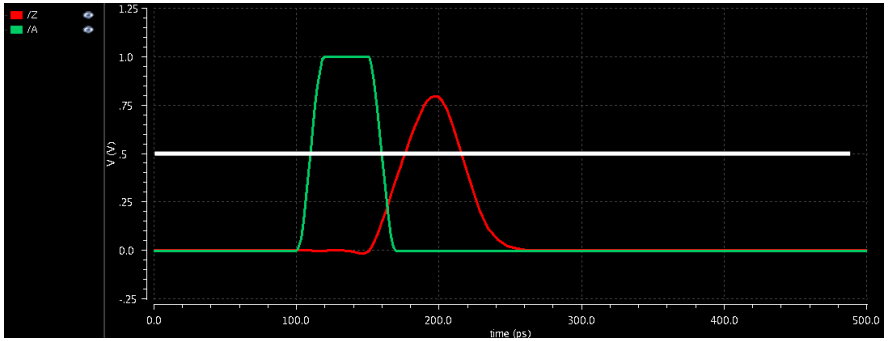


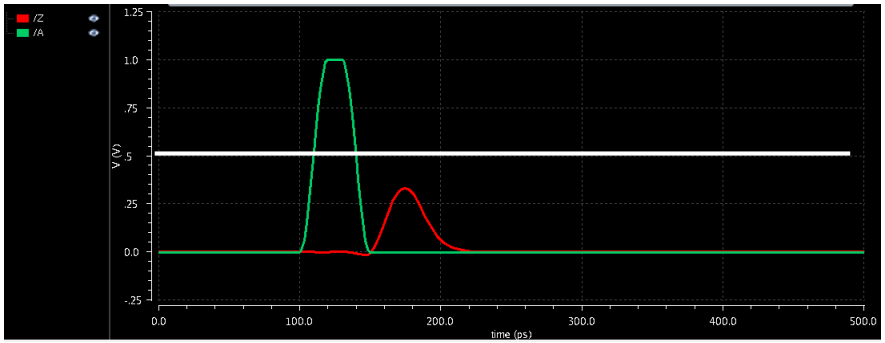
- Typical error:
  - Default delay assignments in gates (e.g. 1ns)
  - → Pulse of <1ns disapppears!

Example: Pulse CMOS Gate

- Simulation of real signal waveforms (Transistor-Level)
- CMOS buffer chain
  - 1. pulse width: 120ps, $t_{d,rise}$ 65ps
  - 2. pulse width: 50ps, $t_{d,rise}$ 65ps
    - Output pulse
    - → Inertial Delay Model not valid
  - 3. pulse width : 30ps, no Z-Pulse
    - → Transport Delay Model not valid







- Inertial and transport delay models are idealized assumptions
- Sufficient for gate-level simulations
- The real delay depends on the inner circuit, which is abstracted here!
- → CMOS Circuits

Delays in Verilog behavioral models

- Assignments can be provided with delays. → #
- ‘timescale <unit>/<resolution>
  - <unit>: unit of time measurement
  - <resolution>: time resolution in simulation

```
//timescale definition
// 1 ns unit, 10ps resolution
`timescale 1ns/10ps
module testbench ();
…
endmodule
```

- Assignments with delay (#) are not synthesizable!

Assignments with delays

- Delays can be specified by:
  - Procedural assignments of the right (RHS) and (LHS) left side
  - Continuous assignments on the left side (LHS)

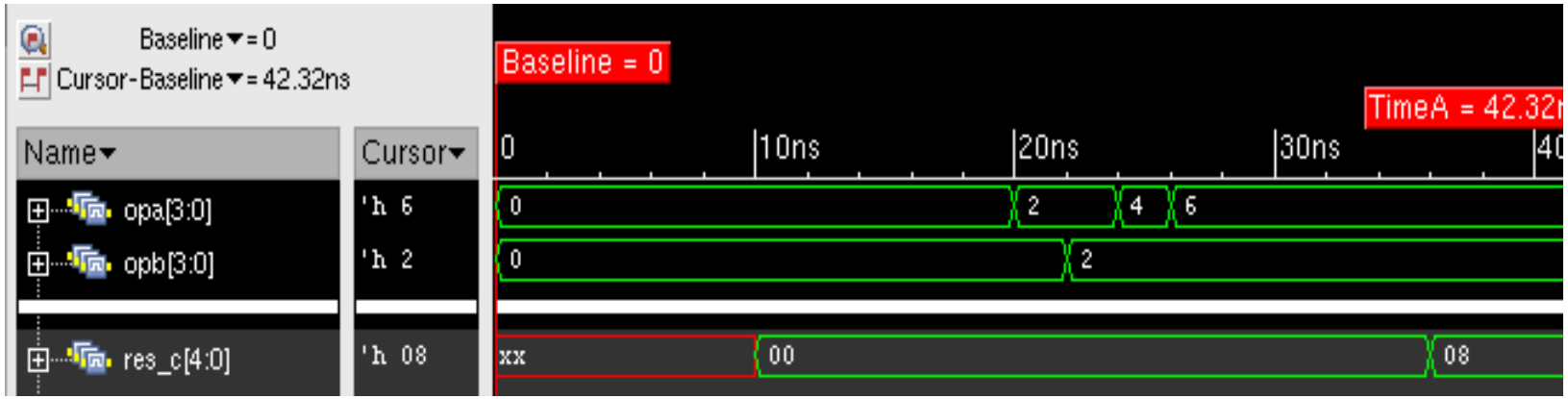
```
reg [3:0] opa, opb;
reg [4:0] res_b_rhs,res_nb_rhs,res_b_lhs,res_nb_lhs;
wire [4:0] res_c;
…

//continuous
assign #10 res_c=opa+opb;

//blocking RHS
always @(opa or opb) res_b_rhs= #10 opa+opb;

//blocking LHS
always @(opa or opb) #10 res_b_lhs= opa+opb;

//non-blocking RHS
always @(opa or opb) res_nb_rhs<= #10 opa+opb;

//non-blocking LHS
always @(opa or opb) #10 res_nb_lhs<= opa+opb;
```

Delay Assignment: Continuous Assignment LHS

- Modeling an inertial delay

```
reg [3:0] opa, opb;
reg [4:0] res_b_rhs,res_nb_rhs,res_b_lhs,res_nb_lhs;
wire [4:0] res_c;
…

//continuous
assign #10 res_c=opa+opb;
```

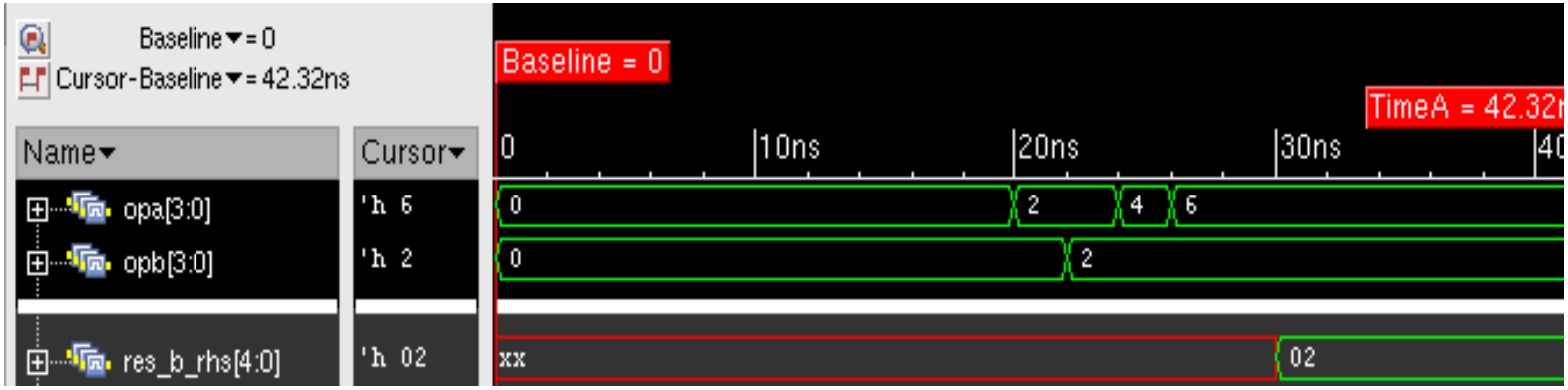
t [ns]	
20	new event 2→res_c @30
22	cancel event 2→ res_c @30 new event 4→ res_c @32
24	cancel event 4→ res_c @32 new event 6→ res_c @34
26	cancel event 6→ res_c @34 new event 8→ res_c @36
36	**8→ res_c**



Delay Assignment: Blocking Assignment (RHS)

```
reg [3:0] opa, opb;
reg [4:0] res_b_rhs,res_nb_rhs,res_b_lhs,res_nb_lhs;
wire [4:0] res_c;
…

//blocking RHS
always @(opa or opb) res_b_rhs= #10 opa+opb;
```

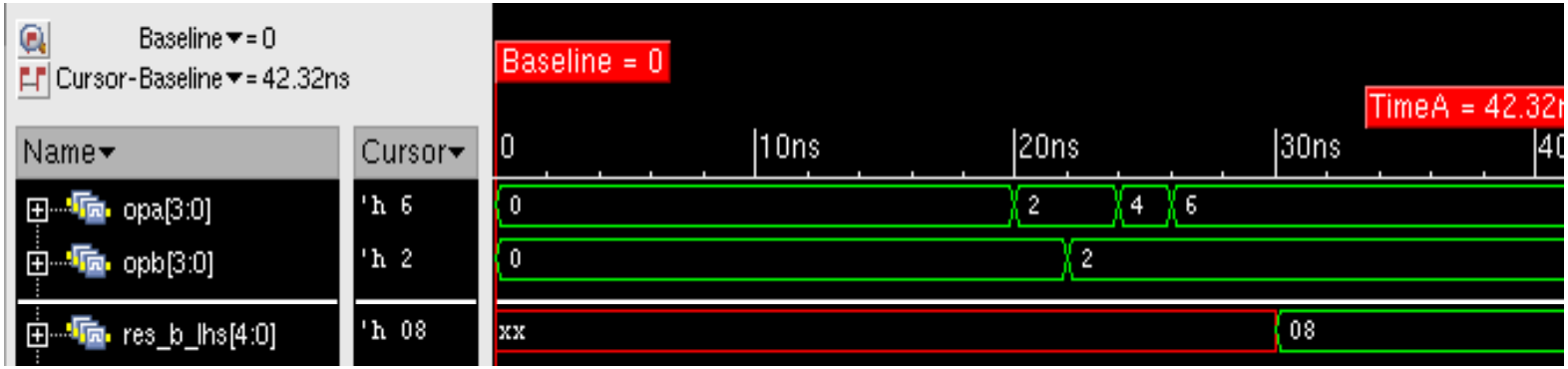
t [ns]	
20	trigger always block new event 2→res_b_rhs @30 stay in always block until 30
22	toggle opb does not trigger!
24	toggle opa does not trigger!
26	toggle opa does not trigger!
30	**2→ res_b_rhs**



Delay Assignment: Blocking Assignment (LHS)

```
reg [3:0] opa, opb;
reg [4:0] res_b_rhs,res_nb_rhs,res_b_lhs,res_nb_lhs;
wire [4:0] res_c;
…

//blocking LHS
always @(opa or opb) #10 res_b_lhs= opa+opb;
```

t [ns]	
20	trigger always block wait until 30
22	toggle opb does not trigger!
24	toggle opa does not trigger!
26	toggle opa does not trigger!
30	new event 8→res_b_lhs @30 **8→ res_b_rhs**

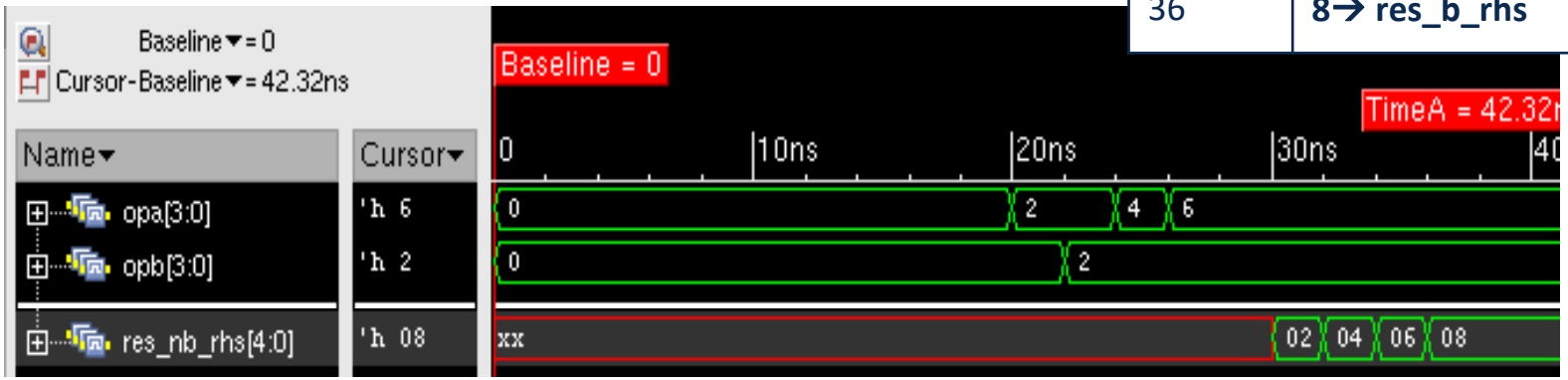


Delay Assignment: Non-blocking Assignment (RHS)

- Modeling a transport delay

```
reg [3:0] opa, opb;
reg [4:0] res_b_rhs,res_nb_rhs,res_b_lhs,res_nb_lhs;
wire [4:0] res_c;
…

//non-blocking RHS
always @(opa or opb) res_nb_rhs<= #10 opa+opb;
```

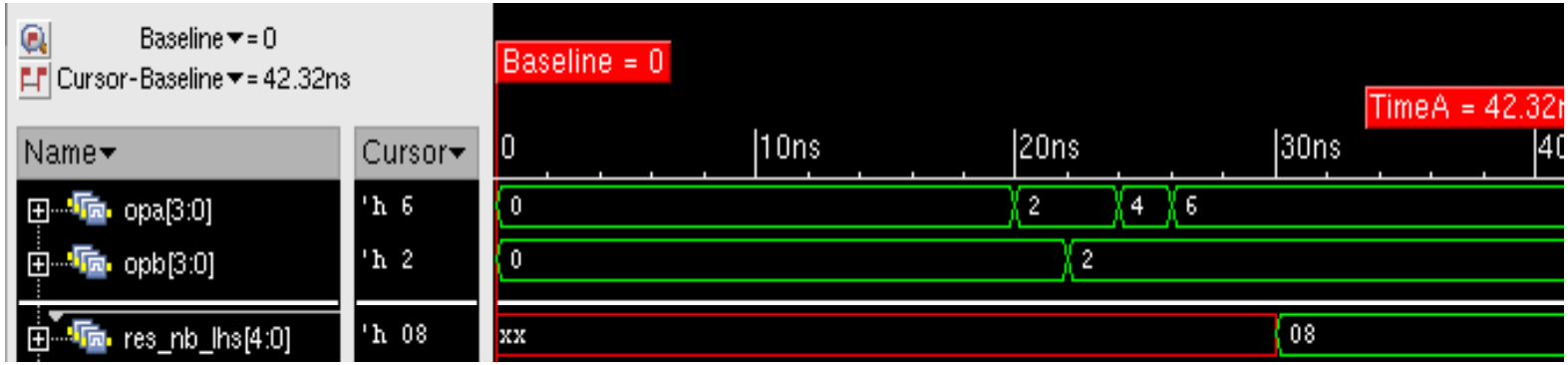
t [ns]	
20	trigger always block new event 2→res_nb_rhs @30
22	trigger always block new event 4→res_nb_rhs @32
24	trigger always block new event 6→res_nb_rhs @34
26	trigger always block new event 8→res_nb_rhs @36
30	**2→ res_b_rhs**
32	**4→ res_b_rhs**
34	**6→ res_b_rhs**
36	**8→ res_b_rhs**



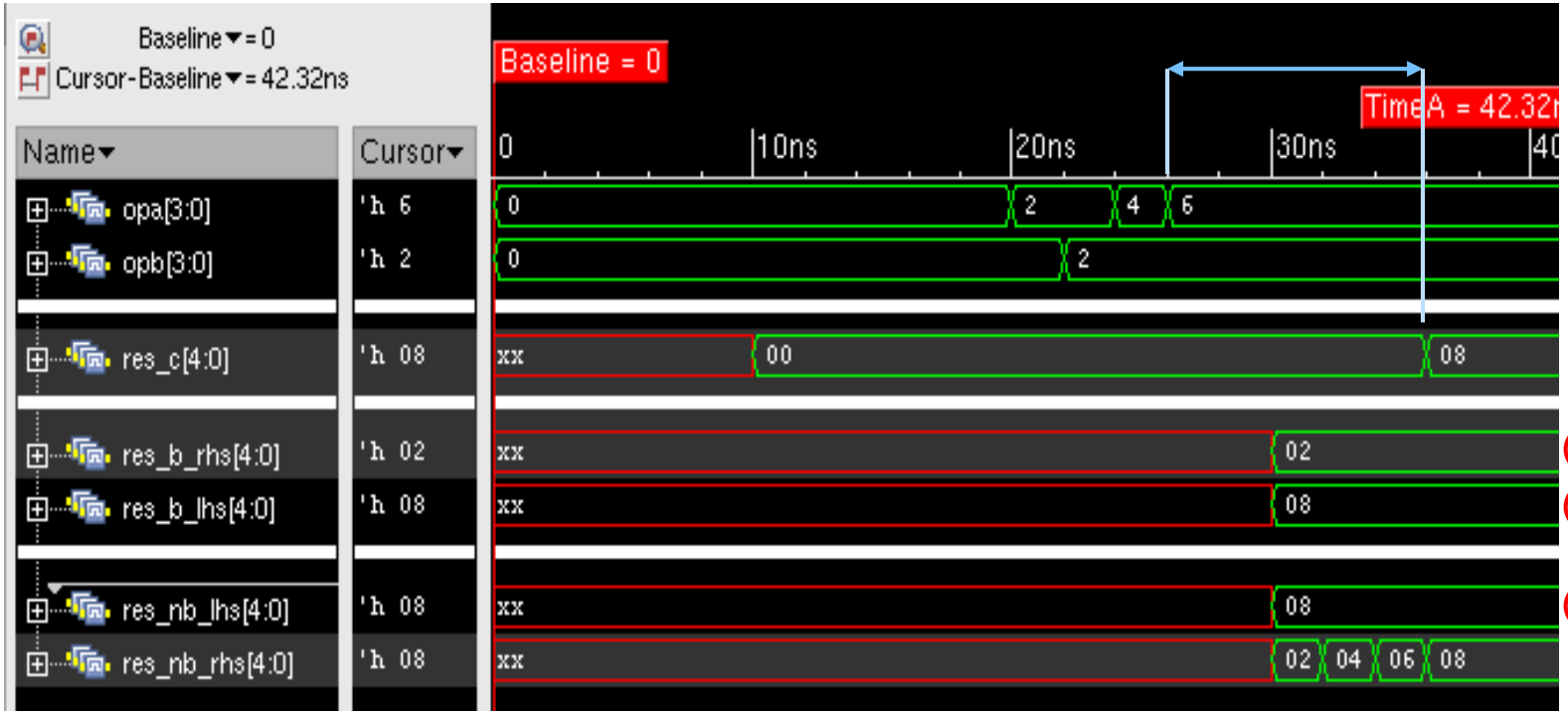
Delay Assignment: Non-blocking Assignment (LHS)

```
reg [3:0] opa, opb;
reg [4:0] res_b_rhs,res_nb_rhs,res_b_lhs,res_nb_lhs;
wire [4:0] res_c;
…

//non-blocking LHS
always @(opa or opb) #10 res_nb_lhs<= opa+opb;
```

t [ns]	
20	trigger always block wait until 30
22	toggle opb does not trigger!
24	toggle opa does not trigger!
26	toggle opa does not trigger!
30	new event 8→res_nb_lhs @30 **8→ res_b_rhs**



Delay Assignments: Summary



inertial delay

transport delay

Summary: Delay Assignments

Delay	Models	Testbench
Continuous Assignment RHS	OK, correct inertial delay modeling	
Blocking Assignment LHS	Not OK	OK, temporal processes/stimuli
Blocking Assignment RHS	Not OK	Not OK
Non-Blocking Assignment LHS	Not OK	Not OK
Non-Blocking Assignment RHS	Transport Delays, **BUT: Non-Blocking Assignments for combinational blocks are not OK**	OK,Modeling transport delays

Modelling of Combinational Logic with Delays

- Recommended method for modelling combinational logic with delay:
  - **Zero-Delay** always block (blocking assignment)
  - Assignment of the delay in continuous assignment (Inertial Delay)

```
reg [3:0] opa, opb;
reg [4:0] res_tmp;
wire [4:0] res;

always @(opa or opb) res_tmp= opa+opb;
assign #10 res=res_tmp;
```

- For details:

Correct Methods For Adding Delays To Verilog Behavioral Models (1999) by Clifford Cummings; International HDL Conference 1999 Proceedings

Specify Blocks

- Complex delay assignment for Gate-Level simulation with **specify s**tatement
  - Separate Rising/Falling Delays
  - Sequential models (e.g. posedge CLK → Q Delay)

- Assignment of delays as **Inertial Delays** for individual **Timing Arcs**
- Annotating the timings typically by synthesis and Place&Route as a result of Static Timing Analysis (STA)
- Additional Specification of constraints possible
  - Setup & Hold times $setuphold()
  - Minimum pluse width $width()

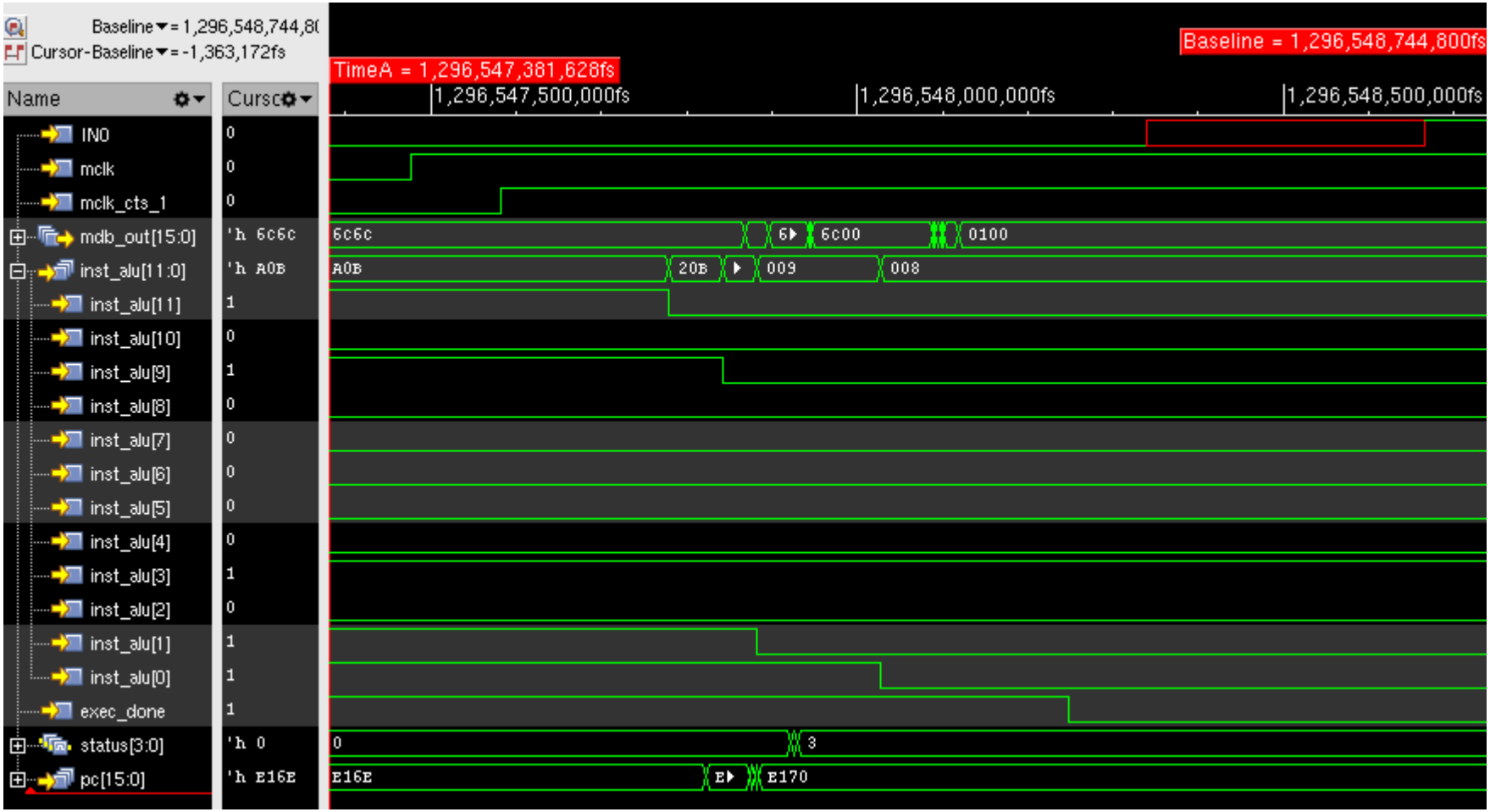
Example: Behavioral model of a standard cell

```
`celldefine
module H_AND2X1_func( B, A, Z );
input A, B;
output Z;

and MGM_BG_0( Z, A, B );

endmodule
`endcelldefine

`celldefine
module H_AND2X1( B, A, Z );
input A, B;
output Z;

H_AND2X1_func H_AND2X1_inst(.B(B),.A(A),.Z(Z));

specify
          (A => Z) = (1.0,1.0); // comb arc A --> Z
          (B => Z) = (1.0,1.0); // comb arc B --> Z
endspecify

endmodule
`endcelldefine
```

Example: Timing model of a FlipFlop

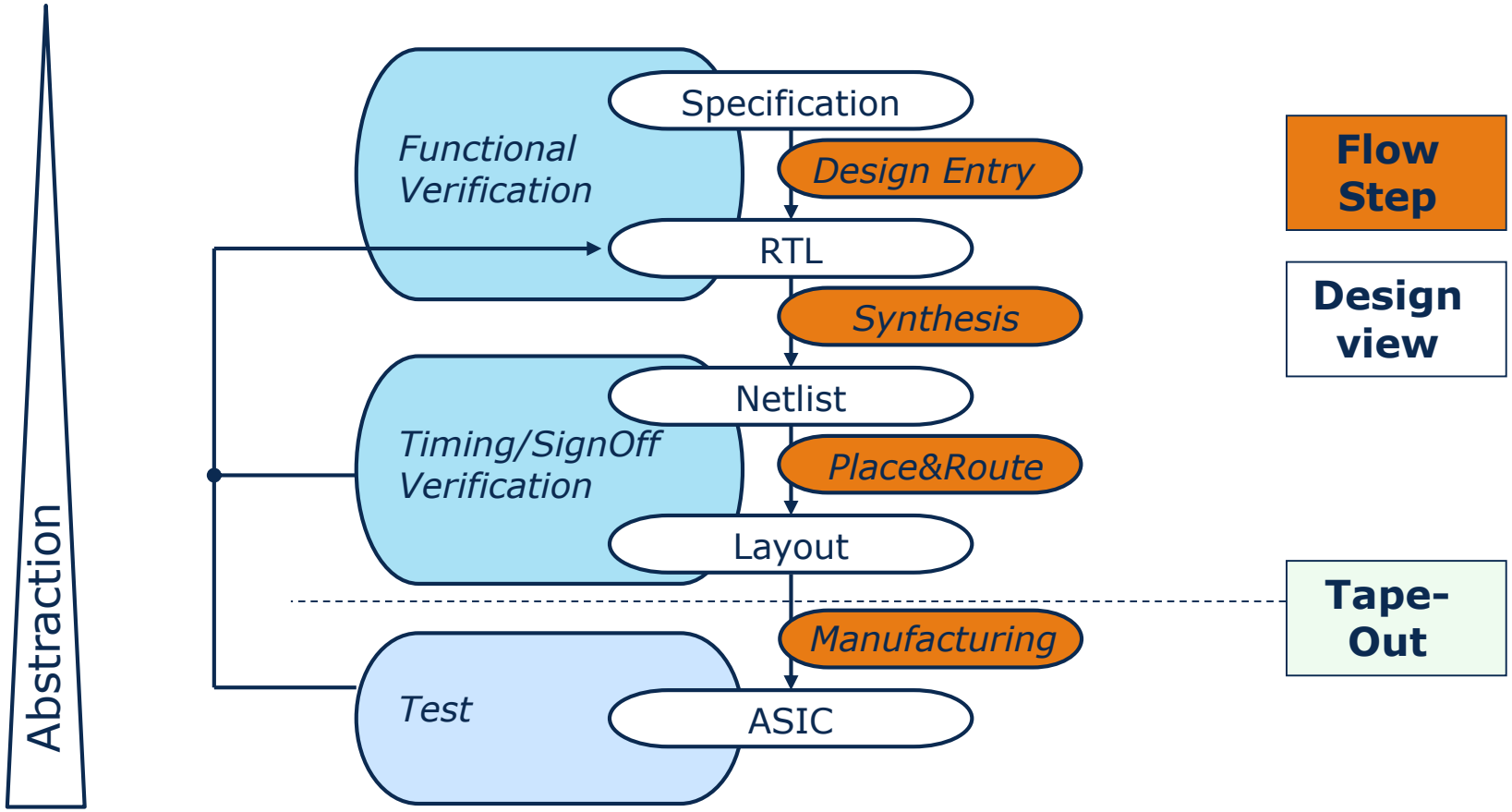
```
`celldefine
module H_DFPQX1( D, CP, Q );
input CP, D;
output Q;
reg notifier;
wire D_delay ;
wire CP_delay ;

H_DFPQX1_func H_DFPQX1_inst(.D(D_delay),.CP(CP_delay),.Q(Q),.notifier(notifier));

   specify
          (posedge CP => (Q : D))  = (1.0,1.0); // seq arc CP --> Q

          $setuphold(posedge CP,posedge D,1.0,1.0,notifier,,,CP_delay,D_delay);
          $setuphold(posedge CP,negedge D,1.0,1.0,notifier,,,CP_delay,D_delay);
          $width(posedge CP,1.0,0,notifier); // mpw CP_lh
          $width(negedge CP,1.0,0,notifier); // mpw CP_hl
   endspecify

endmodule
`endcelldefine
```

Example: Simulation with annotated timing



Verification

Design Flow: Semi-Custom Digital Design



Verification

- Proof of conformity of circuit functionality and specification
- If necessary, comparison of conformity with a reference model

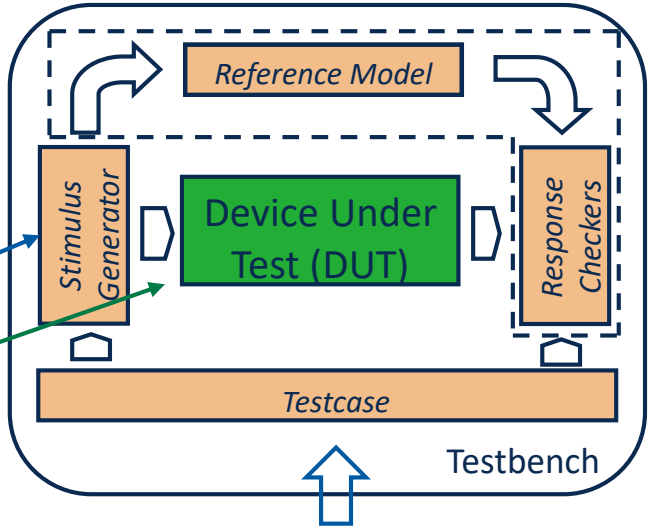
- Verification takes place in different hierarchy levels
  - Block → Module → System

- Verification methods
  - **Verification by circuit simulation**
  - Formal Verification
  - Hardware Emulation/Prototyping (e.g. with FPGAs)

- Current System-on-Chip designs use **up to 80%** of the design time and resources for verification

Testbench

```
module testbench ()

   //Signals
   reg …
   wire …

   `include "my_tasks.v"
   `include "my_functions.v"

   `include „stim_clk_gen.v"

   my_module dut (
      .clk_i(clk),
      .reset_q_i(reset_q),
      .a_i(a),
      .b_o(b));

  `include „testcase.v"

end
endmodule
```




```
// testcase.v
```

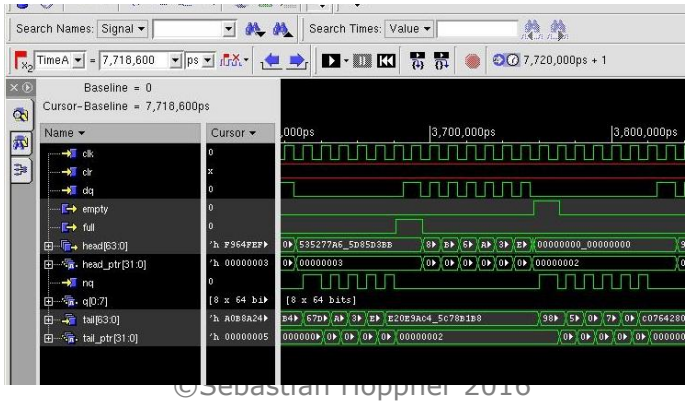
```
// testcase.v
```

```
// testcase.v

initial begin
…
//testcase code
…
end
```

Individual simulation directories

Functional Debugging

- Usage of waveforms in verification of circuit functionality
  - Advantages ☺
    - Intuitive approach
    - Visualizes the behavior of the circuit
    - Gives understanding of the circuit function
    - facilitated debugging
  - Disadvantages ☹
    - Very time consuming for complex systems
    - **Not automated** or **reproducible**
    - Level of verification coverage not measurable

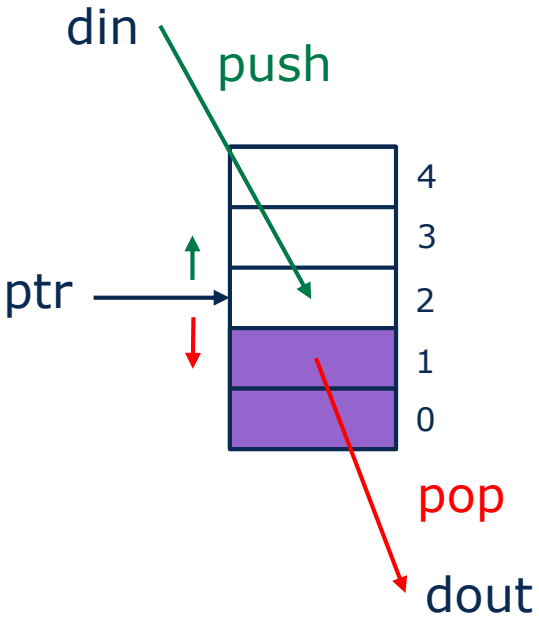


Assertions

- In the description of circuit blocks, **conditions** for the expected circuit behaviour can be formulated
  - Implementation of detailed specifications at RTL level possible
  - Verification is already supported in the RTL implementation
- Checking the conditions takes place only in the circuit model, not in the realized hardware
- Use of **non-systhesizable** HDL constructs
- Verification of these conditions in the **simulation**
- Termination in case of violation of conditions

Example: Stack with Assertions

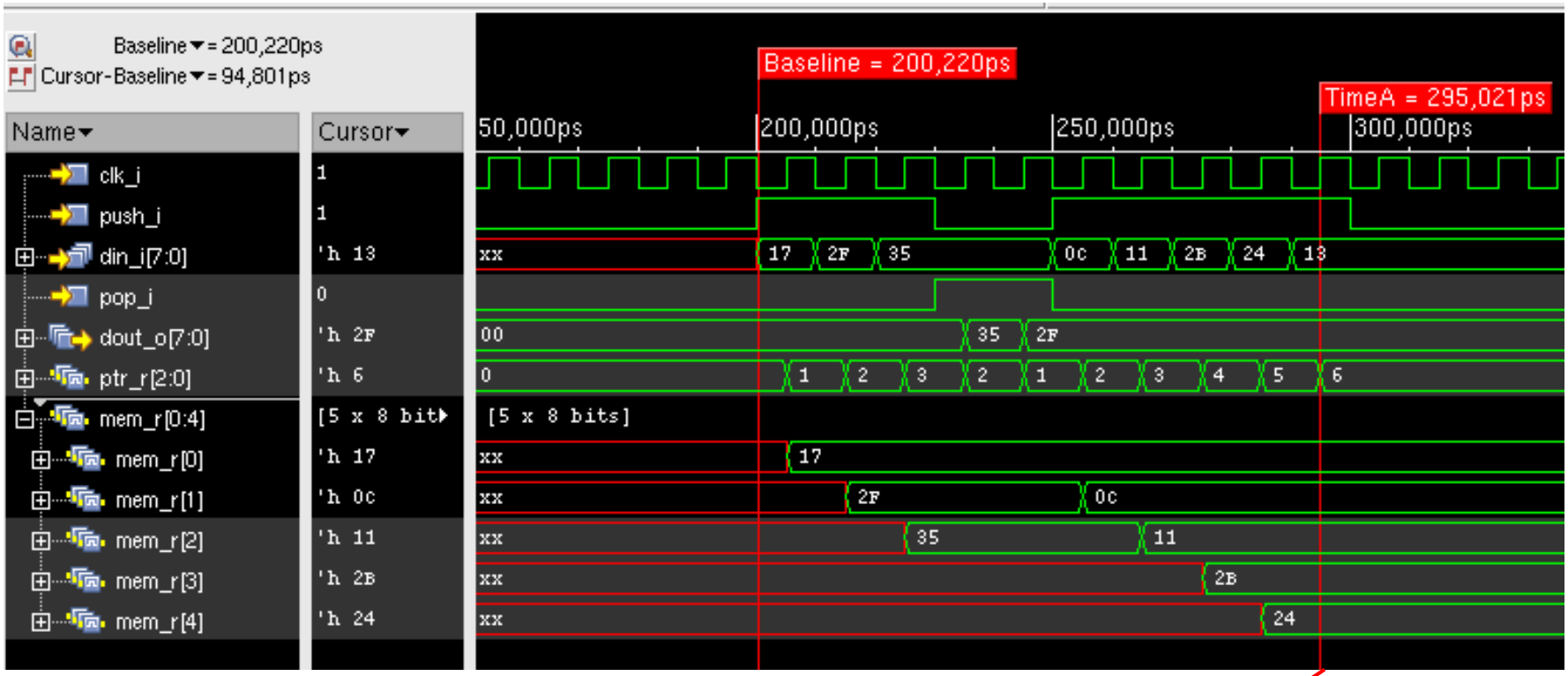
```
module stack
(reset_q_i,clk_i,push_i,pop_i,din_i,dout_o);

parameter DBITS=8;
parameter PTRBITS=3;
parameter DEPTH=5;
input  reset_q_i, clk_i, push_i, pop_i;
input  [DBITS-1:0]        din_i;
output [DBITS-1:0]        dout_o;

reg [DBITS-1:0] mem_r[0:DEPTH-1];
reg [PTRBITS-1:0] ptr_r;
reg [DBITS-1:0] do_r;

wire [PTRBITS-1:0] ptr_m1, ptr_p1;
assign ptr_m1=ptr_r-1;
assign ptr_p1=ptr_r+1;

always @(posedge clk_i) begin
   if (push_i==1'b1) mem_r[ptr_r]<=din_i;
end
…
```



Example: Stack with Assertions

```
…
always @(posedge clk_i or negedge reset_q_i) begin
   if (reset_q_i==1'b0) begin
      ptr_r<=0;
      do_r <=0;
   end
   else begin
      if(push_i==1'b1) begin
         // synopsys translate_off
         if(ptr_r==DEPTH) $display("WARNING: STACK OVERFLOW at time %e",$realtime);
         // synopsys translate_on
         ptr_r<=ptr_p1;
      end
   else if (pop_i==1'b1) begin
      // synopsys translate_off
      if(ptr_r==0) $display("WARNING: STACK UNDERFLOW at time %e",$realtime);
      // synopsys translate_on
      ptr_r<=ptr_m1;
      do_r<=mem_r[ptr_m1];
   end
   // synopsys translate_off
   if(push_i&&pop_i) $display("WARNING: SIMULTANEOUS PUSH & POP at time %e",$realtime);
   // synopsys translate_on
   end
end
assign dout_o=do_r;
endmodule
```

Example: Stack with Assertions




```
ncsim> input -quiet .reinvoke.sim
ncsim> file delete .reinvoke.sim
ncsim> run
DONE
WARNING: STACK OVERFLOW at time 2.950000e+02
DONE
Simulation complete via $finish(1) at time 1 US + 0
```

„Self-checking“ Testbenches

- Check: checks a cricuit output
- Compared with
  - Expected value
  - Specification
  - Reference model
- Formulation in the testbench or in the test case
- Counting checks and failed checks (errors)
- Generation of a final report

- Advantages:
  - Fast repeatability of verification when making design changes

Example: Checks with Reference Model

```
//testbench
…
reg [3:0] add_a,add_b;
reg add_cin;
wire [3:0] add_sum, add_sum_ref;
wire add_cout, add_cout_ref;
wire sum_ref;
integer checkcount=0;
integer errorcount=0;

adder adder_i0 (
   .sum_o(add_sum),
   .c_o(add_cout),
   .c_i(add_cin),
   .a_i(add_a),
   .b_i(add_b)) ;

//Reference Model
assign {add_cout_ref,add_sum_ref}=add_a+add_b+add_cin;

`include "testcase.v"
```

```
task check_add;
input [3:0] a;
input [3:0] b;
input cin;
begin
   checkcount=checkcount+1;
   add_a=a;
   add_b=b;
   add_cin=cin;
   #(CLKPERIODE)
   if ((add_sum==add_sum_ref)&&
      (add_cout==add_cout_ref))
   begin
      $display("check: %d %d %d  PASS",a,b,cin);
   end
   else begin
      $display("check: %d %d %d  FAIL",a,b,cin);
      errorcount=errorcount+1;
   end
end
endtask
```

```
//testcase.v
initial begin
  #200
  check_add(3,5,0);
  check_add(6,5,0);
…
end
```

```
check: 3 5 0 PASS
check: 6 5 0 PASS
…
```

Example: Reference Model as Tables

- Implementation of „Reference Models" as tables

```
//Example state transition table

…
//Instance of state transition logic
…

reg [1:0] state_vec[0:15]
reg [1:0] input_vec[0:15]
reg [1:0] next_state_vec[0:15]
integer i;

initial begin
    state_vec =      '{ 2'b00, 2'b00, 2'b00, 2'b00, 2'b01, 2'b01, … };
    input_vec =      ,{ 2'b00, 2'b01, 2'b10, 2'b11, 2'b00, 2'b01, … };
    next_state_vec = '{ 2'b00, 2'b00, 2'b10, 2'b10, 2'b11, 2'b00, … };
end

for (i=0;i<16;i=i+1) begin
          state=state_vec[i];
          in=input_vec[i]
          if (state_nxt==next_state_vec[i]) $display("check: PASS");
          else                              $display("check: FAIL");
end
…
endmodule
```

inputs

expected outputs

Verification of Sequential Processes

- Provide the input data
  - Load memory
  - Set inputs
- Start the circuit block
- Wait until done
- Pick up the output data
  - Read memory
- Compare results
  - Compared with reference data

```
//testcase.v
integer i;
initial begin
   for (i=0;i<CHECKS;i=i+1) begin
      load_mem_data(i);
      set_inputs(i);
      run_dut;
      get_mem_data;
      compare_result;
   end
end
```

Testbench Results

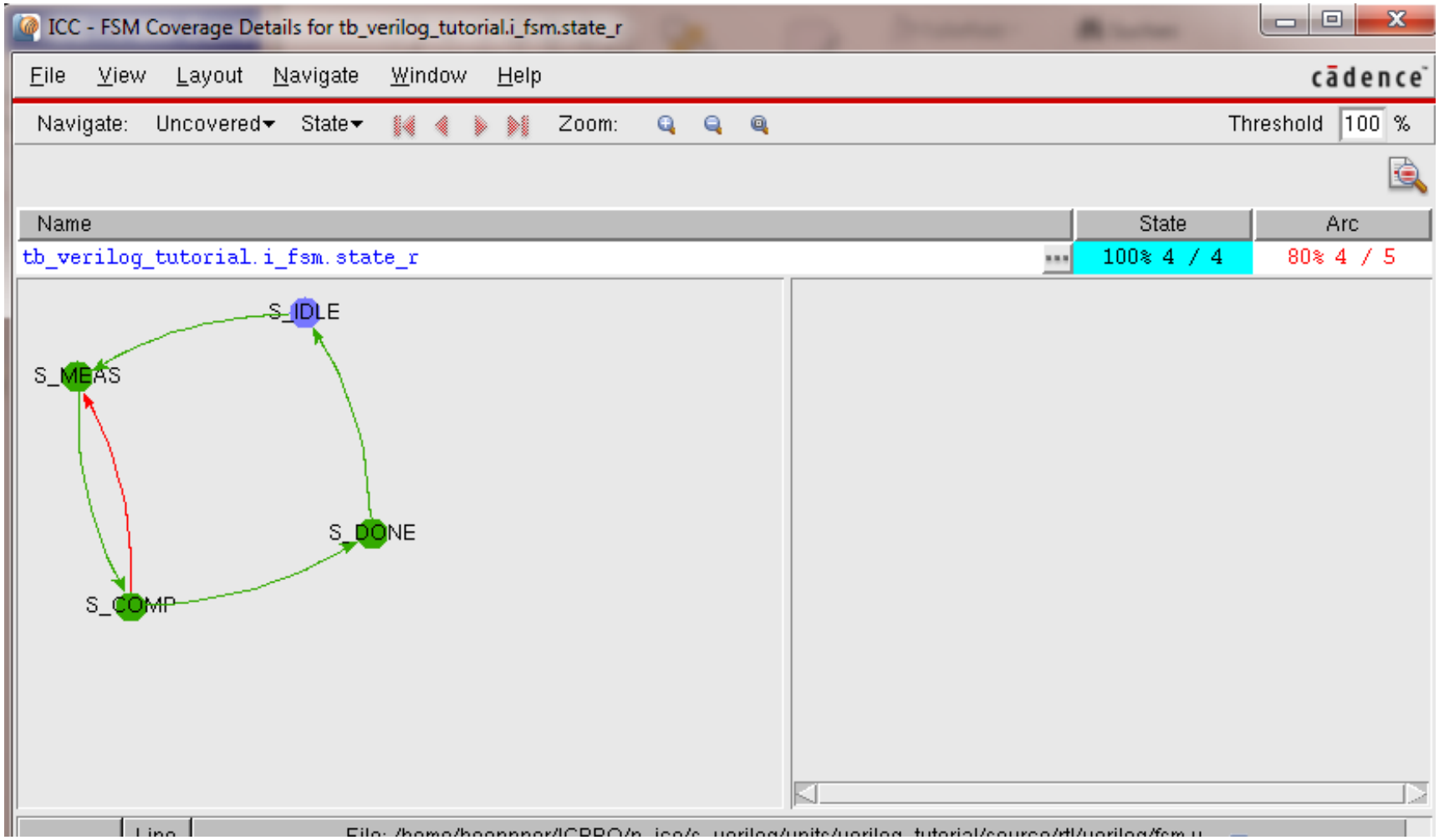
```
task printTestcaseResults; begin
$write("\n");
if ( checkcount == 0 )        begin
   $display(" #   #  #   #  #  #  #   #   ###   #   #  #   #");
   $display(" #   #  ##  #  # #   ##  #  #   #  #   #  ##  #");
   $display(" #   #  # # #  ###   # # #  #   #  # # #  # # #");
   $display(" #   #  #  ##  # #   #  ##  #   #  # # #  #  ##");
   $display("  ###   #   #  #  #  #   #   ###   ## ##  #   #");
   $display("\nTUD_TESTBENCH:WARNING:SIMUNKNOWN: Test result Unknown");
end
else if ( errorcount > 0 )        begin
   $display(" #####   ###   #  #      #####  #### ");
   $display(" #      #   #  #  #      #      #   #");
   $display(" ###    #####  #  #      ###    #   #");
   $display(" #      #   #  #  #      #      #   #");
   $display(" #      #   #  #  #####  #####  #### ");
   $display("\nTUD_TESTBENCH:ERROR:SIMFAIL: Test FAILed");
end
else        begin
   $display(" ####    ###    ####  ####  #####  #### ");
   $display(" #   #  #   #  #     #      #      #   #");
   $display(" ####   #####   ###   ###   ###    #   #");
   $display(" #      #   #      #     #  #      #   #");
   $display(" #      #   #  ####  ####   #####  #### ");
   $display("\nTUD_TESTBENCH:NOTE:SIMPASS: Test PASSed");
end
   $display("Checks run     = %0d", checkcount);
   $display("Errors         = %0d", errorcount);
end
endtask // printTestcaseResults
```

Coverage Analysis

- Measuement of the source code coverage by the simulation
  - Block Coverage
    - Covering code blocks (begin … end)
  - Expression Coverage
    - Covering terms and expressions
  - Toggle Coverage
    - Covering signal changes
  - FSM Coverage
    - Covering states and state transitions (arc)

- **Be aware: the coverage analysis does not check if the output of the circuit has been checked.**

Example: FSM Coverage



Example: Block Coverage

Test : test

Instance	Block	Expression	Toggle
tb_verilog_tutorial.i_fsm	95% 20 / 21	92% 11 / 12	92% 12 / 13

File: /home/hoeppner/ICPRO/p_ice/s_verilog/units/verilog_tutorial/source/rtl/verilog/fsm.v

```
1   55                  end
    56              end
1   57              S_MEAS: begin
1   58                      if (valid_i==1'b1) begin
    59                              state_nxt=S_COMP;
    60                      end
1   61                      else begin
    62                              state_nxt=S_MEAS;
    63                      end
    64              end
1   65              S_COMP: begin
0   66                      if (neg_i==1'b1) begin
    67                              state_nxt=S_MEAS;
    68                      end
1   69                      else begin
    70                              state_nxt=S_DONE;
    71                      end
    72              end
1   73              S_DONE: begin
    74                      state_nxt=S_IDLE;
```

Coverage Report: Uncovered Blocks

Marking:

```
        Instance name: tb_verilog_tutorial.i_fsm
        Module/Entity name: fsm
        File name: /home/hoeppner/ICPRO/p_ice/s_verilog/units/verilog_tutorial/source/rtl/verilog/fsm.v
        Number of uncovered blocks: 1 of 21
        Number of blocks marked COV: 0
        Number of blocks marked IGN: 0

index   uncovered block line no.      line origin description
-----------------------------------------------------------------------
(   9)      66     true part of          66   if (neg_i==1'b1) begin
```

Example: Expression Coverage

Instance	Block	Expression	Toggle
tb_verilog_tutorial.i_fsm	95% 20 / 21	92% 11 / 12	92% 12 / 13

File: /home/hoeppner/ICPRO/p_ice/s_verilog/units/verilog_tutorial/source/rtl/verilog/fsm.v

```
 57                 S_MEAS: begin
 58                         if (valid_i==1'b1) begin
 59                                 state_nxt=S_COMP;
 60                         end
 61                         else begin
 62                                 state_nxt=S_MEAS;
 63                         end
 64                 end
 65                 S_COMP: begin
50% 66                         if (neg_i==1'b1) begin
 67                                 state_nxt=S_MEAS;
 68                         end
 69                         else begin
 70                                 state_nxt=S_DONE;
 71                         end
```

Coverage Report: Uncovered Expressions

Marking:

```
        Instance name: tb_verilog_tutorial.i_fsm
        Module/Entity name: fsm
        File name: /home/hoeppner/ICPRO/p_ice/s_verilog/units/verilog_tutorial/source/rtl/verilog/fsm.v
        Number of uncovered expressions: 1 of 12
        Number of expr items marked COV: 0
        Number of expr items marked IGN: 0

Line    Index  Coverage         Expression description
-----------------------------------------------------------------------------
  66 (   3)  50% (1/2)         if (neg_i==1'b1) begin

           neg_i == 1'b1
          <-1-->   <--2-->

             index  hit | <1>      <2>
         == -------------------------------
            (   1)    0 | lhs  ==  rhs
```

Example: Toggle Coverage (Stack)

Coverage Report: Uncovered Toggles

Marking:

```
Instance name: tb_verilog_tutorial.stack_i
Module/Entity name: stack
File name: /home/hoeppner/ICPRO/p_ice/s_verilog/units/verilog_tutorial/source/rtl/verilog/stack.v
Number of signal bits fully toggled: 20 of 37
Number of signal bits partially toggled(rise): 11 of 37
Number of signal bits partially toggled(fall): 0 of 37
Number of signal bits marked COV: 0
Number of signal bits marked IGN: 0
```

Hit(Full)	Hit(Rise)	Hit(Fall)	Signal
0	0	0	din_i[7]
0	0	0	din_i[6]
0	0	0	dout_o[7]
0	0	0	dout_o[6]
0	1	0	dout_o[5]
0	1	0	dout_o[3]
0	1	0	dout_o[2]
0	1	0	dout_o[1]
0	1	0	dout_o[0]
0	1	0	ptr_r[2]
0	0	0	do_r[7]
0	0	0	do_r[6]
0	1	0	do_r[5]
0	1	0	do_r[3]
0	1	0	do_r[2]
0	1	0	do_r[1]
0	1	0	do_r[0]

Coverage Driven Verification Strategies

- Full coverage simulation
  - Simulation of all input vectors in combinational circuits: n-Inputs → 2^n Checks
    - e.g. all input combinations of the FSM logic
  - Advantages:
    - Very simple test case with full coverage verification
  - Disadvantages:
    - Long runtime with many inputs
    - Difficult in sequential circuits

- Guided test cases
  - Explicit simulation of the circuit, especially of the „corner cases"
  - Advantages:
    - high Coverage possible
  - Disadvantages:
    - Requires detailed knowledge of the circuit
    - Large number of test cases → costly and complex setup

Coverage Driven Verification Strategies

- Random test cases
  - Random simulation of the circuit
  - Restrict random range to the allowed values and sequences („constrained random")
    - e.g. deterministic starting (start_i) of a sequential circuit at random input values (din_i)
    - Advantages:
      - few test cases for high coverage
      - Suitable for combinational and sequential circuits
      - Simulation runtime increases coverage
    - Disadvantages:
      - Corner Cases very unlike to happen

- **→ practically combination of several methods used.**

Regression Runs

- **complex designs have many test cases**
- Regressions
  - Self-Checking test cases
  - Automated simulation
  - parallelism possible
  - Automatically generated report
  - Coverage analysis
- Advantages:
  - Automated Verification
  - „measurement" of the current verification status → „Management Report"
  - fast repetition of verification with:
    - Design changes
    - Bug-Fixes
    - RTL→ Synthesis → Place&Route

Regression: fast_reg

Regression Group: *default*

Simulation Run *Unit/Simulator/Testcase [Variant]*	State *Results, Pass/ Fail*	Simulator *Defines/ Params*
blizzard_top_tc_avfs_bypass_mode	finished simulation	
blizzard_top/ncsim/tc_avfs_bypass_mode [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_avfs_bypass_mode_pmbus	finished simulation	
blizzard_top/ncsim/tc_avfs_bypass_mode_pmbus [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_avfs_closed_loop	finished simulation	
blizzard_top/ncsim/tc_avfs_closed_loop [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_clkmeas *primary*	finished simulation	
blizzard_top/ncsim/tc_clkmeas [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_dhry_programmable	finished simulation	
blizzard_top/ncsim/tc_dhry_programmable [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_finish_led	finished simulation	
blizzard_top/ncsim/tc_finish_led [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_hpm_bypass_mode	finished simulation	
blizzard_top/ncsim/tc_hpm_bypass_mode [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_libtest	finished simulation	
blizzard_top/ncsim/tc_libtest [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_pll_change	finished simulation	
blizzard_top/ncsim/tc_pll_change [*default*]	C:2 W:1 N:1 ?	
blizzard_top_tc_power_load_ctrl	finished simulation	
blizzard_top/ncsim/tc_power_load_ctrl [*default*]	C:11 W:10 N:1 ?	
blizzard_top_tc_sanity	finished simulation	
blizzard_top/ncsim/tc_sanity [*default*]	C:2 W:1 N:1 ?	

Summary and Overview

Summary

- Summary:
  - Overview Design Flow
  - Hrdware description concepts
  - HDL Verilog basics
  - Strategies for verification by simulation

- Overview on further topics:
  - Synthesizable Verilog Description
  - RTL-to-GDS Flow (Synthesis, Place&Route, Timing Analysis)
  - Advanced Verification Methods

- **→ Lecture „VLSI Processor Design“**