



**TECHNISCHE  
UNIVERSITÄT  
DRESDEN**

**Elektrotechnik und Informationstechnik, Stiftungsprofessur hochparallele VLSI Systeme und Neuromikroelektronik**

# **Schaltkreis- und Systementwurf**

## Hardwarebeschreibungssprache Verilog



**FAKULTÄT ELEKTROTECHNIK  
UND INFORMATIONSTECHNIK**



**DRESDEN  
concept**  
Exzellenz aus  
Wissenschaft  
und Kultur

- Verilog HDL
  - Übersicht
  - Grundlagen
  - Prozedurale Zuweisungen
  - Hierarchische Elemente (Module, Funktionen, Tasks)
  - Gate-Level Modellierung
  - Verhaltensmodellierung
  - Systemfunktionen
- Schaltungssimulation und Verifikation

- Dieser Vorlesungsteil wird:
  - Grundkonzepte der Verilog HDL vorstellen
  - Methoden zur Beschreibung und Modellierung digitaler Schaltungsblöcke vermitteln
  - Verifikationsstrategien vermitteln
  - → Grundlage für den Einsatz von Verilog im Praktikum
  - Anregung zum Selbststudium geben
- Dieser Vorlesungsteil wird **NICHT**:
  - eine komplette Sprachreferenz der Verilog HDL bereitstellen
  - die Übungen im Praktikum ersetzen

# Verilog - Übersicht

- Hardwarebeschreibungssprache (engl. Hardware Description Language HDL)
- Beschreibung von digitalen Schaltungen in maschinenlesbarer Textform
- Anwendung zur
  - Modellierung
  - Simulation
  - Verifikation
  - Synthese
- Möglichkeit zur Beschreibung von Hardware-Eigenschaften, z.B.
  - zeitliche Abläufe
  - Gleichzeitigkeit

- 1983/84 entwickelt von Phil Moorby (Gateway Design Automation) als **Simulationssprache** „Verilog“
- 1985 erweiterter Sprachumfang und Simulator „Verilog-XL“
- 1988 Synthesewerkzeuge basierend auf Verilog verfügbar (Synopsys Design Compiler)
- 1990 Übernahme durch Cadence Design Systems
- 1993 85% aller ASIC Chipentwürfe in Verilog
- seit 1995 freier Standard → IEEE Standard 1364-1995 (Verilog-95)
- 2001 Erweiterung IEEE Standard 1364-2001, „Verilog2001“

- Standard
  - IEEE Standard Verilog® Hardware Description Language; IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)
    - *verfügbar über IEEE Explore*
- Bücher
  - 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](http://www.ece.umd.edu/courses/enee359a/verilog_tutorial.pdf)

- Algorithmisch/Verhalten
  - High-level Design Konstrukte
- Register-Transfer-Level (RTL)
  - Datenfluss zwischen Registern
  - Grundlage für die Synthese
- Gatter-Level
  - Beschreibung von Logikgattern und deren Verbindungen
- Schalter-Level
  - Beschreibung von Schaltern (Transistoren) sowie Speicherknotten

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

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

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

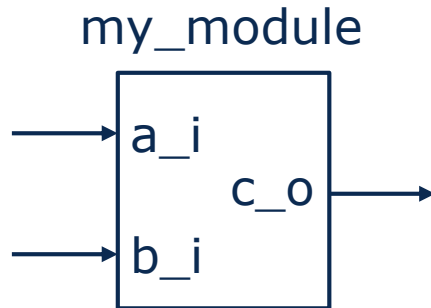
```
...  
wire g, s, d;  
nmos(d,s,g);  
...
```

Gemeinsame Simulation aller Abstraktionsebenen möglich!



- VerilogA
  - Sprache zur Modellierung von analogen (zeit- und wertkontinuierlich) Komponenten
  - keine reinen digitalen Verilog Konstrukte möglich
  - Simulation erfordert reinen Analogsimulator
  - Anwendungen:
    - Bauelementmodellierung
    - Modellierung von analogen Komponenten (z.B. OPV)
- VerilogAMS
  - Erweiterung des Verilog Sprachumfangs zur Modellierung von analogen Signalen
  - Erlaubt Mixed-Signal Simulation unter Nutzung von digitalem und analogem Solver
  - Anwendungen:
    - Modellierung von Mixed-Signal Systemen
- System Verilog
  - Hardware-Beschreibungs- und Verifikationssprache (HDVL)
  - Enthält komplexere Datentypen (ähnlich C/C++), Möglichkeit der objektorientierten Programmierung sowie Konstrukte zur Verifikation
  - Anwendungen:
    - Modellierung und Verifikation komplexer Systeme
    - Testbenches und Verifikations IP

# Verilog - Grundlagen



```

// Company : tud
// Author : hoepfner
// 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:
  - für RTL 1 Module pro Verilog .v File
  - für Bibliotheken (z.B. Standardzellen) auch mehrere Module pro Datei

- Verilog beschreibt 4 Logikpegel

- Logische 1 : 1`b1



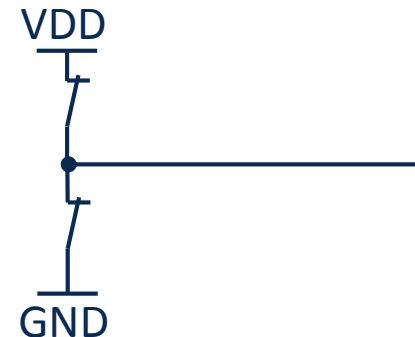
- Logische 0 : 1`b0



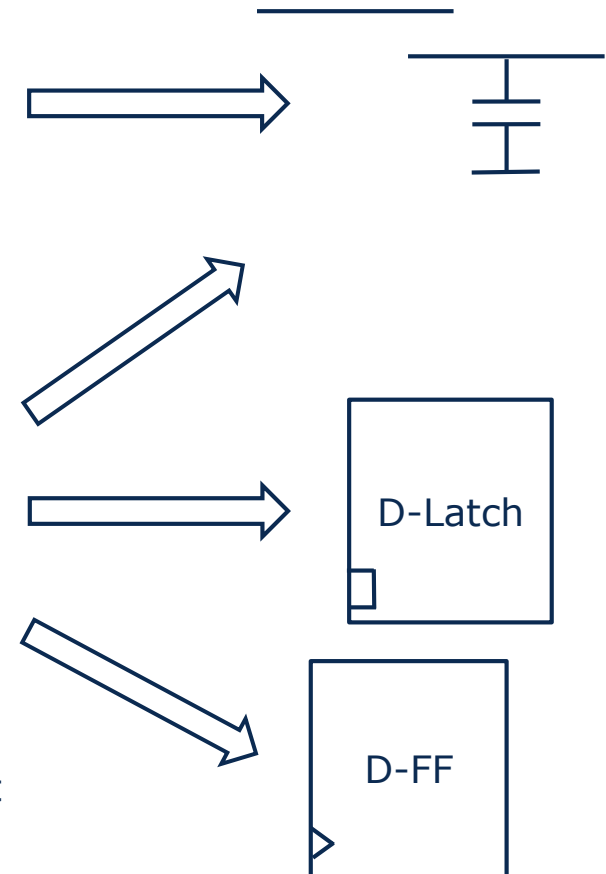
- hochohmig/tri-state : 1`bz



- unbekannt : 1`bx



- Es gibt 2 verschiedene Datentypen zur Beschreibung elektrischer Signale:
- „Netz“ Datentyp
  - Speichert keinen Wert
  - muss kontinuierlich getrieben werden
  - Beispiel:
    - **wire** → elektrische Leitung
- „Register“ Datentyp
  - Abstraktion eines Speicherknotens
  - beschreibt „Register“ im Simulator
  - Speichert Wert zwischen einzelnen Zuweisungen
  - Beispiele:
    - **reg** → logisches Signal
    - integer → 32 Bit „Register“
    - real → reelle Zahl
    - time → Zeitpunkt
- **ACHTUNG:** der Register Datentyp kann verwendet werden zur Beschreibung von
  - kombinatorischen Signalen
  - Speichern (Latch, FlipFlop, SRAM)

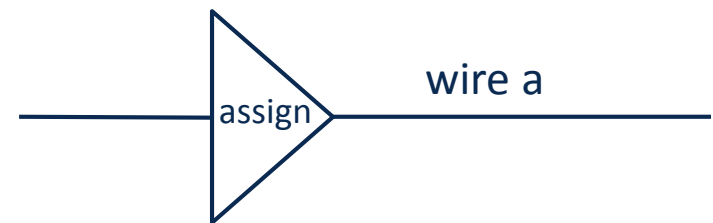


- Beschreibt elektrische Signale
- Kontinuierliche Wertzuweisung durch
  - **assign** statement
  - Gatter- oder Schalter
  - Modulinstanzen
- Verhalten bei Mehrfachzuweisungen:

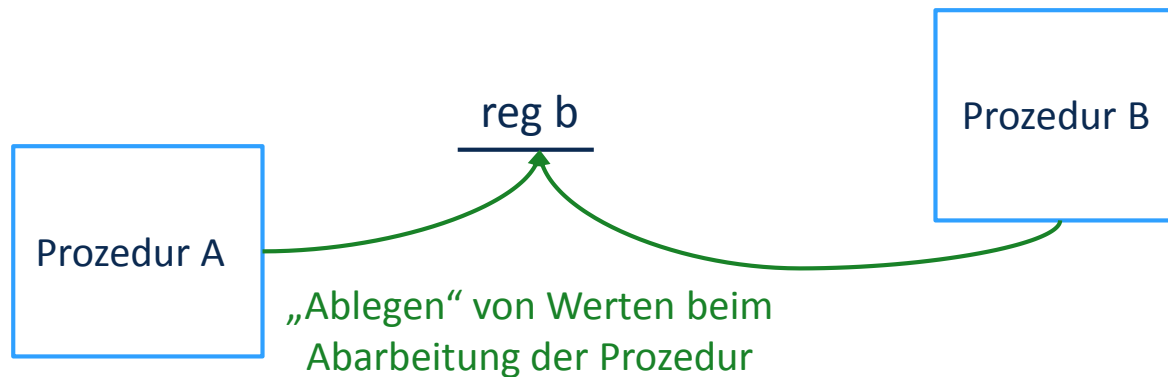
```

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

| wire | 0 | 1 | x | z |
|------|---|---|---|---|
| 0    |   |   |   |   |
| 1    |   |   |   |   |
| x    |   |   |   |   |
| z    |   |   |   |   |



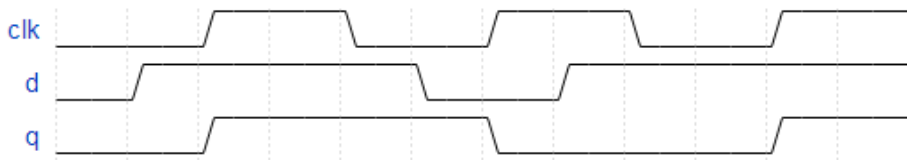
- Register Datentyp reg als Abstraktion eines **Speicherknottens**
- beschreibt „Register“ im Simulator
- Zuweisungen durch **Prozeduren**
- Speichert Wert zwischen einzelnen Zuweisungen



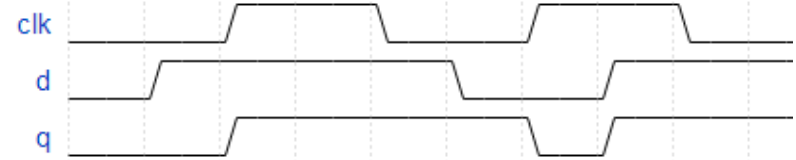
- Register Datentyp reg zur Beschreibung von:
  - kombinatorischen Signalen
  - Latches und RAM Zellen (zustandsgesteuert)
  - Flip-Flops (flankengesteuert)

```
//Kombinatorik
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
```



**Der reg Datentyp beschreibt nicht zwingend ein physisches Register!**



- Möglichkeiten der übersichtlichen Zuweisung von Konstanten:

```
//Beispiele zur Zuweisung von Konstanten  
  
//sized  
reg [7:0] a =8'd23;           //dezimaler Wert  
reg [7:0] b =8'b00010111;    //binärer Wert  
reg [7:0] c =8'h17;          //hexadezimaler Wert  
  
//unsized  
reg [7:0] d =23;             //führt zu 8'b00010111  
reg [7:0] e ='h3;           //führt zu 8'b00000011  
reg [7:0] f ='hf3;          //führt zu 8'b11110011  
  
// '_ ' zur besseren Lesbarkeit  
reg [7:0] f =8'b0001_0111;   //binärer Wert
```

- Die Nutzung von Parametern kann Source Code übersichtlicher gestalten sowie dessen **Wiederverwendbarkeit** erleichtern
- Parameter verhalten sich als **konstante Werte**
- Parameter lassen sich bei der Modul-Instantiierung überschreiben

```
//Beispiel Modul mit Parametern  
module incremter (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
```

- Signale und Modul Pins können zu Bussen zusammengefasst werden
- Beispiele:

```
//8-Bit Register ohne 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
```

```
//Signalzuweisungen mit Bussen
wire a;
wire [7:0] b,c,d,e; //8-Bit Bus
reg [3:0] r; //4-Bit Register

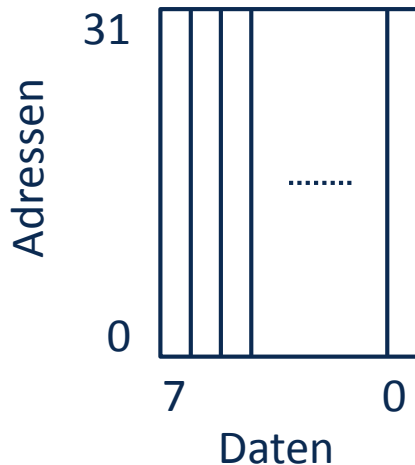
assign a=b[7]; //1-Bit Zuweisung
assign c=8'b0011_11xz; //8-Bit Konstante

//Zusammengesetztes 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]; //Teil-Wort Zuweisung
end
```

- Speicherblöcke lassen sich als Array definieren



```

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
end

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

```

- Signaltreiber können verschiedene Treiberstärken besitzen

|   | Name    | Level | Abbk. |           |
|---|---------|-------|-------|-----------|
| 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   |           |

- Kontinuierliche Zuweisungen

```
//Beispiele für Treiberstärken für kontinuierliche Zuweisungen  
assign (strong1, pull0) a = b; //Unsymmetrischer Treiber  
assign (highz1, strong0) c = d; //Open-Drain Treiber
```

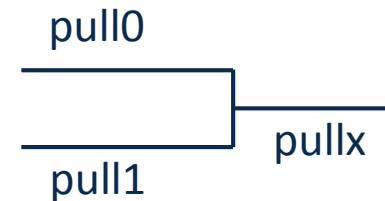
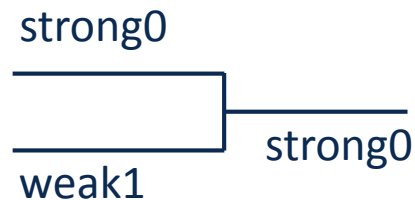
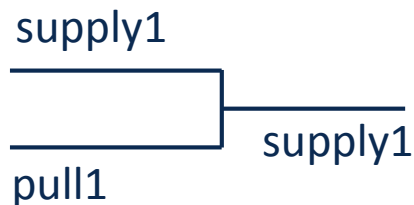
- Gatter Schaltungen

```
//Beispiele für Treiberstärken von Gattern  
and (strong1, pull0) (out,in1,in2);  
nor (strong1, highz0) (out,in1,in2);
```

- Ein Netz kann durch mehrere Zuweisungen getrieben werden

```
//Beispiel Treiberstärken
wire a, b, c;
assign (strong1, pull0) a = b;
assign (pull1, strong0) a = c;
```

- Es können Konflikte beim Zusammenschalten aufgelöst werden
- Wichtigste Fälle:
  - unterschiedliche Treiberstärken; unterschiedliche Logikpegel
    - → Logischer Pegel des stärkeren Signal
    - → Treiberstärke des stärkeren Signals
  - gleiche Treiberstärken; unterschiedliche Logikpegel
    - → Logischer Pegel ergibt sich zu 1'bx bei gleicher Treiberstärke
- Beispiele:




- Operatoren verknüpfen und/oder modifizieren Signale

|           |                    |
|-----------|--------------------|
| { }       | 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          |



|               |  |
|---------------|--|
| ! ~           | höchste Priorität  |
| * / %         |  |
| + -           |  |
| << >>         |  |
| < <= > >=     |  |
| == != === !== |  |
| &             |  |
| ^ ^~          |  |
|               |  |
| &&            |  |
|               |  |
| ?:            | niedrigste Priorität   |

//Beispiel Priorität von Operatoren  
 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 | - |

←

←

- Beispiele zu arithmetischen Operatoren

```
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
```

- Beispiele zu logischen Operatoren und Vergleichsoperatoren

```
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
```

| Gleichheits Operatoren |  |
|------------------------|--|
| a === b                | a gleich b, einschließlich x und z, <b>nicht synthesefähig!</b>  |
| a !== b                | a ungleich b, einschließlich x und z <b>nicht synthesefähig!</b> |
| a == b                 | a gleich b, Ergebnis kann x sein                                 |
| a != b                 | a ungleich b, Ergebnis kann x sein                               |

- Beispiele zu Bit-wise und Reduction Operatoren

```
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 |

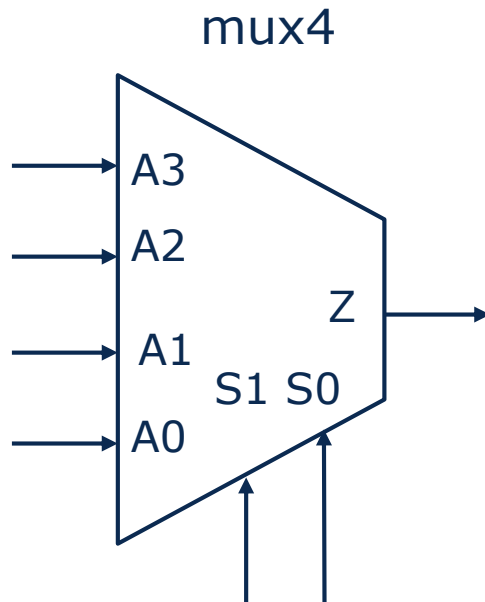
|   | 0 | 1 | x |
|---|---|---|---|
| 0 | 0 | 1 | x |
| 1 | 1 | 1 | 1 |
| x | x | 1 | x |

| ^ | 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 |

- Beispiele zu Shift Operatoren

```
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
```



```

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

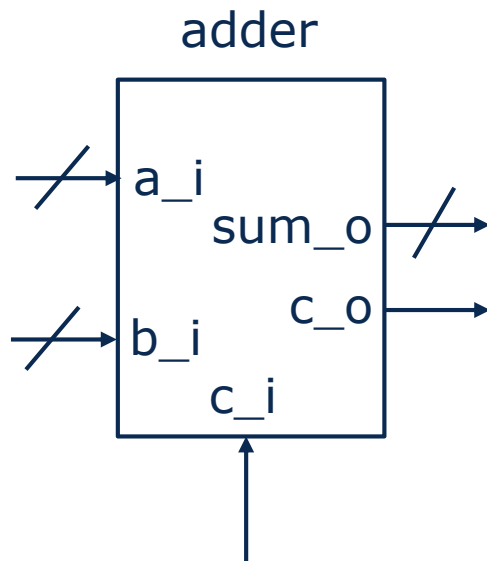
    wire s32,s10,s3210;

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

    //output assignment
    assign Z=s3210;

endmodule

```



```
//Addierer
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
```

```

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

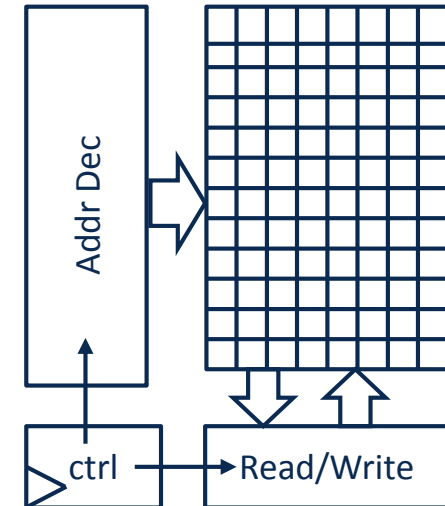
    parameter P_ADDR_WIDTH=8;
    parameter P_DATA_WIDTH=128;

    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;

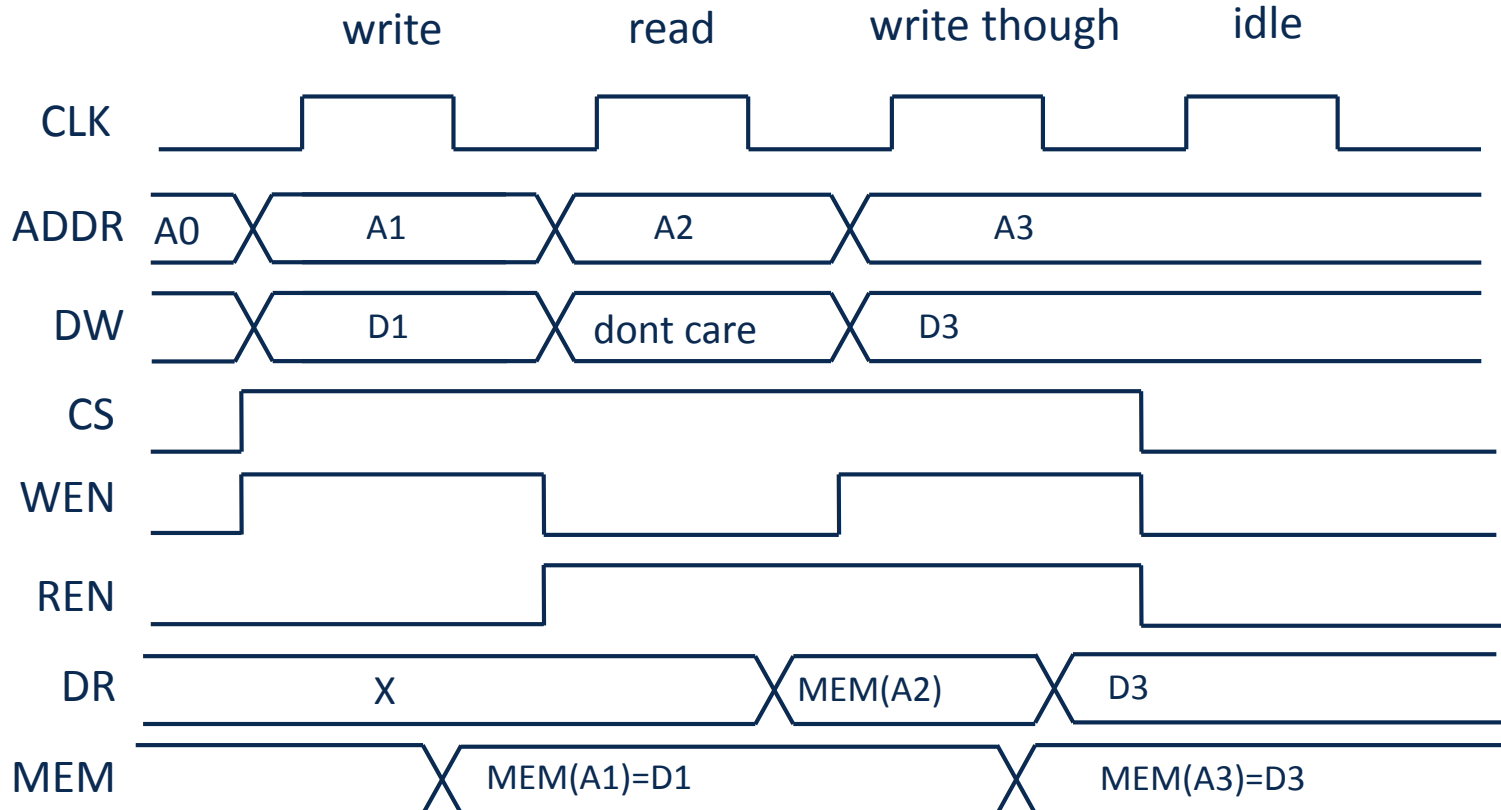
    reg [P_DATA_WIDTH-1:0] mem [0:2**P_ADDR_WIDTH-1];
    reg [P_DATA_WIDTH-1:0] dr_r;

    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

```



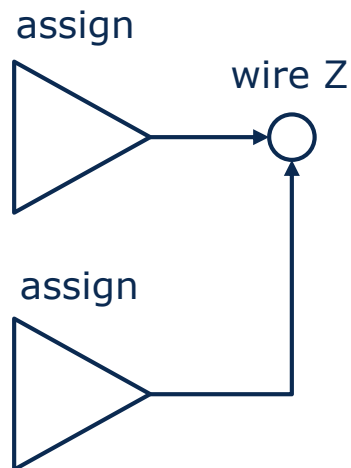




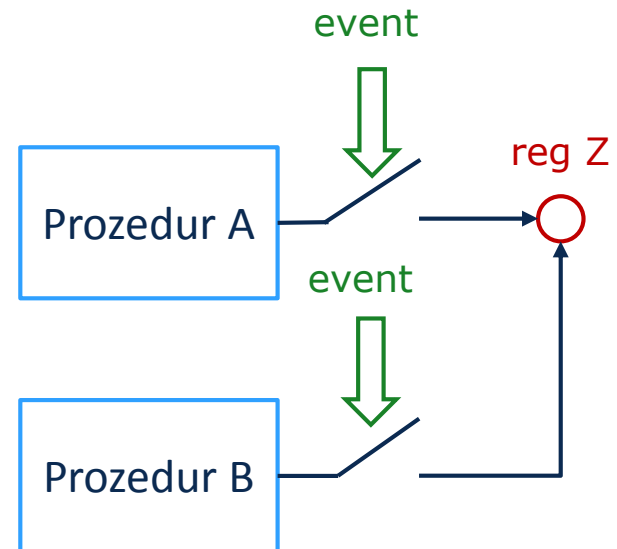
# Verilog – Prozeduren

- Kontinuierliche Zuweisungen (assign)
  - → Treiben ein Netz (wire) permanent
  - → Wert ändert sich sobald die „inputs“ der Zuweisung sich ändern
- Prozedurale Zuweisungen
  - → schreiben Wert auf einen Register-Datentyp (z.B. reg, integer)
  - → Register-Datentyp hält Wert bis zum nächsten prozeduralen Zugriff
  - → Zuweisungen werden getriggert durch die Control Flow Blöcke (Prozeduren)
    - initial
    - always
    - task
    - function
- Prozeduren können parallel und konkurrierend ablaufen

## Kontinuierliche Zuweisung



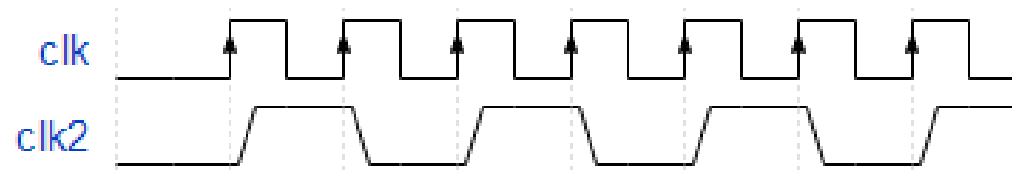
## Prozedurale Zuweisung



```
`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
```



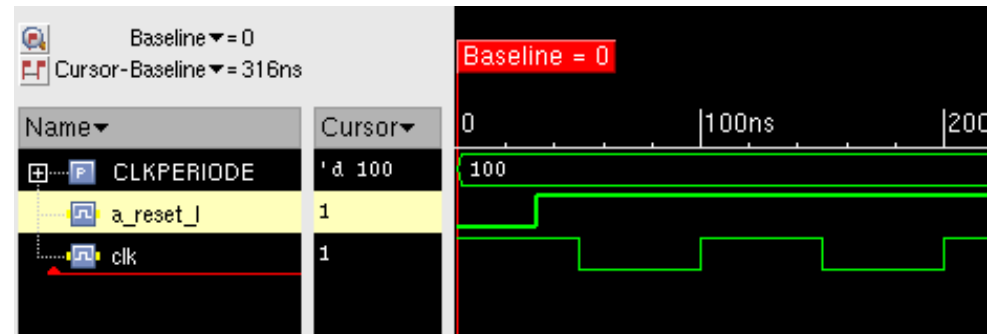
- **initial**

- Ausführung zu Beginn der Simulation im Zeitschritt 0.
- typische Anwendungen
  - Initialisieren von Registern
  - Starten von Stimuli (Waveforms) in Testbench Umgebungen
- **nicht synthesefähig!**

```
//Beispiel Initial Prozedur
reg clk;
reg a_reset_l;
initial clk = 1'b1;

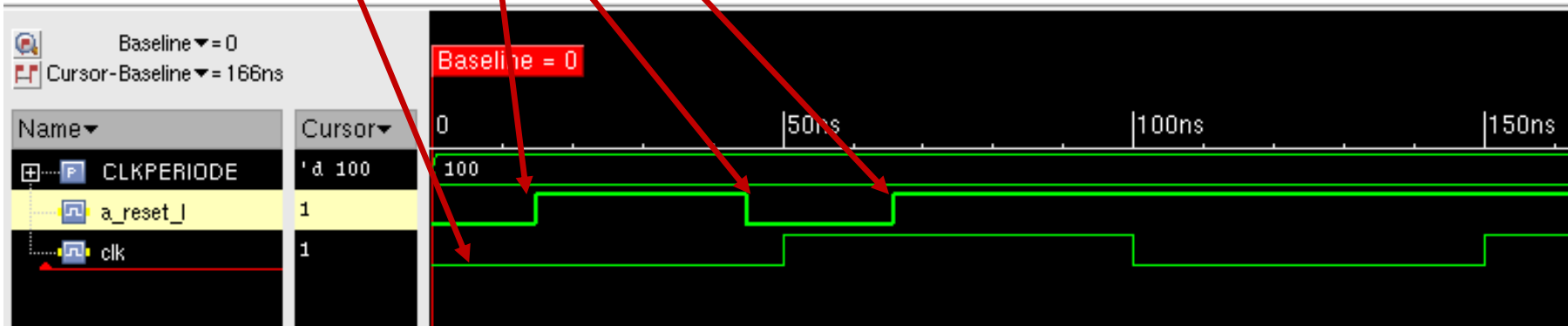
always #(CLKPERIODE/2) clk = !clk;

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



```
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  
end
```

- Konkurrierende Ausführung von Prozeduren möglich
- Keine x-Werte wie bei kontinuierlicher Mehrfachzuweisung



- Initialisierung von Register Datentypen ist bei Deklaration möglich
- Komfortabel für Testbenches
- **nicht synthesefähig!**

```
//Beispiel Initialisierung von Register Datentypen  
reg clk=1'b1;  
real a=3.245;  
integer i=1;
```



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

- Permanente Ausführung
- Sinnvolle Anwendung mittels „timing control“ Statements, z.B.
  - Verzögerte Zuweisungen
  - Event Steuerung
  - Sensitivity Liste
- geeignet zur Modellierung und Beschreibung von
  - sequentieller Logik
  - kombinatorischer Logik

- Ausführung von always Prozeduren steuerbar durch „event control statements“ @
  - @ <identifizier> → Signalwechsel
  - @ posedge → steigende flanke
  - @ negedge → fallende flanke
- Kombination der events mit ‚OR‘ möglich

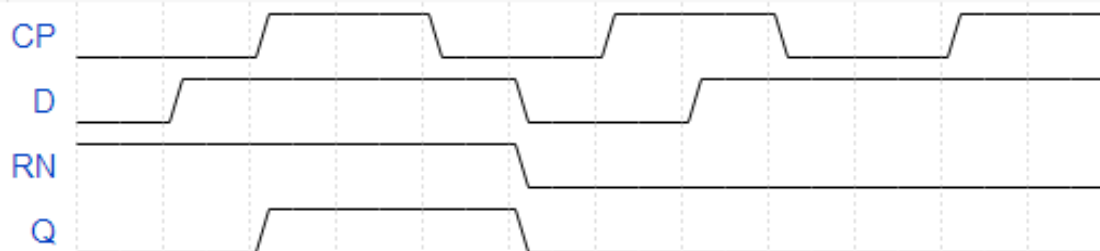
```
// Beispiele event control
always #(CLKPERIODE/2) clk = !clk; //permanente Ausführung

always @(b) a=b; //Zuweisung getriggert durch b

always @(posedge clk) q <= d; //steigende Taktflanke

always @(negedge clk) q <= d; //fallende Taktflanke
```

```
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
```

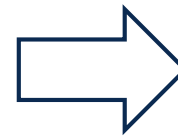


- Ein kombinatorischer always Block wird durch eine Sensitivity Liste getriggert
- Coding Guidelines:
  - Alle rechtsseitigen Signale in Zuweisungen **müssen enthalten sein!**
  - Linksseitige Signale **sollen nicht enthalten** sein → kein „self-triggering“

```

// Beispiel Sensitivity Liste (XOR)

//vollständige Sensitivity Liste
always @(a or b) begin
    c1=a^b;
end
//fehlendes Signal in Sensitivity Liste
always @( a ) begin
    c2=a^b;
end
    
```

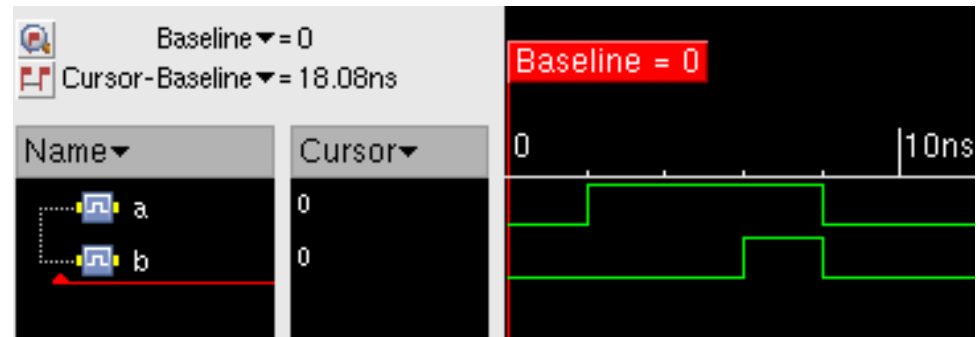


Mismatch zwischen RTL Simulation und Synthesergebnis



- Blocking Assignments (= ) werden **vor** dem folgenden Statement in der Prozedur **unmittelbar** ausgeführt
- Unterbricht (blockt) den prozeduralen Ablauf

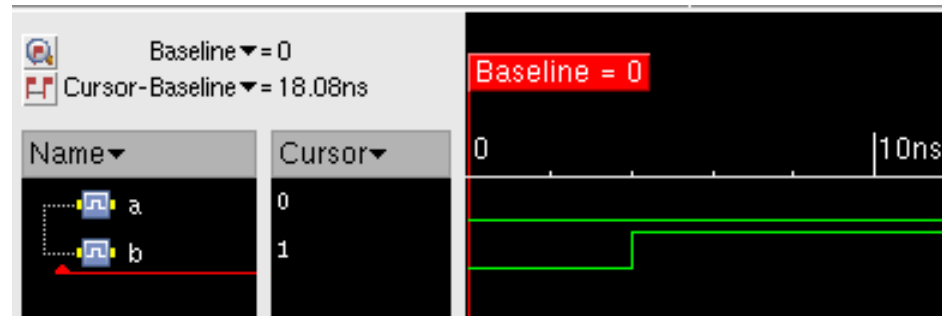
```
// Beispiel Blocking Assignment
initial begin
    a=0;
    b=0;
    a= #2 1;
    b= #4 1;
    a= #2 0;
    b=0;
end
```



- Blocking Assignments eignen sich zur Beschreibung von Signalabläufen (Waveforms, Stimuli)

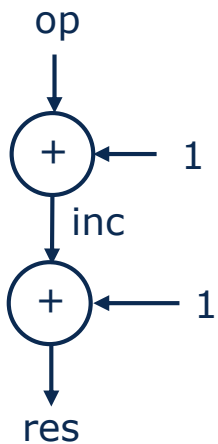
- Non-Blocking Assignments (`<=`) unterbrechen den prozeduralen Ablauf nicht
- Ausführung durch den Simulator in 2 Schritten:
  1. Auswertung der rechten Seite, **Vormerken** der Zuweisung auf die linke Seite
  2. **Durchführen** der Zuweisung zum Zeitpunkt des Event Control Statements

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



- Non-Blocking Assignments eignen sich zur Beschreibung von sequentiellen Logikblöcken deren Timing durch die Event Control Statements definiert ist (z.B. `@(posedge clk)`)

## Kombinatorische Logik



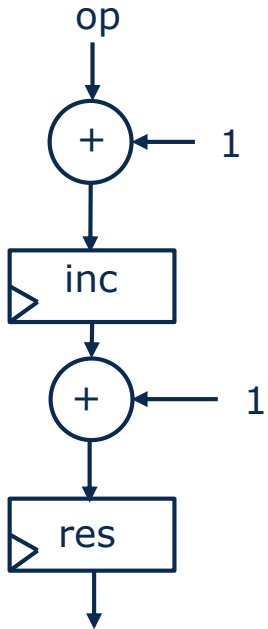
```
// Blocking Assignment
always @(op) begin
    inc1=op+1;
    res1=inc1+1;
end
```

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



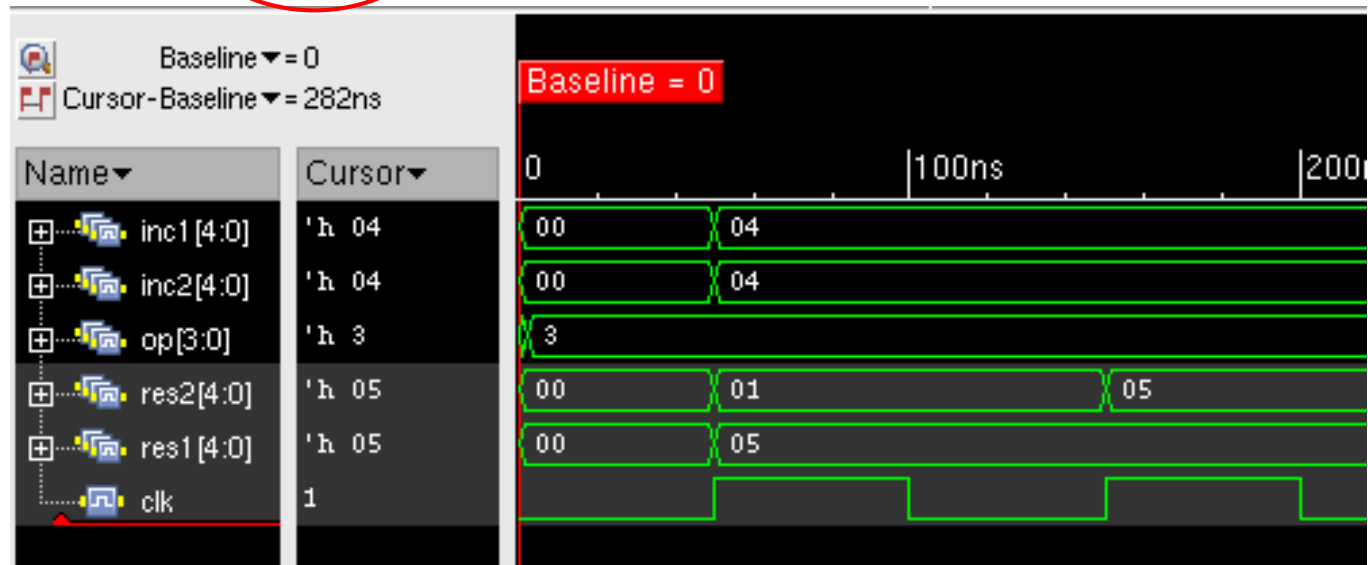
- Nutzung von **Blocking Assignments** zur Beschreibung **kombinatorischer Logik!**

Sequentielle  
Logik



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

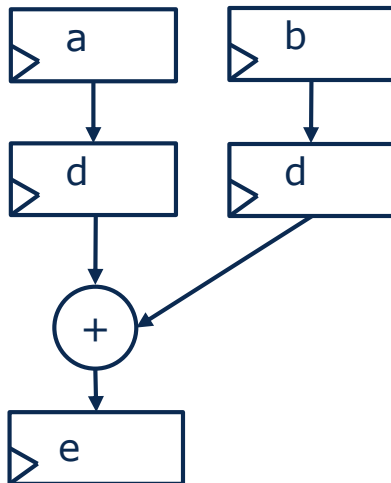
```
// Non-Blocking Assignment
always @(posedge clk) begin
    inc2<=op+1;
    res2<=inc2+1;
end
```



- Nutzung von **Non-Blocking Assignments** zur Beschreibung **sequentieller Logik!**



- Nur eine Zuweisung pro always Block
- Ist die Nutzung von blocking oder non-blocking assignments egal?
- Verwendung mehrerer dieser Blöcke (z.B. Instanzen eines Registers)
- Beispiel:

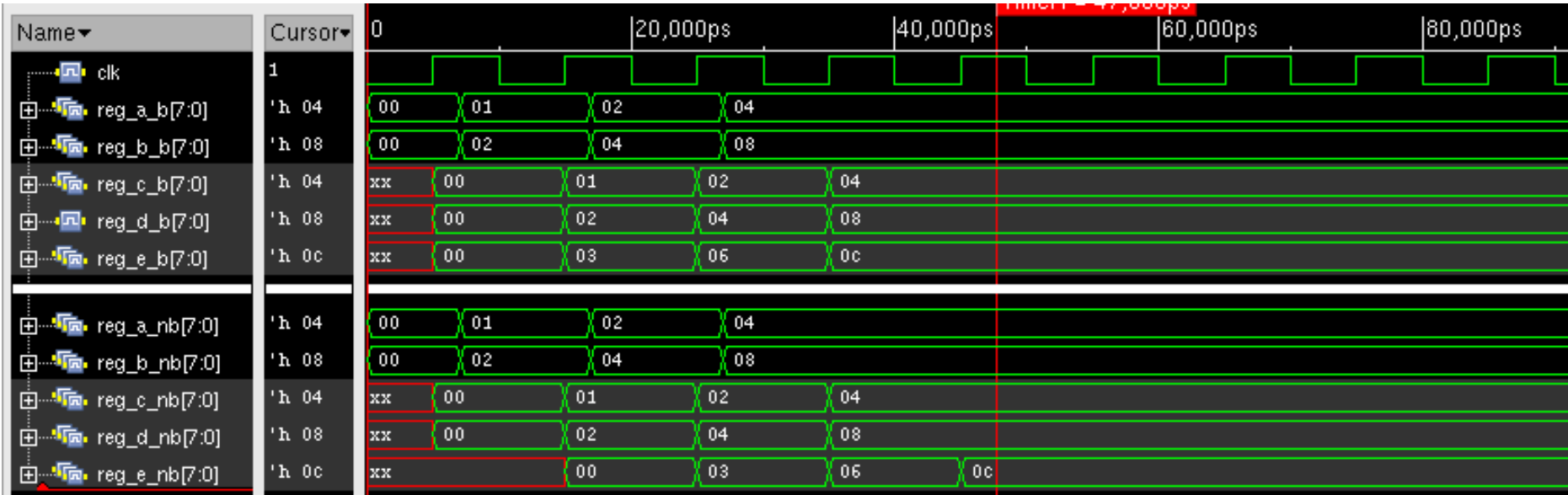


```

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
  
```



- Nutzung von **Non-Blocking Assignments** zur Beschreibung **sequentieller Logik!**
- Auch bei nur **einer Zuweisung** pro always Block

- **Delta Cycle:** Konzept in HDL Simulationen um Events zu ordnen, welche mit dem Zeitabstand von 0 (null) stattfinden
- **Zeitschritt:** Fortschreiten der Simulationszeit

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

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

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

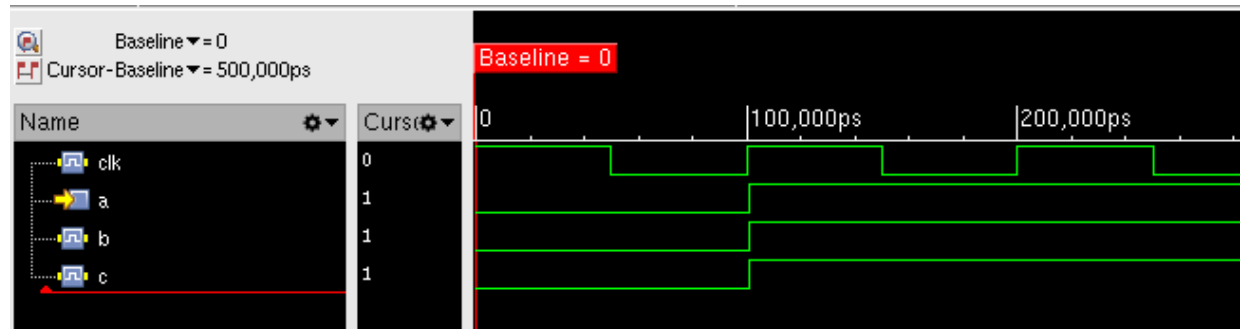


Delta cycle:

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

nächster Zeitschritt:

*keine Zuweisung*



→ **sofortige Signalzuweisung**

```

input clk;
input a;
reg b,c;

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

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



Delta cycle:

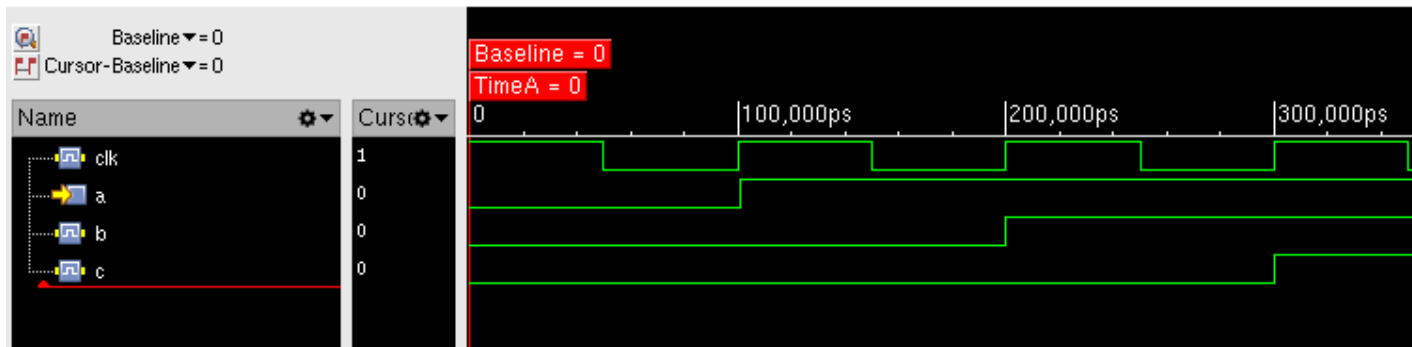
```

tmp_b <- a
tmp_c <- b
    
```

nächster Zeitschritt:

```

b <- tmp_b
c <- tmp_c
    
```



→ Signalzuweisung bei Zeitschritt

- Innerhalb von prozeduralen Zuweisungen können Control Statements verwendet werden zur bedingten Ausführung von Blöcken
  - if/else , case

```
// Beispiel 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
```

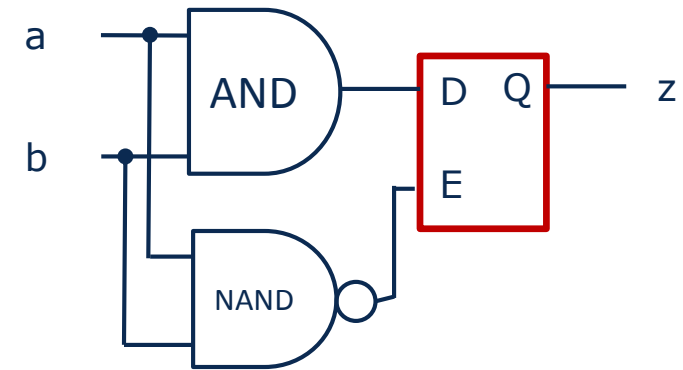
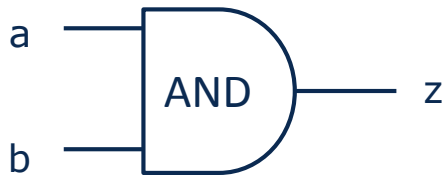
```
// Beispiel case 1
wire [1:0] a;
always @(a) begin

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

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

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

- Bei Control Statements zur Beschreibung kombinatorischer Logik sind Default Zuweisungen zwingend!
- Andernfalls werden sequentielle Elemente (Latches) beschrieben

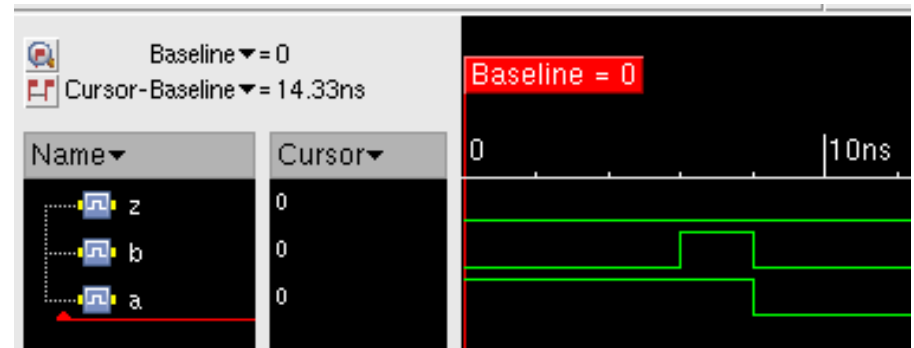
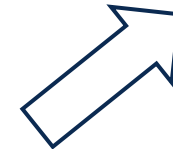


Latch eingefügt!

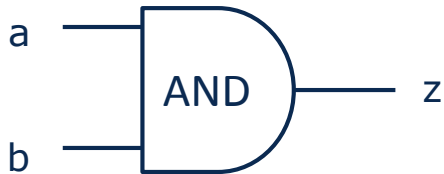
```

// Beispiel IF/ELSE
// ohne default
always @(a or b) begin
  if ((a==0) || (b==0)) begin
    z=0;
  end
end
end

```



- Korrekte Beschreibungsvarianten:

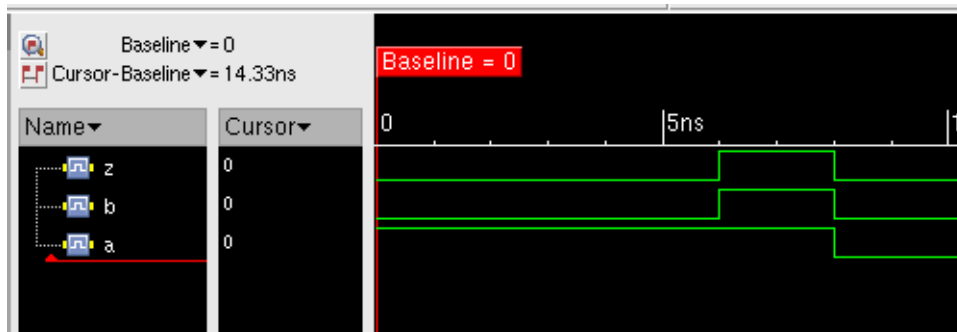


```
// Beispiel AND case 1
always @(a or b) begin
  case {a,b}
    2'b00: z=0;
    2'b01: z=0;
    2'b10: z=0;
    2'b11: z=1;
  endcase
end
```

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

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

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



- Blöcke in prozeduralen Zuweisungen können in Schleifen wiederholt werden
  - forever → Endloswiederholung
  - repeat → Wiederholungen gegebener Anzahl
  - while → Wiederholung solange Bedingung erfüllt
  - for → Klassische for loop



```
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;

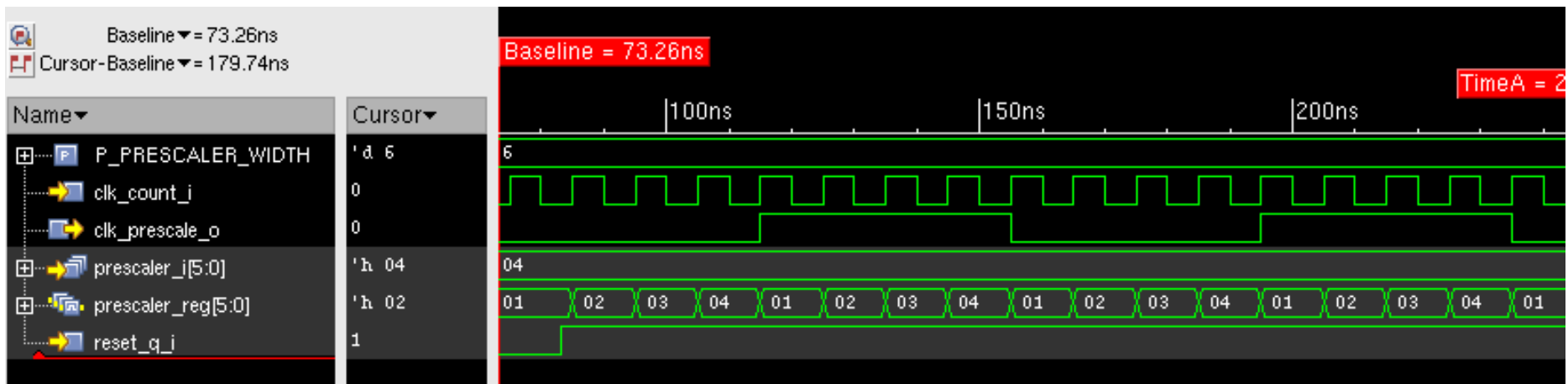
    reg [THERMOBITS-1:0] thermo_o;
    //thermo decoder block
    integer i;

    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
```

```
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;

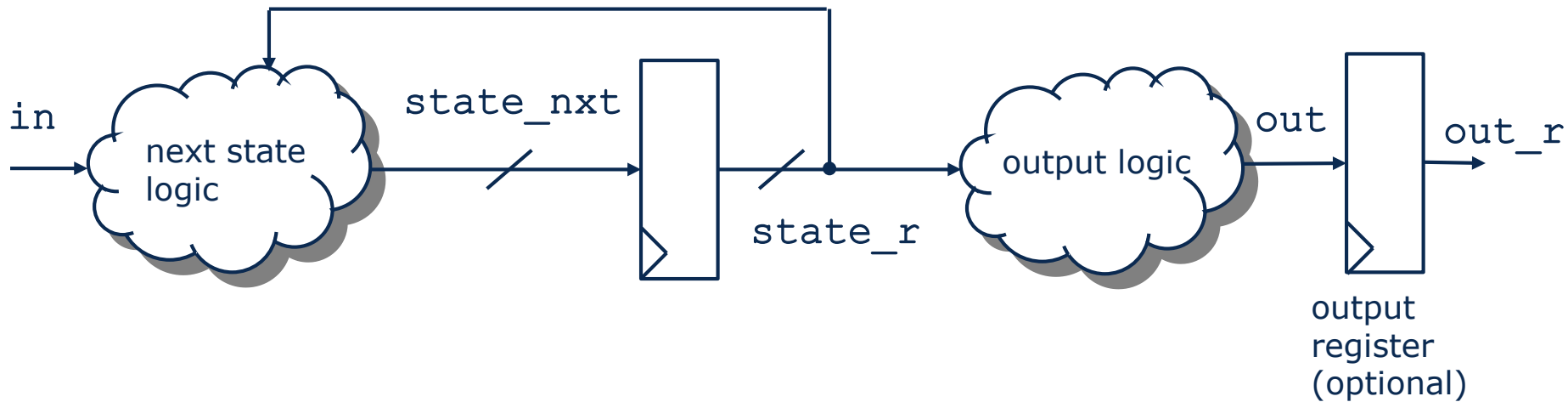
always @(posedge clk_count_i or negedge reset_q_i) begin
    if (reset_q_i == 1'b0) begin
        prescaler_reg<=1;
        clk_prescale_reg<=0;
    end
    else begin
        if (prescaler_reg==prescaler_i) begin
            clk_prescale_reg<=~clk_prescale_reg;
            prescaler_reg<=1;
       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

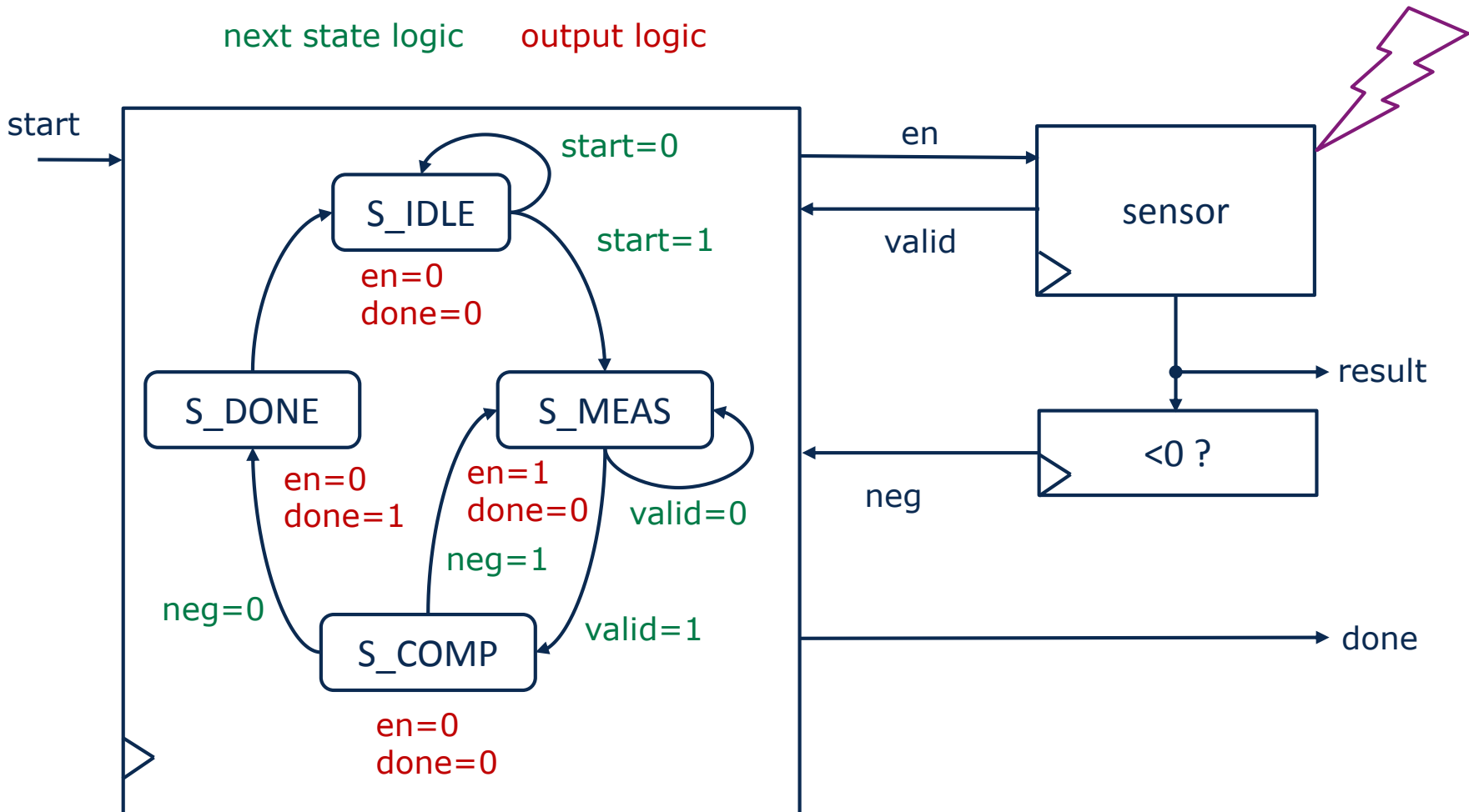


- Nutzung von always Blöcken möglich
- Vollständige Sensitivity Liste nötig
- Nutzung von Blocking Assignments (=)
- Default Zuweisungen oder vollständige Auscodierung von Control-Statements (if/else, case) zur Vermeidung von ungewollten Latches
- Zuweisen von kombinatorischen Signalen in nur **einem** always Block (keine konkurrierenden Zuweisungen)

- Nutzung von always Blöcken zwingend
- Flankensensitives Event Control Statement als Trigger (z.B. @(posedge clk))
- Asynchrones set/reset möglich
- Always Blöcke mit asynchronem set/reset müssen alle Register Signale im Reset Block und im funktionalen Block enthalten
- Nutzung von non-blocking assignments ( $\leq$ )
- Schreiben von Signalen in nur **einem** always block (keine konkurrierenden Zuweisungen)

- FSM: Finite State Machine
- Modelliert als Moore-Automat
  - Zustandsübergangsfunktion:  $state\_nxt = G(state\_r, in)$
  - Ausgabefunktion:  $out = F(state\_r)$





```
// Company           : tud
// Author            : hoepfner
// 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;
...

```



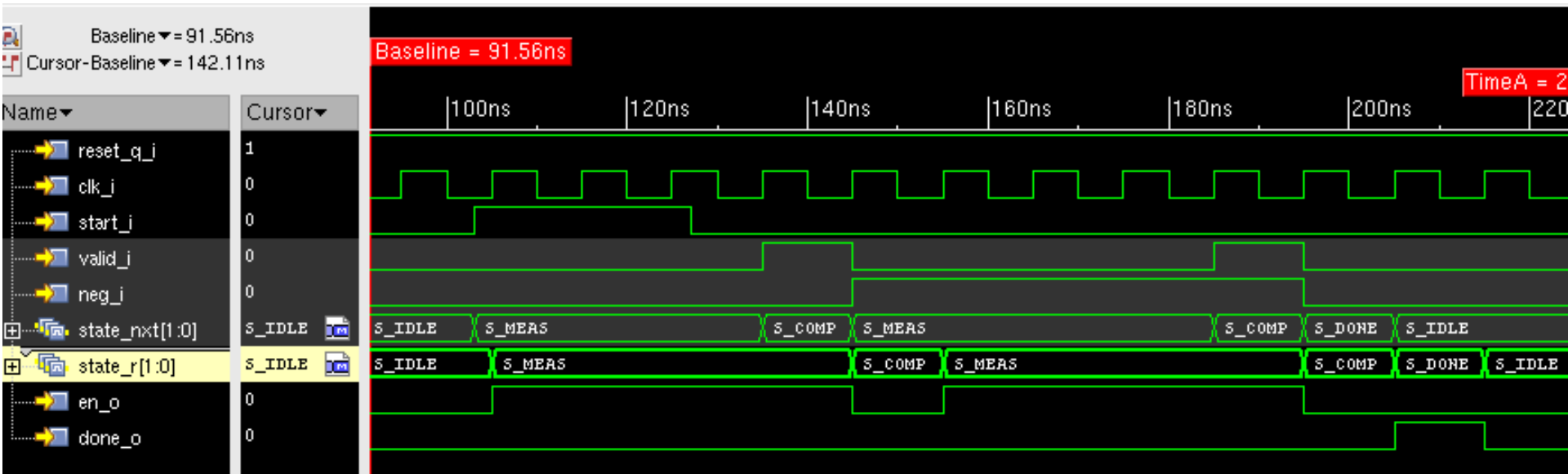
```
...  
parameter S_IDLE=0;  
parameter S_MEAS=1;  
parameter S_COMP=2;  
parameter S_DONE=3;  
  
//registers  
reg [1:0] state_r;  
  
//comb signals  
reg [1:0]          state_nxt;  
reg              en;  
reg              done;  
...
```

```
//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
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
...
```

```
...  
//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  
...
```

```
...  
//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
```



- Nutzung von Parametern zur Benennung der Zustände
- Separierung von kombinatorischen und sequentiellen Schaltungsblöcken
  - Register
  - Zustandsübergangsfunktion
  - Ausgabefunktion
- Keine Zustandsübergangsfunktionen im sequentiellen Teil beschreiben
  - Ausnahme: synchrones Reset
- Separate Zuweisung der Ausgangssignale auf die „outputs“

# Verilog – Hierarchisches Design

- Verilog bietet die Möglichkeit zur Partitionierung des Designs und zur Wiederverwendung von Designelementen
  - Module
  - Tasks
  - Funktionen
- Einbinden von Code Blöcken mit File Includes ( `'include` )



## Deklaration

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

## Instanziierung

```
...  
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)  
);
```

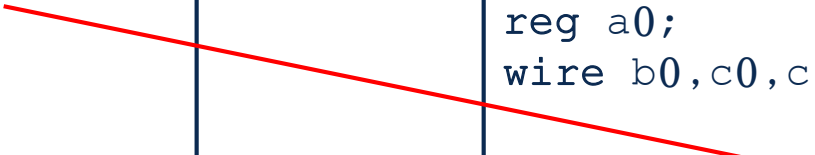
- Module können instanziiert werden
- Inputs können von „reg“ und „wire“ Datentypen getrieben werden
- Outputs werden auf „wire“ Datentypen zugewiesen (kontinuierliche Zuweisung)

## Deklaration

```
module my_module (a, b, c);  
  
    parameter P0=1;  
    parameter P1=2;  
  
    input a,b;  
    output c;  
    ...  
endmodule
```

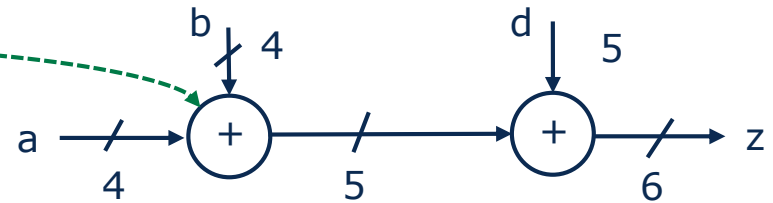
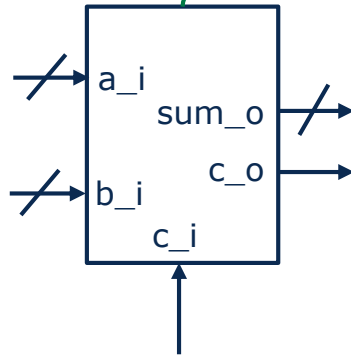
## Instanziierung

```
...  
reg a0;  
wire b0,c0,c1;  
  
my_module #(.P0(2), .P1(3))  
    my_module_i0 (  
        .a(a0),  
        .b(b0),  
        .c(c0)  
    );
```



- Parameter von Modulen können bei der Instanziierung überschrieben werden
- Verschiedene Instanzen können individuell Parametrisiert werden

adder



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

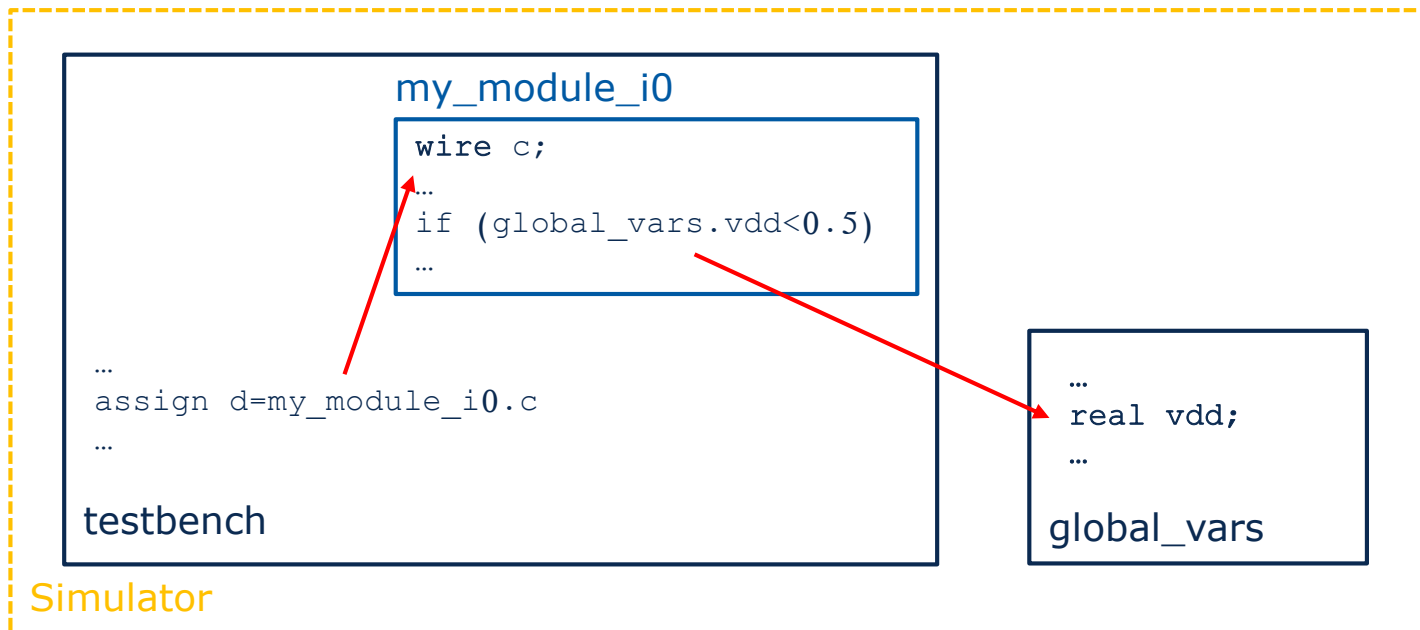
  input [C_DWIDTH-1:0] a_i, b_i;
  input c_i;
  output [C_DWIDTH-1:0] sum_o;
  output c_o;

  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};
```

- Signale in Verilog sind global verfügbar
- Instanz-Hierarchien werden mit „ . „ im Signalnamen getrennt.
- **Nicht synthesegerecht!**
- Anwendungen:
  - Monitoring von Signalen in Testbenches zum Debugging und zur Verifikation
  - Für globale Netze die keine Signalpins sind (z.B. Versorgungsspannungen)
- **ACHTUNG: Signale, die nicht über die Ports laufen sind ist auch in der implementierten Schaltung von außen nicht schreibbar/lesbar!**



## 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



zufällige  
Initialisierung

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

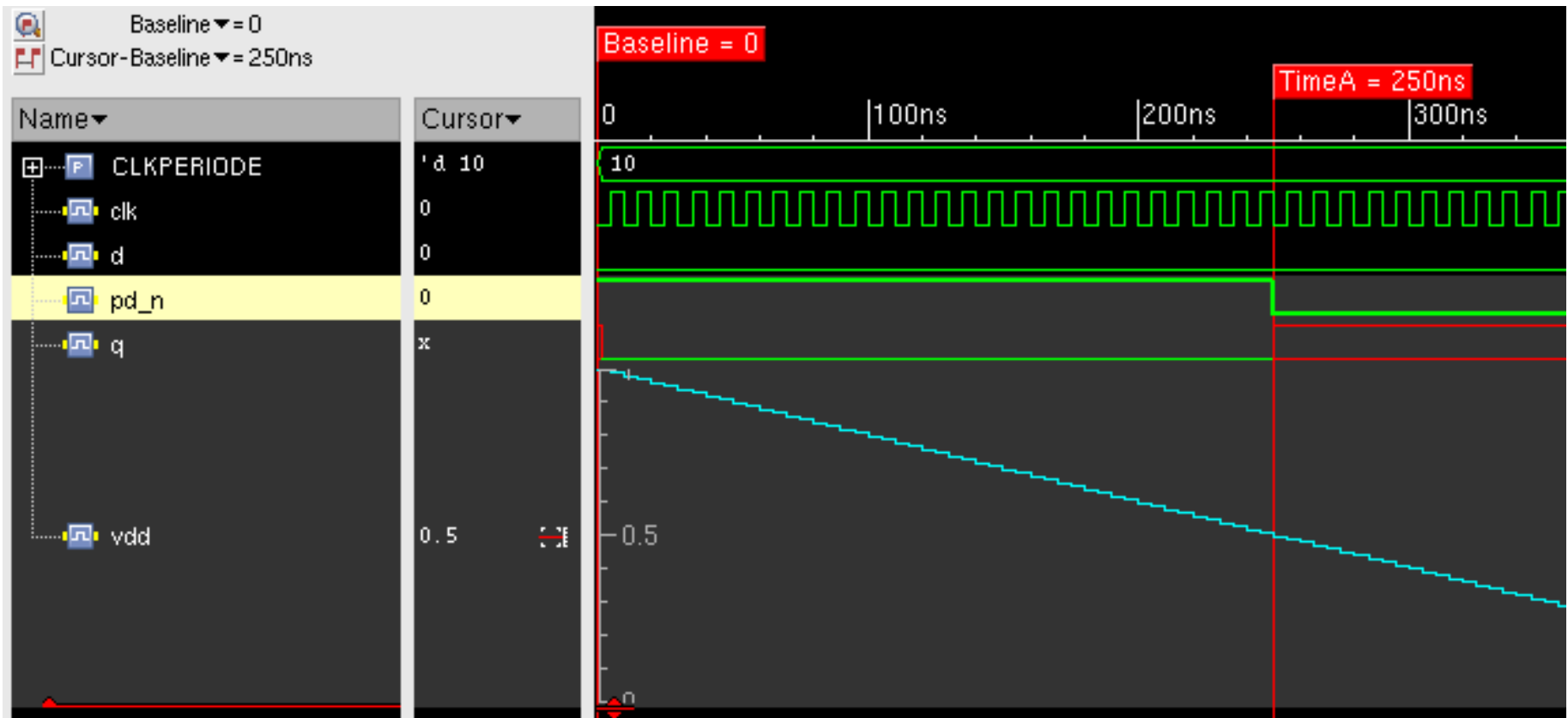
input d, cp;
output q;

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
```



- Funktionen können komplexe Zuweisungen kapseln:
  - → übersichtlicher Code
  - → Wiederverwendbarkeit
- Funktionen haben **einen** Rückgabewert.
- Eine Funktionsdefinition muss eine **Zuweisung** auf die Funktion beinhalten.
- Funktionen werden in **einem** Zeitschritt ausgeführt
- Eine Funktionsdefinition darf **keine zeitlichen Abläufe** (Delays, Event Control Statements) enthalten.
- Funktionen können keine Tasks aufrufen.
- Eine Funktion muss **mindesten ein** Input Argument haben.
- Eine Funktionsdefinition darf **keine** Argumente der Typen **output** oder **inout** besitzen.
  
- → Funktionen eignen sich zur Beschreibung kombinatorischer Logik

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

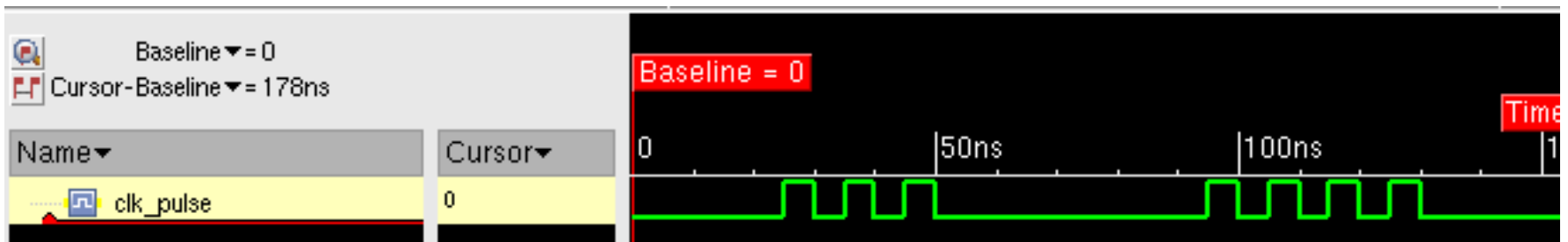


- Tasks können komplexe Zuweisungen einschließlich zeitlicher Abfolgen kapseln:
  - → übersichtlicher Code
  - → Wiederverwendbarkeit
- Tasks können **beliebig viele Inputs und Outputs** haben.
- Task können **zeitliche Abläufe** (Delays, Event Control Statements) enthalten.
- Tasks können Funktionen und andere Tasks aufrufen.
- Tasks können auf globale Variablen zugreifen.
  
- Tasks eignen sich zur Beschreibung kombinatorischer Logik, sequentieller Logik und Stimuli

```
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
```



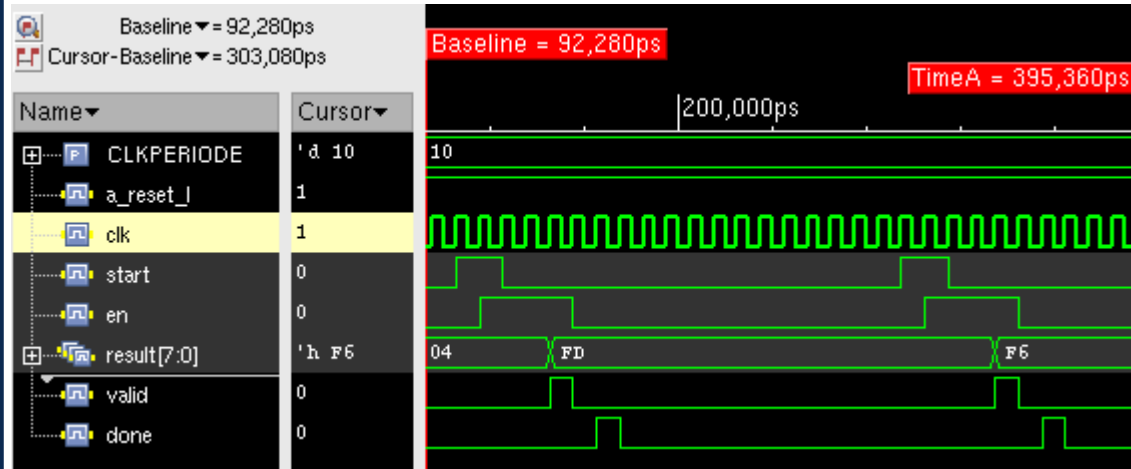
- Starten der FSM aus Beispiel ab Folie 81

```

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
  
```



- Funktionen und Tasks müssen in dem Modul definiert werden, in dem sie genutzt werden.
- File includes erlauben die Verwendung vordefinierter Code-Fragmente

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

    `include "my_tasks.v"

    `include "my_functions.v"

    `include „stim_clk_gen.v“

    initial begin
        ...
        //testcase code
        ...
    end
endmodule
```

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

function my_func2;
...
endfunction
```

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

task my_task2;
...
endtask
```

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

- Verilog besitzt eingebaute Systemfunktionen und -tasks
- Hier einige wichtige:
  - `$display`
    - Formatierte Ausgabe auf die Kommandozeile
  - `$fopen`, `$fwrite`, `$fflush`, `$fclose`
    - Schreiben in eine Datei
  - `$readmemh`, `$readmemb`
    - Laden eines Speichers aus einer Textdatei
  - `$random`
    - Erzeugen einer Zufallszahl
  - `$dist_normal`
    - Erzeugung einer Gauss-Verteilten Zufallszahl
  - `$realtime`
    - Rückgabe der aktuellen Simulationszeit als real
  - `$finish`
    - Beenden der 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

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

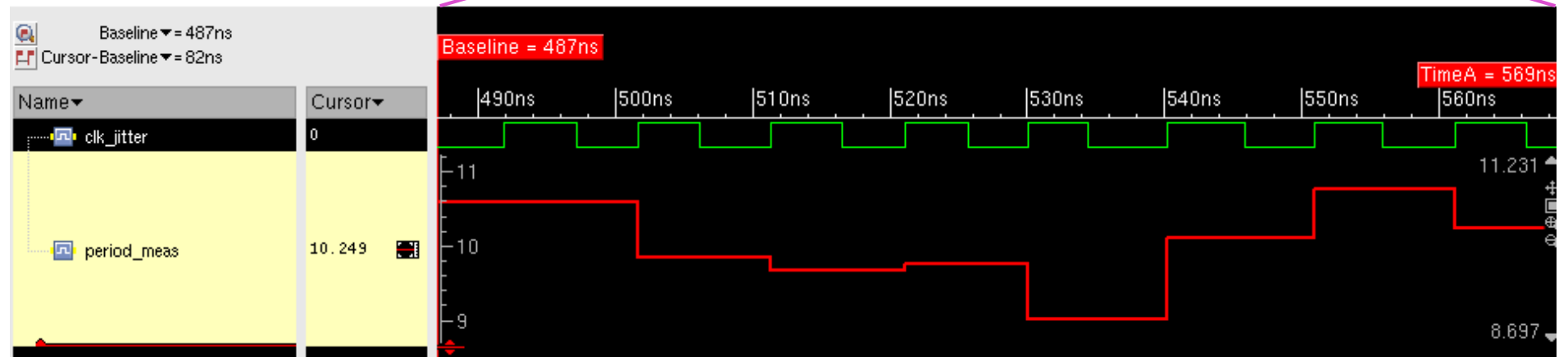
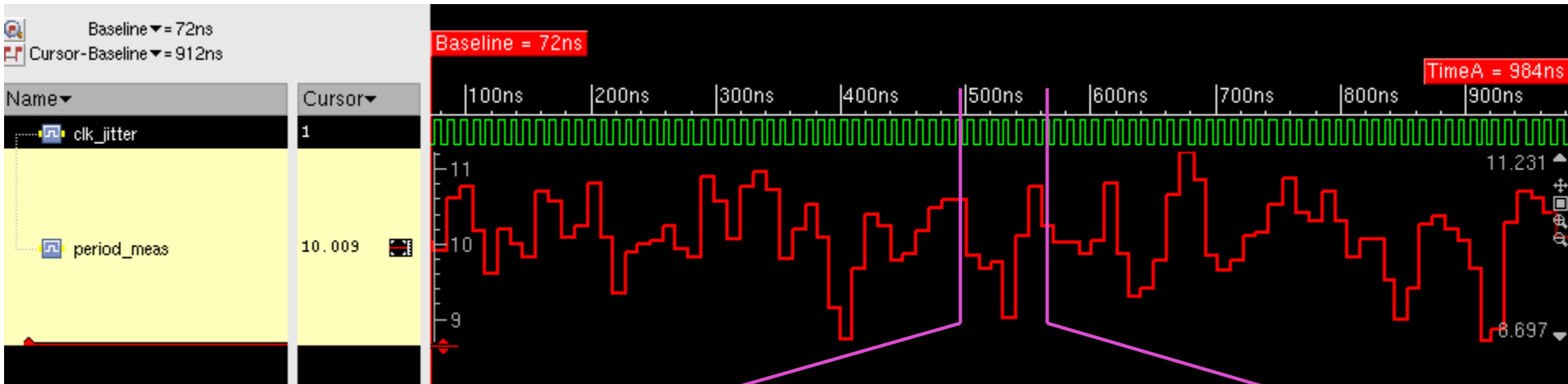
reg clk_jitter=0;
real period_jitter=0;
real period=CLKPERIODE;
integer seed=NOISE_SEED, mean=0, stdev=1000000, random_value_int;

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

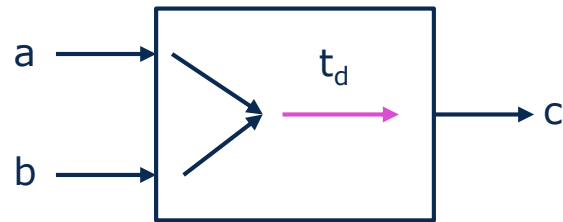
always #(period/2) clk_jitter=~clk_jitter;

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
```





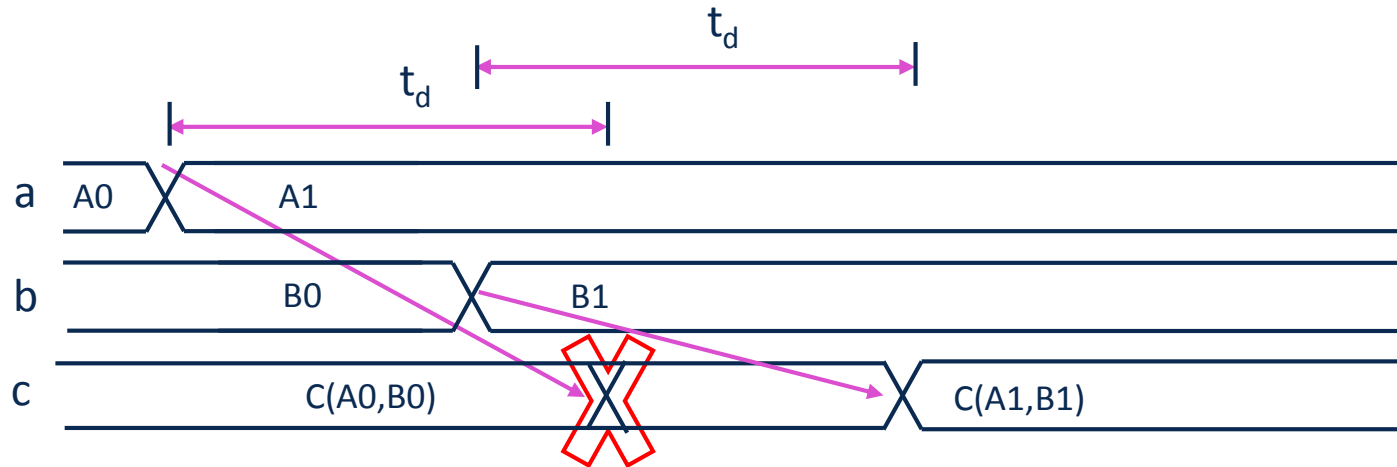
# Verilog – Modellierung von Verzögerungszeiten



- Abstraktion der Verzögerungszeiten von kombinatorischen Logikblöcken
- Zuweisung von Delays  $t_d$  für Timing Arcs
- Berücksichtigung bei der digitalen Schaltungssimulation
- Modellierung zur Simulation als
  - **Inertiales Delay** oder **Transport Delay**

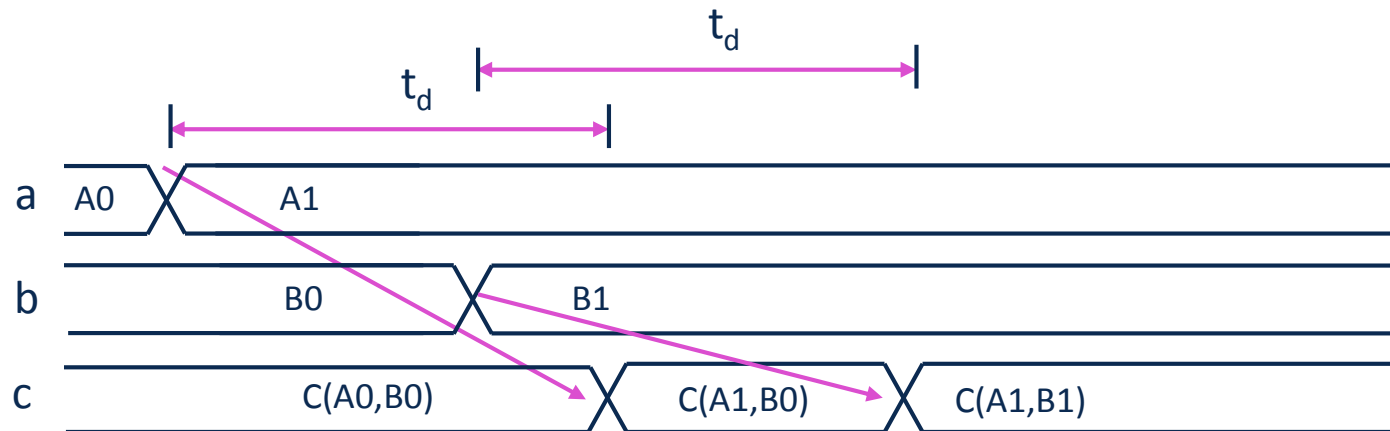
- **Inertiale Delays:**

- Bei Eingangssignaländerung nach  $t < t_d \rightarrow$  Generierung eines neuen Events, Verwerfen des vorherigen Events
- Signal-Propagierung zu einem Ausgangssignal **nachdem** die Eingangssignale stabil sind

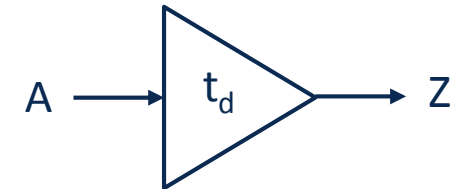
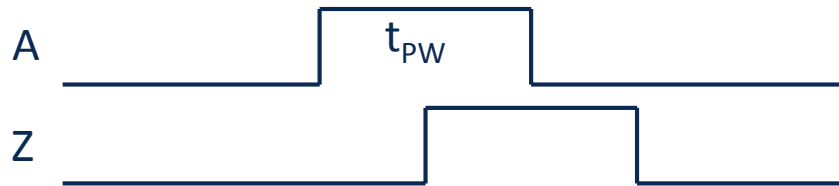


- **Transport Delays:**

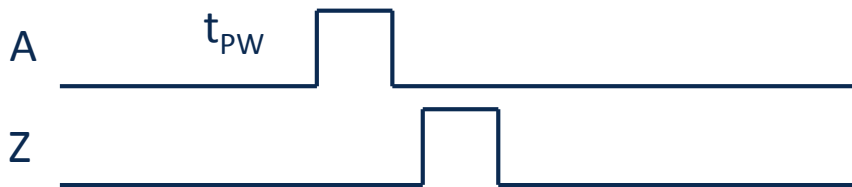
- Propagierung **aller** Eingangsänderungen an den Ausgang
- Erzeugung von neuen Events bei jeder Eingangssignaländerung
- Nachteile bei Modellierung kombinatorischer Logik:
  - Propagierung von auch kurzer Glitches
  - Erzeugen vieler Events → langsamere Simulation



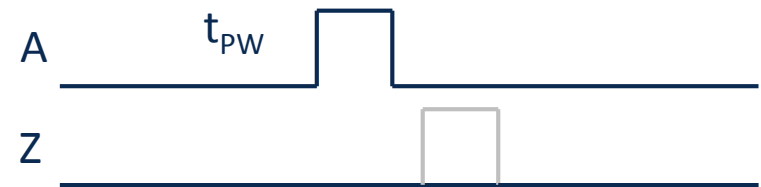
$t_{PW} > t_d$



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

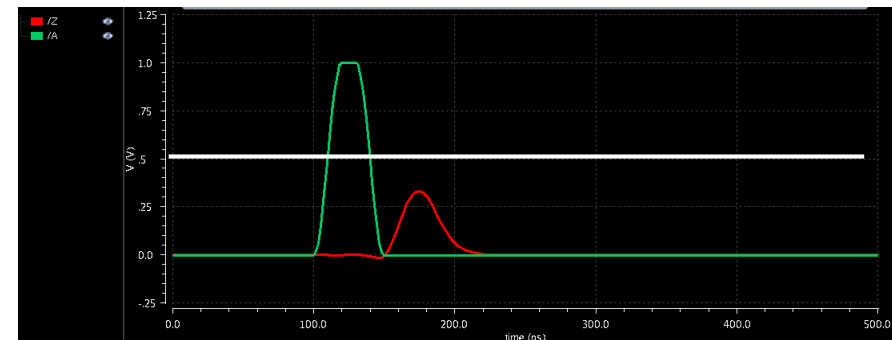
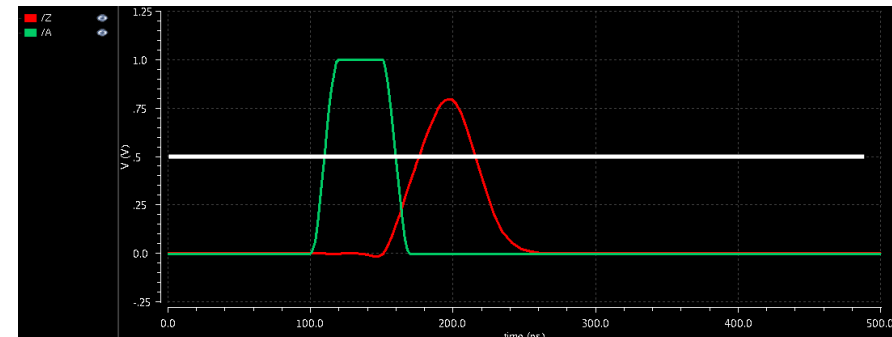
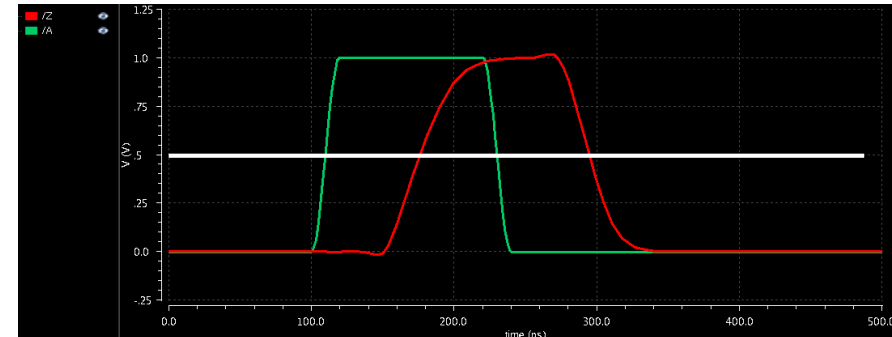


$t_{PW} < t_d$ , Inertiales Delay Modell



- Typischer Fehler:
  - Default Delay Zuweisungen in Gattern (z.B. 1ns)
  - → Puls von  $< 1\text{ns}$  verschwindet!

- Simulation von realen Signalverläufen (Transistor-Level)
- CMOS Buffer Kette
  - 1. Pulsweite: 120ps,  $t_{d,rise}$  65ps
  - 2. Pulsweite: 50ps,  $t_{d,rise}$  65ps
    - Puls am Ausgang
    - → Intertial Delay Modell nicht gültig
  - 3. Pulsweite: 30ps, kein Z-Puls
    - → Transport Delay Modell nicht gültig
- Intertial und Transport Delay Modelle sind idealisierte Annahmen
- Ausreichend für Gate-Level Simulationen
- Das reale Delay ist abhängig von der inneren Schaltung, welche hier abstrahiert ist!
- → CMOS Schaltungen



- Zuweisungen können mit Verzögerungszeiten versehen werden.  
→ #
- `timescale <unit>/<resolution>
  - <unit>: Maßeinheit der Zeitangaben
  - <resolution>: Zeitauflösung in Simulation

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

- Zuweisungen mit Delay (#) sind nicht synthetisierbar!

- Verzögerungszeiten können angegeben werden:
  - bei prozeduralen Zuweisungen der rechten (RHS) und linken Seite (LHS)
  - bei kontinuierlichen Zuweisungen auf der linken Seite (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;
```

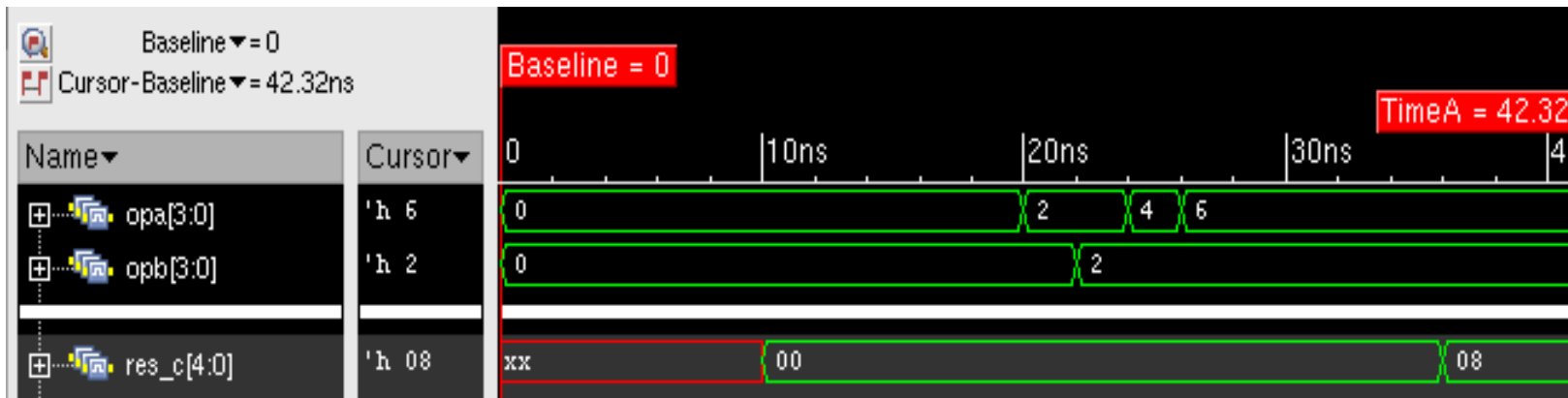


- Modellierung eines inertialen Delays

```
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<br>new event 4 → res_c @32 |
| 24     | cancel event 4 → res_c @32<br>new event 6 → res_c @34 |
| 26     | cancel event 6 → res_c @34<br>new event 8 → res_c @36 |
| 36     | 8 → res_c   |



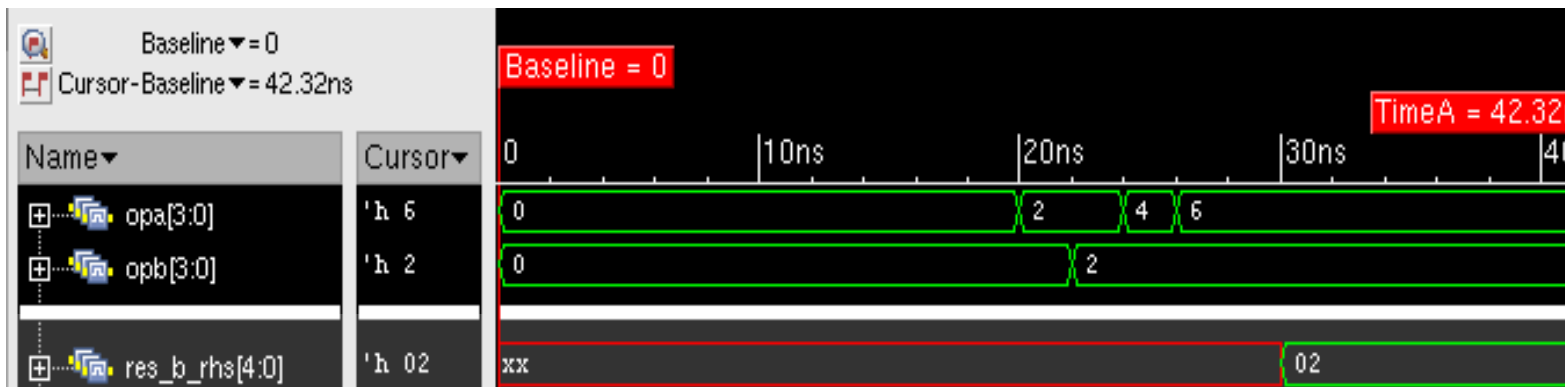
```

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<br>new event 2 → res_b_rhs @30<br>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  |



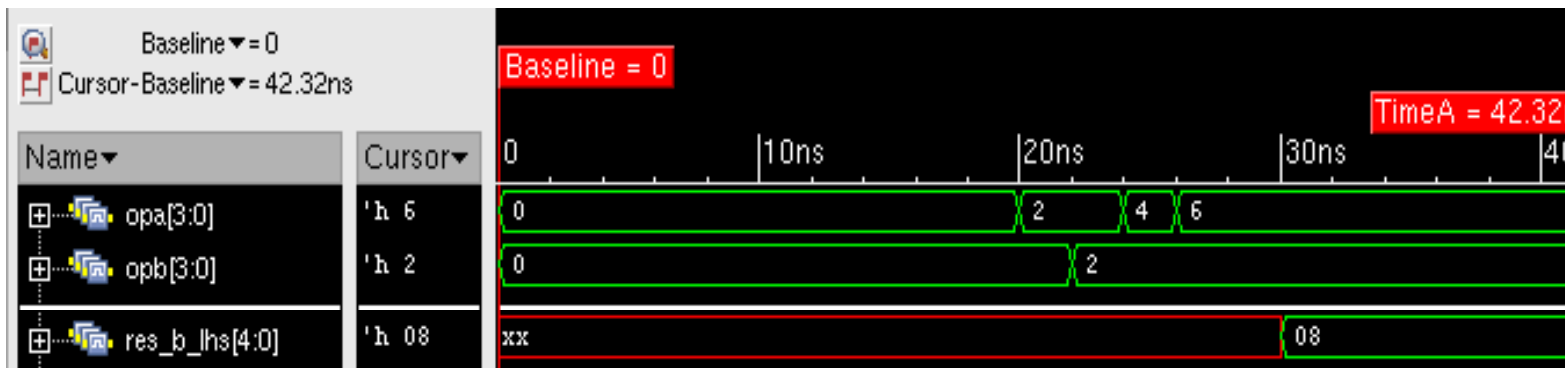
```

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<br>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<br>8→ res_b_rhs |

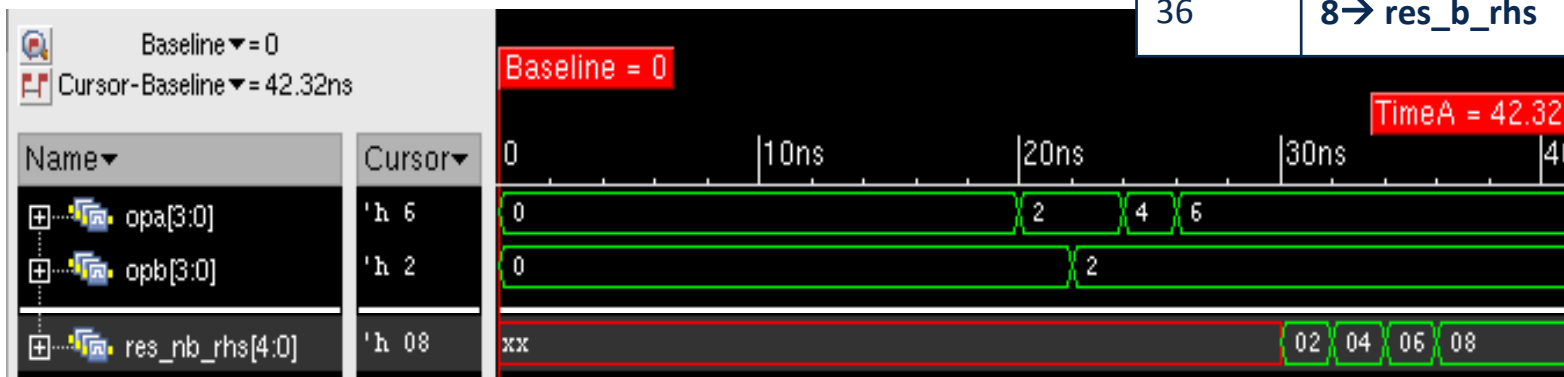


- Modellierung eines Transport Delays

```
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<br>new event 2→res_nb_rhs @30 |
| 22     | trigger always block<br>new event 4→res_nb_rhs @32 |
| 24     | trigger always block<br>new event 6→res_nb_rhs @34 |
| 26     | trigger always block<br>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                                       |



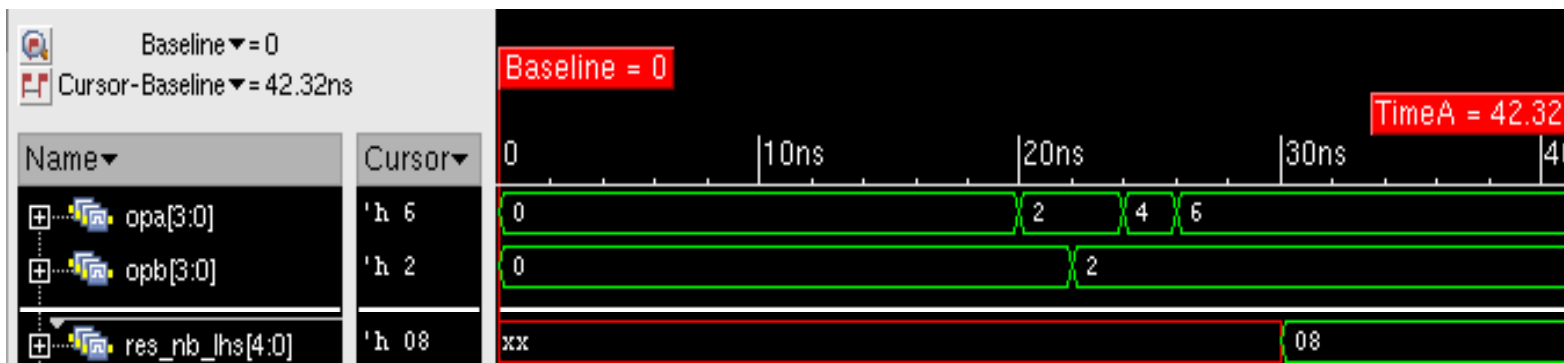
```

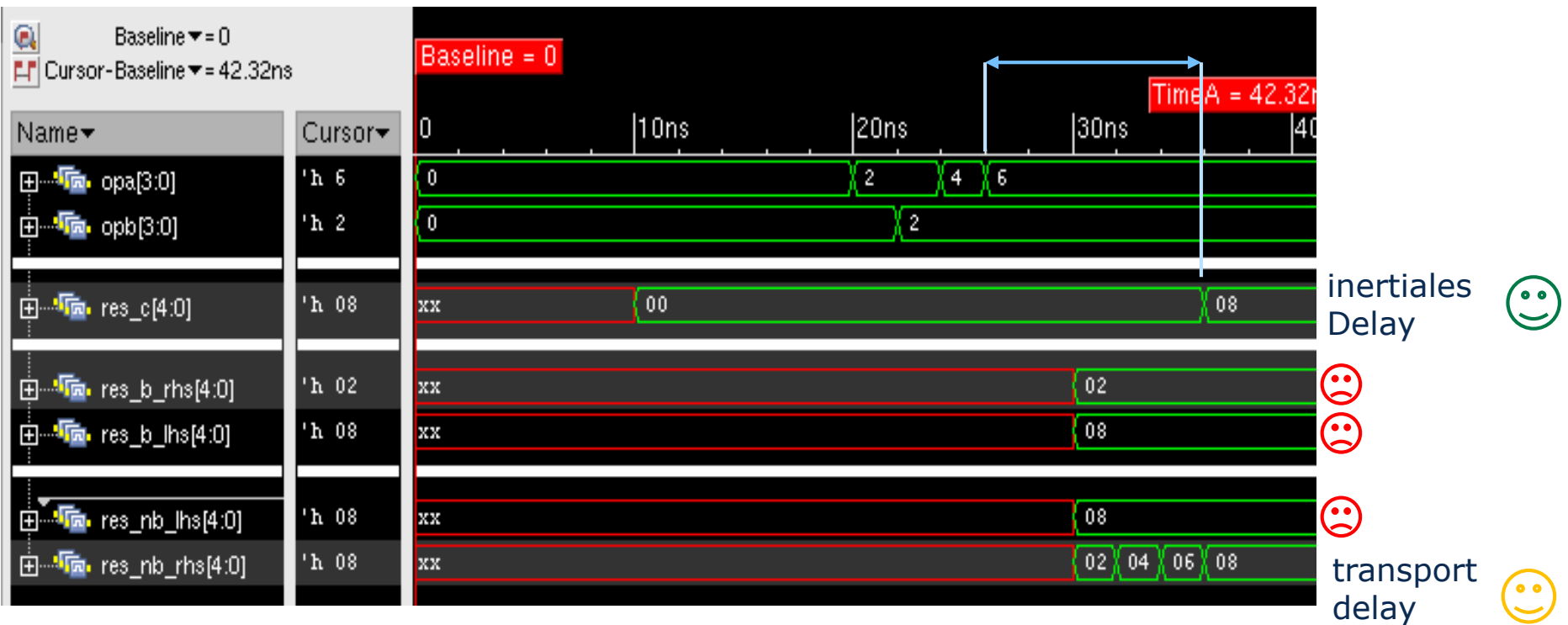
reg [3:0] opa, opb;
reg [4:0] res_b_rhs, res_nb_rhs, res_b_lhs, res_nb_lhs;
wire [4:0] res_c;
...

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

```

| t [ns] |  |
|--------|--|
| 20     | trigger always block<br>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<br>8→ res_b_rhs |





| Delay                       | Modelle  | Testbench                             |
|-----------------------------|--|---------------------------------------|
| Continuous Assignment RHS   | OK, korrekte Inertial-Delay Modellierung   |                                       |
| Blocking Assignment LHS     | Nicht OK   | OK, zeitliche Abläufe/Stimuli         |
| Blocking Assignment RHS     | Nicht OK   | Nicht OK                              |
| Non-Blocking Assignment LHS | Nicht OK   | Nicht OK                              |
| Non-Blocking Assignment RHS | Transport Delays,<br><b>ABER: Non-Blocking Assignments für kombinatorische Blöcke nicht OK</b> | OK, Modellierung von Transport Delays |

- Empfohlene Methode zur Modellierung kombinatorischer Logik mit Delay:
  - **Zero-Delay** always block (blocking assignment)
  - Zuweisung des Delays in kontinuierlicher Zuweisung (Intertiales 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;
```

- Details unter:

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



- Komplexe Delay Zuweisung für Gate-Level Simulation mit **specify** Statement
  - Separate Rising/Falling Delays
  - Sequentielle Modelle (z.B. posedge CLK → Q Delay)
- Zuweisung von Delays als **Inertiale Delays** für einzelne **Timing Arcs**
- Annotieren der Timings typischer Weise nach Synthese und Place&Route als Ergebnis der Statischen Timing Analyse (STA)
- Zusätzlich Angabe von Constraints möglich
  - Setup & Hold Zeiten \$setuphold()
  - Minimale Pulsweite \$width()

```
`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
```

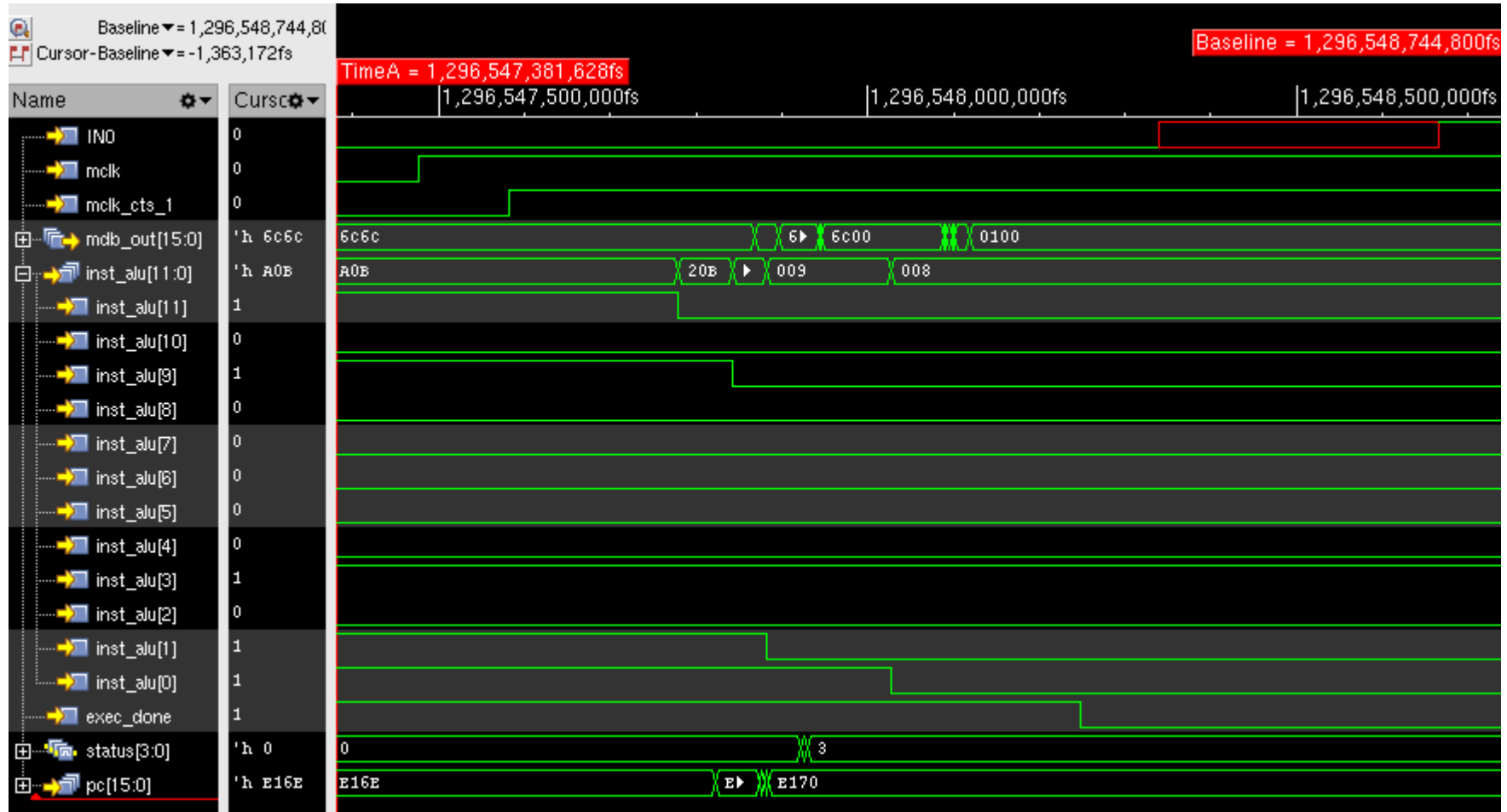
```
`celldefine
module H_DFPQX1( D, CP, Q );
input CP, D;
output Q;
reg notifier;
wire D_delay ;
wire CP_delay ;

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

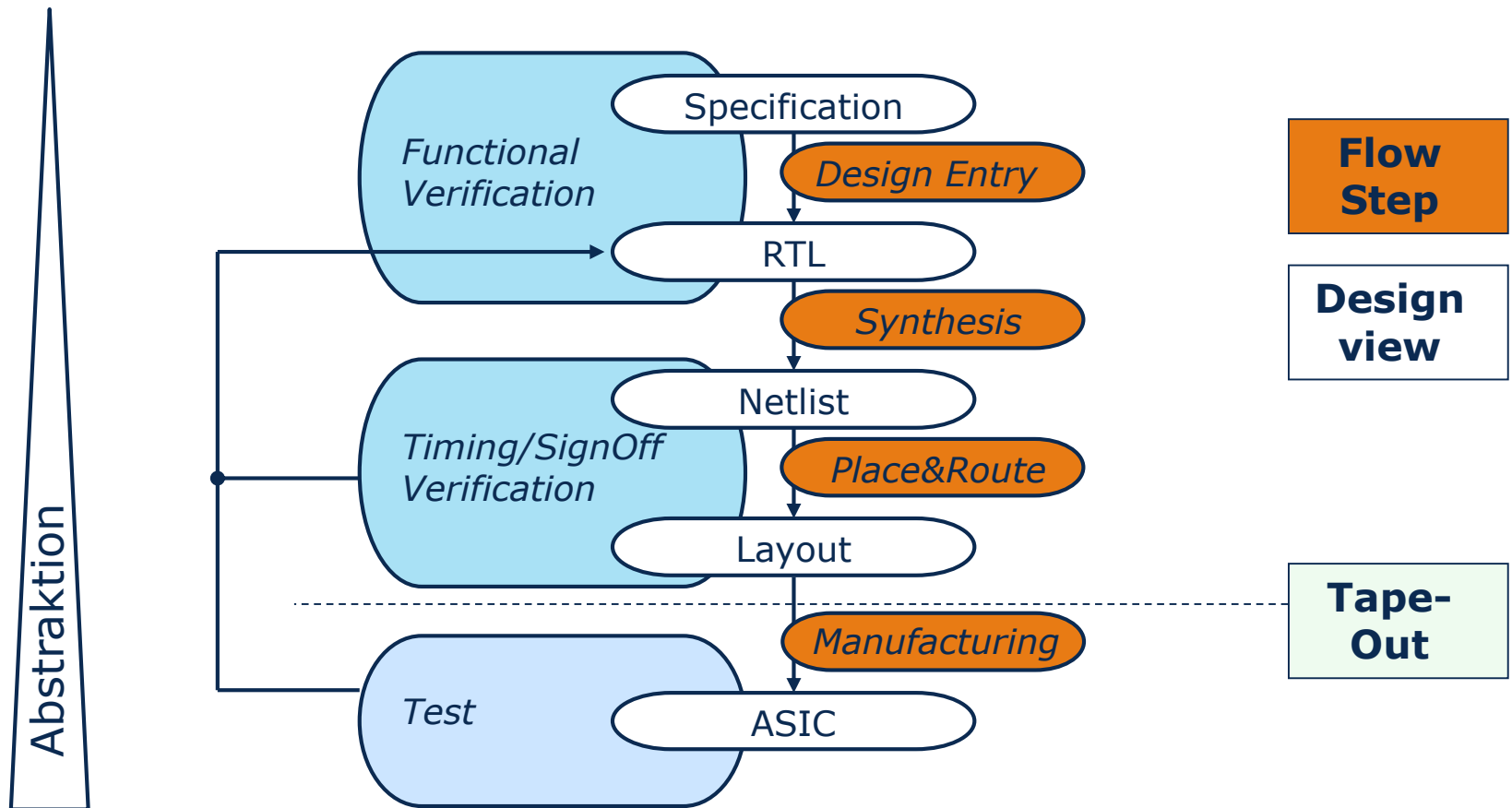
    specify
        (posedge CP => (Q : D)) = (1.0,1.0); // seq arc CP --> Q

        $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_1h
        $width(negedge CP,1.0,0,notifier); // mpw CP_hl
    endspecify

endmodule
`endcelldefine
```



# Verifikation



- Nachweis der Übereinstimmung von Schaltungsfunktionalität und Spezifikation
- ggf. Vergleich der Übereinstimmung mit einem Referenzmodell
- Verifikation findet in verschiedenen Hierarchieebenen statt
  - Block → Modul → System
- Verifikationsmethoden
  - **Verifikation durch Schaltungssimulation**
  - Formale Verifikation
  - Hardware Emulation/Prototyping (z.B. mit FPGAs)
- Bei aktuellen System-on-Chip Designs werden **bis zu 80%** der Entwurfszeit- und Ressourcen zur Verifikation verwendet

```

module testbench ()

  //Signals
  reg ...
  wire ...

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

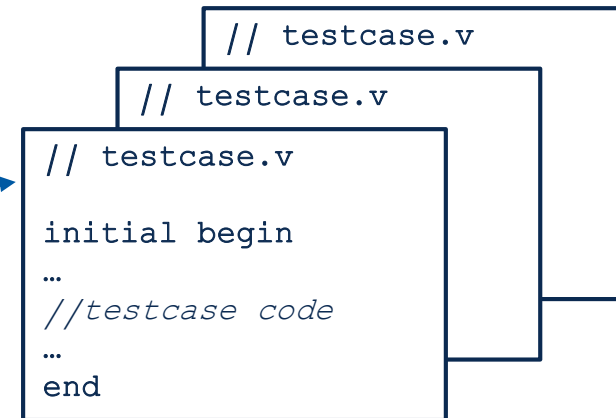
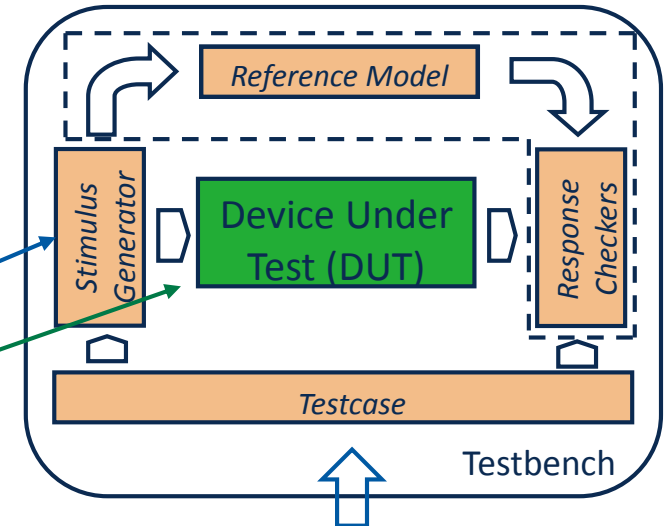
  `include „stim_clk_gen.v“

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

  `include „testcase.v“

end
endmodule

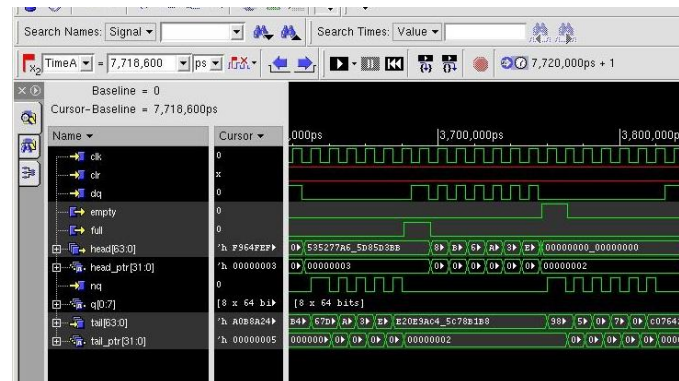
```



individuelle  
Simulationsverzeichnisse



- Nutzung von Waveforms zur Verifikation Schaltungsfunktionalität
  - Vorteile 😊
    - Intuitiver Ansatz
    - Visualisiert das Verhalten der Schaltung
    - Vermittelt Verständnis zur Funktion des Schaltung
    - Erleichtert Debugging
  - Nachteile ☹️
    - Sehr zeitaufwändig bei komplexen Systemen
    - **Nicht automatisierbar** und **reproduzierbar**
    - Grad der Verifikationsabdeckung nicht messbar



- In der Beschreibung von Schaltungsblöcken können **Bedingungen** für das erwartete Schaltungsverhalten formuliert werden
  - Einbringen von detaillierten Spezifikationen auf RTL-Level möglich
  - Verifikation wird bereits bei der RTL Implementierung unterstützt
- Prüfung der Bedingungen findet nur im Schaltungsmodell statt, nicht in der realisierten Hardware
- Nutzung von **nicht synthesefähigen** HDL Konstrukten
- Überprüfung dieser Bedingungen bei der **Simulation**
- Ausgaben bei Verletzung der Bedingungen

```

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

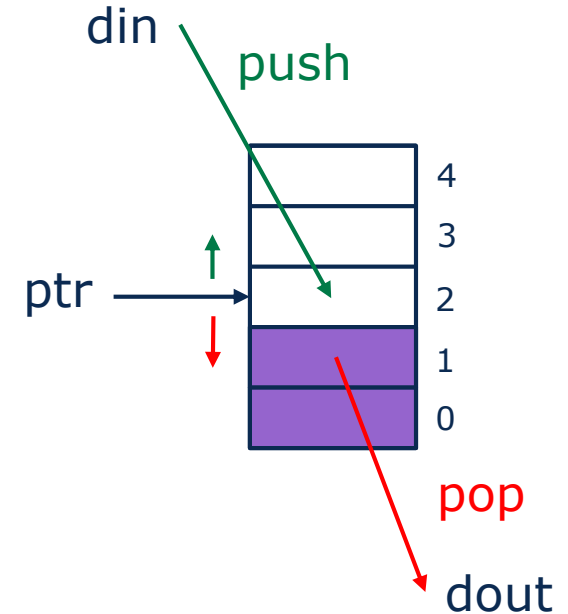
parameter DBITS=8;
parameter PTRBITS=3;
parameter DEPTH=5;
input  reset_q_i, clk_i, push_i, pop_i;
input  [DBITS-1:0]    din_i;
output [DBITS-1:0]    dout_o;

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

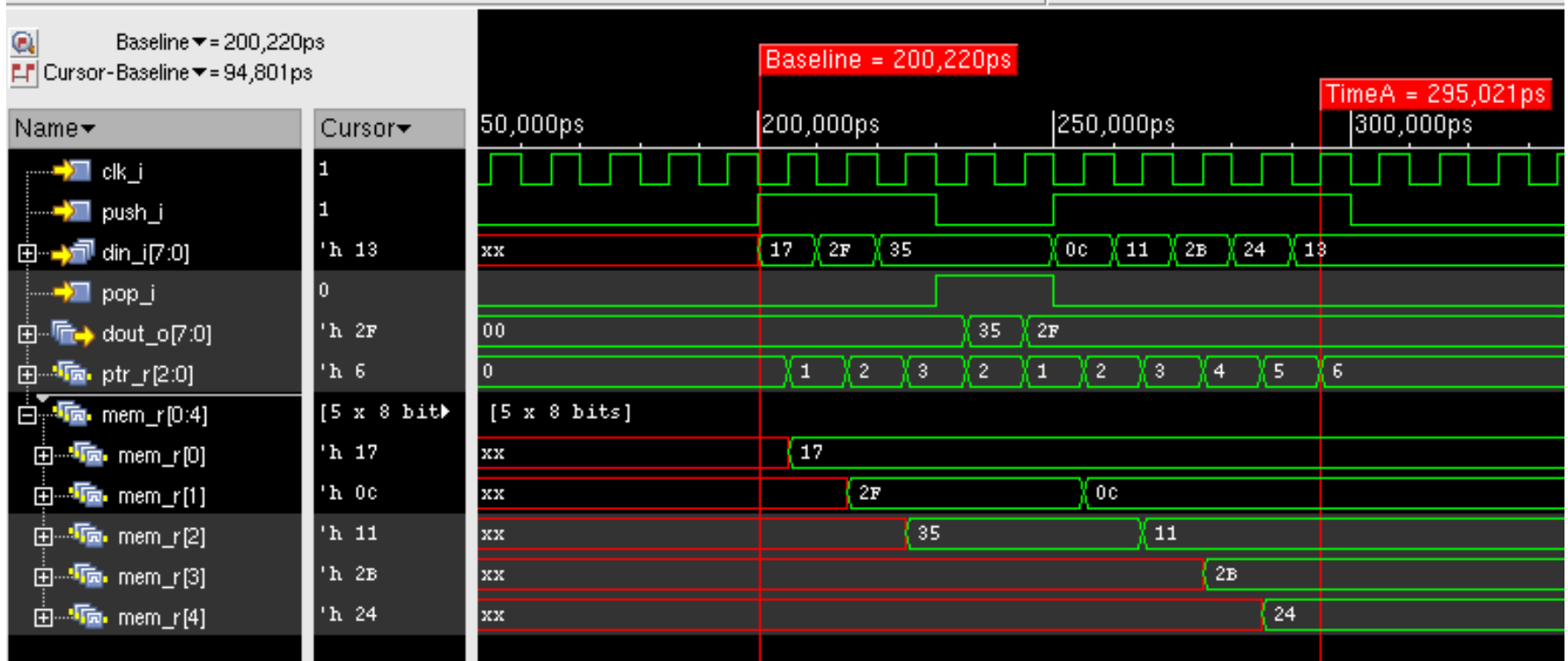
wire [PTRBITS-1:0] ptr_m1, ptr_p1;
assign ptr_m1=ptr_r-1;
assign ptr_p1=ptr_r+1;

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

```



```
...
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_pl;
    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_ml;
      do_r<=mem_r[ptr_ml];
    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
```



```
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
```

- Check: Überprüfen einer Schaltungsausgabe
- Vergleich mit
  - Erwartungswert
  - Spezifikation
  - Referenzmodell
- Formulierung in der Testbench oder im Testcase
- Zählen der Checks und der fehlgeschlagenen Checks (errors)
- Generierung eines finalen Reports
  
- Vorteile:
  - Schnelle Wiederholbarkeit der Verifikation bei Entwurfsänderungen

```
//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
...
```

- Implementierung des „Referenzmodells“ als Tabelle

```
//Beispiel Zustandsübergangstabelle
...
//Instanz der Zustandübergangslogik
...

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



- 
- Bereitstellen der Eingangsdaten
    - Memory Laden
    - Setzen von Inputs
  - Starten des Schaltungsblocks
  - Warten bis fertig
  - Abholen der Ausgangsdaten
    - Speicher lesen
  - Ergebnisse Vergleichen
    - Vergleich mit Referenzdaten

```
//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
```

```


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

```

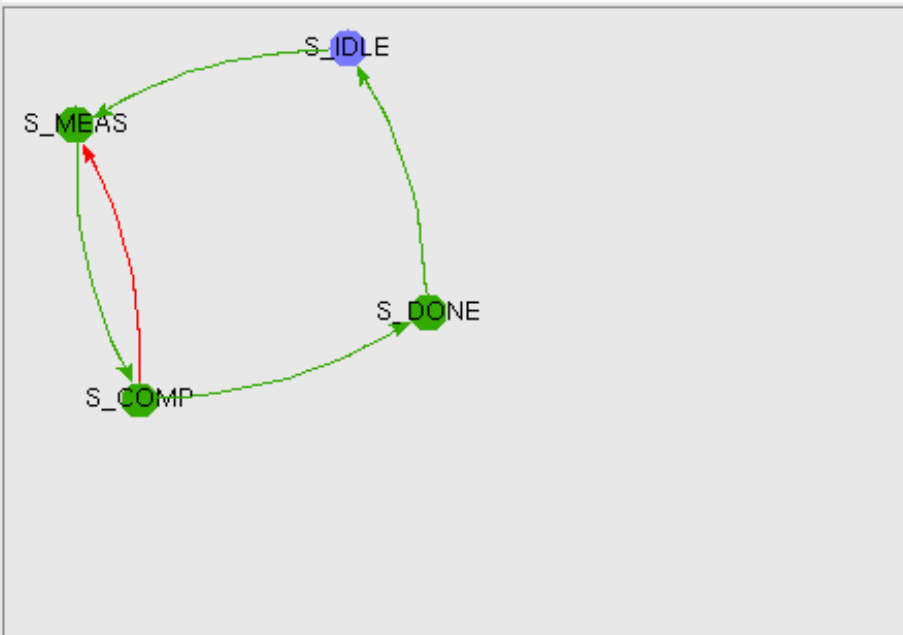
- Messung der Abdeckung des Quellcodes durch die Simulation
  - Block Coverage
    - Abdeckung von Code Blöcken (begin ... end)
  - Expression Coverage
    - Abdeckung von Ausdrücken
  - Toggle Coverage
    - Abdeckung von Signalwechsell
  - FSM Coverage
    - Abdeckung von Zuständen (state) und Zustandsübergängen (arc)
- **ACHTUNG: Die Coverage Analyse prüft nicht ob die Ausgabe der Schaltung geprüft wurde.**

ICC - FSM Coverage Details for tb\_verilog\_tutorial.i\_fsm.state\_r

File View Layout Navigate Window Help cadence™

Navigate: Uncovered State Zoom:  Threshold 100 %

| Name                              | State      | Arc       |
|-----------------------------------|------------|-----------|
| tb_verilog_tutorial.i_fsm.state_r | 100% 4 / 4 | 80% 4 / 5 |



```

graph TD
    S_IDLE((S_IDLE)) --> S_MEAS((S_MEAS))
    S_MEAS --> S_COMP((S_COMP))
    S_COMP --> S_DONE((S_DONE))
    S_DONE --> S_IDLE
    S_MEAS -.-> S_COMP
        
```

Line | File: /home/hoepner/CRCO/icc/c\_verilog/units/verilog\_tutorial/course/tl/verilog/fsm\_u






Test:

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

```

File: /home/hoepfner/ICPRO/p_ice/s_verilog/units/verilog_tutorial/source/rtl/verilog/fsm.v
55         end
56     end
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
66         if (neg_i==1'b1) begin
67             state_nxt=S_MEAS;
68         end
69         else begin
70             state_nxt=S_DONE;
71         end
72     end
73     S_DONE: begin
74         state_nxt=S_IDLE;
75     end

```

Coverage Report: Uncovered Blocks  Marking:    

```

Instance name: tb_verilog_tutorial.i_fsm
Module/Entity name: fsm
File name: /home/hoepfner/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 |

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

| Line | Code                     |
|------|--------------------------|
| 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            |
| 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/hoepfner/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

```

Coverage Report: Uncovered Toggles  Marking: 

```

Instance name: tb_verilog_tutorial.stack_i
Module/Entity name: stack
File name: /home/hoepfner/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]   |

- Vollabdeckende Simulation
  - Simulation aller Eingangsvektoren bei kombinatorischen Schaltungen:  
n-Inputs  $\rightarrow 2^n$  Checks
    - z.B. alle Eingangskombinationen der FSM Logik
  - Vorteile:
    - Sehr einfacher Testcase bei vollabdeckender Verifikation
  - Nachteile:
    - Lange Laufzeit bei vielen Inputs
    - Schwierig bei sequentiellen Schaltungen
- Guided Testcases
  - Explizites Stimulieren der Schaltung, insbesondere von Grenzfällen („corner cases“)
  - Vorteile:
    - Hohe Coverage möglich
  - Nachteile:
    - setzt detaillierte Kenntnis der Schaltung voraus
    - große Anzahl von Testcases  $\rightarrow$  aufwändiges Setup



- Random Testcases
  - Zufälliges Stimulieren der Schaltung
  - Einschränken des Zufallsbereichs auf zulässige Werte und Abfolgen („constrained random“)
    - z.B. deterministisches Starten (`start_i`) einer sequentiellen Schaltung bei zufälligen Eingangswerten (`din_i`)
    - Vorteile:
      - wenige Testcases für hohe Coverage
      - geeignet für kombinatorische und sequentielle Schaltungen
      - Simulationslaufzeit erhöht Coverage
    - Nachteile:
      - Corner Cases sehr unwahrscheinlich
- **→ Praktisch meist Kombination aus mehreren Methoden verwendet.**

- **Komplexe Designs haben viele Testcases**
- Regressions
  - Self-Checking Testcases
  - Automatisierte Simulation
  - Parallelisierung möglich
  - Automatisch erzeugter Report
  - Coverage Analyse
- Vorteile:
  - Automatisierte Verifikation
  - „Messung“ des aktuellen Verifikationsstatus → „Management Report“
  - Schnelle Wiederholung der Verifikation bei
    - Entwurfsänderungen
    - Bug-Fixes
    - RTL → Synthese → Place&Route

## Regression: fast\_reg

### Regression Group: default

| Simulation Run<br>Unit/Simulator/Testcase [Variant]    | State<br>Results, Pass/<br>Fail | Simulator<br>Defines/<br>Params |
|--|---------------------------------|---------------------------------|
| <b>blizzard_top tc_avfs_bypass_mode</b>                | finished simulation             |                                 |
| blizzard_top/ncsim/tc_avfs_bypass_mode [default]       | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_avfs_bypass_mode_pmbus</b>          | finished simulation             |                                 |
| blizzard_top/ncsim/tc_avfs_bypass_mode_pmbus [default] | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_avfs_closed_loop</b>                | finished simulation             |                                 |
| blizzard_top/ncsim/tc_avfs_closed_loop [default]       | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_clkmeas_primary</b>                 | finished simulation             |                                 |
| blizzard_top/ncsim/tc_clkmeas [default]                | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_dhry_programmable</b>               | finished simulation             |                                 |
| blizzard_top/ncsim/tc_dhry_programmable [default]      | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_finish_led</b>                      | finished simulation             |                                 |
| blizzard_top/ncsim/tc_finish_led [default]             | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_hpm_bypass_mode</b>                 | finished simulation             |                                 |
| blizzard_top/ncsim/tc_hpm_bypass_mode [default]        | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_libtest</b>                         | finished simulation             |                                 |
| blizzard_top/ncsim/tc_libtest [default]                | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_pll_change</b>                      | finished simulation             |                                 |
| blizzard_top/ncsim/tc_pll_change [default]             | C:2 W:1 N:1                     | ?                               |
| <b>blizzard_top tc_power_load_ctrl</b>                 | finished simulation             |                                 |
| blizzard_top/ncsim/tc_power_load_ctrl [default]        | C:11 W:10 N:1                   | ?                               |
| <b>blizzard_top tc_sanity</b>                          | finished simulation             |                                 |
| blizzard_top/ncsim/tc_sanity [default]                 | C:2 W:1 N:1                     | ?                               |

# Zusammenfassung und Ausblick

- Zusammenfassung:
  - Übersicht Design Flow
  - Konzepte zur Hardwarebeschreibung
  - Grundlagen der HDL Verilog
  - Strategien zur Verifikation durch Simulation
- Ausblick auf weitere Themen:
  - Synthesegerechte Verilog Beschreibung
  - RTL-to-GDS Flow (Synthese, Place&Route, Timing Analyse)
  - Erweiterte Verifikationsmethoden
- → **Vorlesung und Praktikum „VLSI Prozessorentwurf“**