

Digitaltechnik– Kapitel 4



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prof. Sarah Harris, Ph.D.
Fachgebiet Eingebettete Systeme und ihre Anwendungen (ESA)
Fachbereich Informatik

WS 15/16



Kapitel 4: Themen

- Einleitung
- Kombinatorische Logik
- Strukturelle Beschreibung
- Sequentielle Logik
- Mehr kombinatorische Logik
- Endliche Zustandsautomaten
- Parametrisierte Modelle
- Testumgebungen

Vivado

Kapitel 4: Themen

- Einleitung
- Kombinatorische Logik
- Strukturelle Beschreibung
- Sequentielle Logik
- Mehr kombinatorische Logik
- Endliche Zustandsautomaten
- Parametrisierte Modelle
- Testumgebungen

**Es wird sehr schwer sein
die Klausur zu bestehen
wenn Sie SystemVerilog
nicht gut können.**

- **Hardware-Beschreibungssprachen**
 - *Hardware Description Languages (HDL)*
- Erlauben **textuelle** Beschreibung von Schaltungen
 - Auf **verschiedenen** Abstraktionsebenen
 - **Struktur** (z.B. Verbindungen zwischen Gattern)
 - Verhalten (z.B. Boole'sche Gleichungen)
- **Entwurfswerkzeuge** erzeugen Schaltungsstruktur daraus automatisch
 - Computerprogramme
 - Computer-Aided Design (CAD) oder Electronic Design Automation (EDA)
 - Schaltungssynthese
 - Grob vergleichbar mit Übersetzung (Compilieren) von konventionellen Programmiersprachen

Vivado

Einleitung

- Fast alle **kommerziellen** Hardware-Entwürfe mit HDLs realisiert
- **Zwei** HDLs haben sich durchgesetzt
 - SystemVerilog ←
 - VHDL
- Sie werden **SystemVerilog** lernen müssen

SystemVerilog

- 1984 von der Firma Gateway Design Automation entwickelt
- Seit 1995 ein IEEE Standard (1364)
 - Überarbeitet 2001 und 2005
 - Neuer Dialekt SystemVerilog (Obermenge von Verilog-2005)
- Weit verbreitet in zivilen US-Firmen ←
- In Darmstadt im Fachbereich Informatik
 - Rechnerorganisation (ESA, Prof. Koch – Prof Harris, Sommer '15)
 - Eingebettete Systeme und ihre Anwendungen (ESA, Prof. Koch)
- In Darmstadt im Fachbereich Elektrotechnik
 - Rechnersysteme (RS, Prof. Hochberger)

VHDL

- ***Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language***
- Entwickelt 1981 durch das US **Verteidigungsministerium**
 - Inspiriert durch konventionelle Programmiersprache **Ada**
- Standardisiert in 1987 durch **IEEE (1076)**
 - Überarbeitet in 1993, 2000, 2002, 2006, 2008
- Weit verbreitet in
 - US-Rüstungsfirmen
 - Vielen europäischen Firmen
- In Darmstadt im Fachbereich Elektrotechnik
 - **Integrierte elektronische Systeme** (IES, Prof. Hofmann)

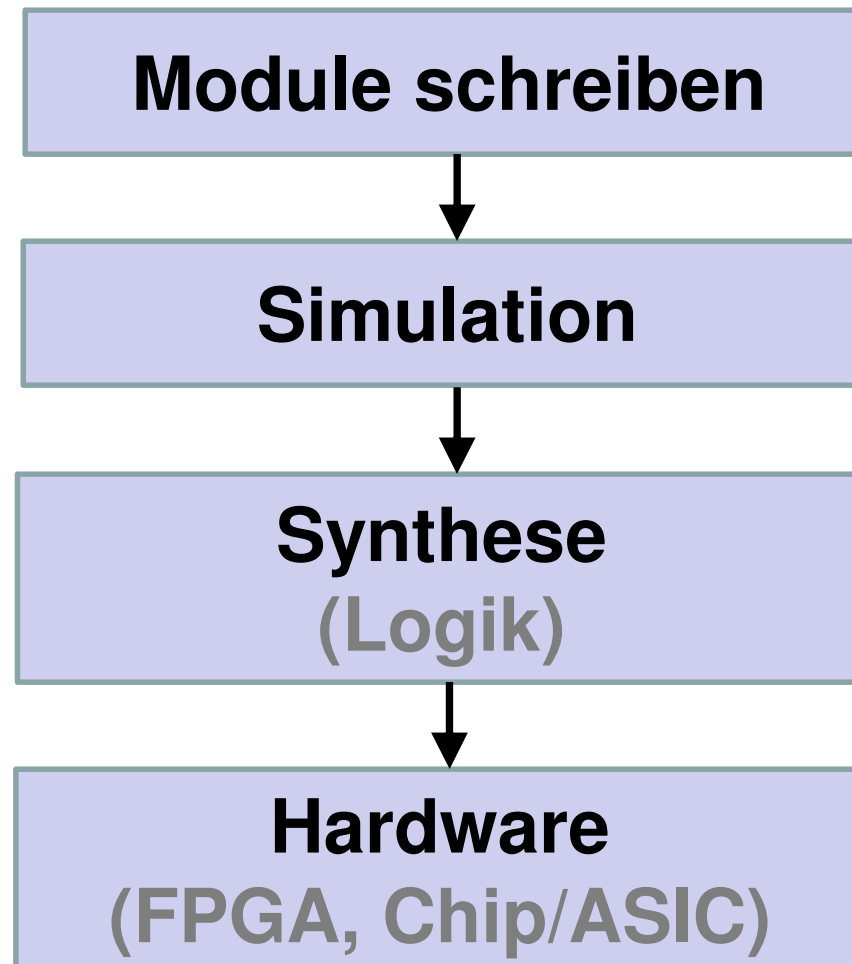
In dieser Iteration der Vorlesung

- In den **Vorlesungen** SystemVerilog
 - Häufig kompakter zu schreiben
 - Eher auf Einzelfolien darstellbar
- Hier gezeigte Grundkonzepte sind in beiden Sprachen identisch
- Nur andere **Syntax**
 - VHDL-Beschreibung ist aber in der Regel länger
- Im Buch werden beide Sprachen nebeneinander gezeigt
 - Kapitel 4
 - Moderne Entwurfswerkzeuge können in der Regel **beide** Sprachen

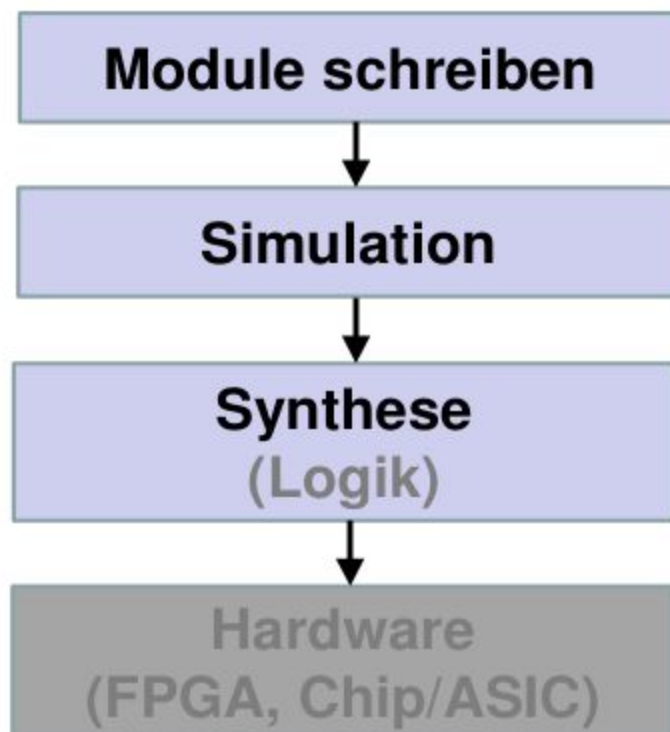
Von einer HDL zu Logikgattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Von einer HDL zu Logikgattern



Von einer HDL zu Logikgattern

▪ Simulation

- Eingangswerte werden in HDL-Beschreibung eingegeben
 - Beschriebene Schaltung wird **stimuliert**
- Berechnete Ausgangswerte werden auf Korrektheit **geprüft**
- Fehlersuche viel einfacher und billiger als in realer Hardware

▪ Synthese

- Übersetzt HDL-Beschreibungen in **Netzlisten**
 - **Logikgatter** (Schaltungselemente)
 - **Verbindungen** (Verbindungsknoten)

Von einer HDL zu Logikgattern

▪ Simulation

- Eingangswerte werden in HDL-Beschreibung eingegeben
 - Beschriebene Schaltung wird **stimuliert**
- Berechnete Ausgangswerte werden auf Korrektheit **geprüft**
- Fehlersuche viel einfacher und billiger als in realer Hardware

▪ Synthese

- Übersetzt HDL-Beschreibungen in **Netzlisten**
 - **Logikgatter** (Schaltungselemente)
 - **Verbindungen** (Verbindungsknoten)

WICHTIG:

Beim Verfassen von HDL-Beschreibungen ist es essentiell wichtig, immer die vom Programm beschriebene **Hardware** im Auge zu behalten!



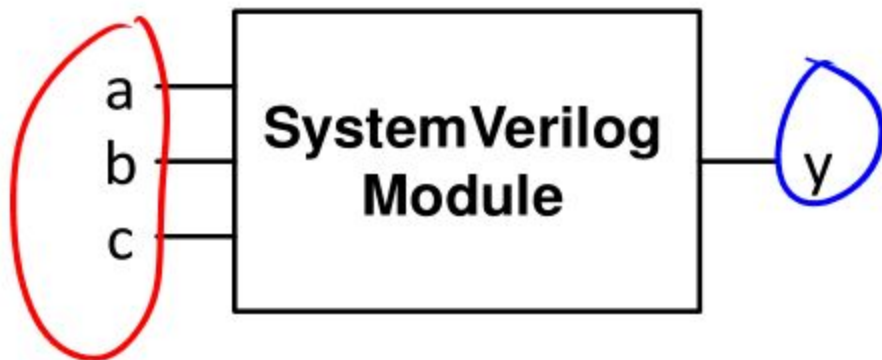
Zwei Arten von Beschreibungen in Modulen:

- **Verhalten:** Was tut die Schaltung?
- **Struktur:** Wie ist die Schaltung aus Untermodulen aufgebaut?

Beispiel für Verhaltensbeschreibung

SystemVerilog:

```
module example(input logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```



Beispiel für Verhaltensbeschreibung

SystemVerilog:

```
module example(input  logic a, b, c,  
                output logic y);
```

→ assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;
endmodule

$$\bar{a} \cdot \bar{b} \cdot \bar{c} + a \cdot \bar{b} \cdot \bar{c} + a \bar{b} c$$

- module/endmodule: Anfang/Ende der Module
- example: Namen der Module
- Operatoren:

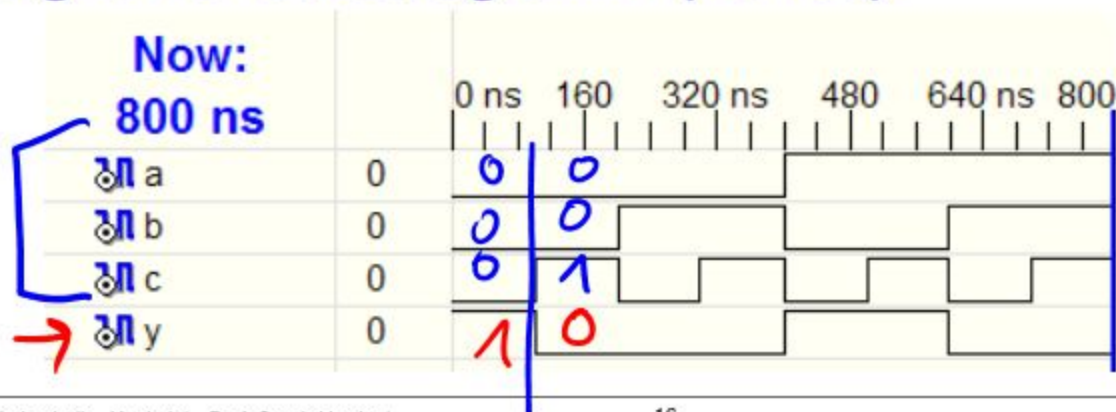
~: NOT (nicht)
&: AND (und)
|: OR (oder)

Simulation von Verhaltensbeschreibungen

SystemVerilog:

```
module example(input  logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

Signalverlaufdiagramm (*waves*):

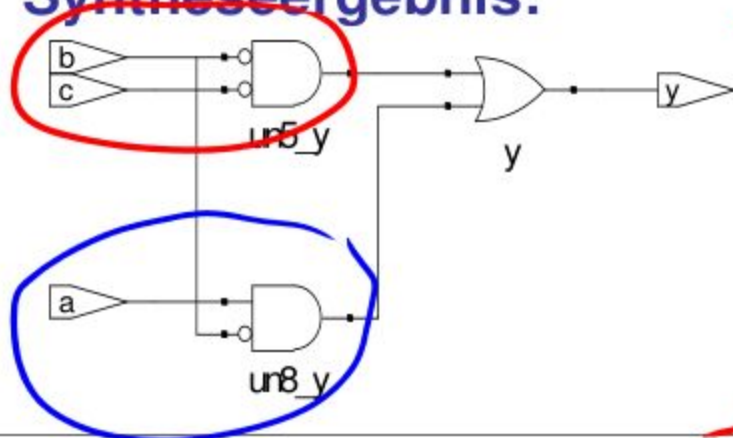


Synthese von Verhaltensbeschreibungen

SystemVerilog:

```
module example(input  logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

Syntheseresult:



$$y = \bar{b}\bar{c} + a\bar{b}$$

SystemVerilog Demo / Tutorial



TECHNISCHE
UNIVERSITÄT
DARMSTADT

**Aufzeichnung auf der ESA Webseite ab Heute
nachmittag.**

Organisatorische

Weihnachtspause

- **Nächste Vorlesung: 13. Januar 2016**

Noch später:

- Letzte Vorlesung: 10. Februar 2016
- Klausur: 1. März 2016



Bonbon Automat

- **3 Stufen:**
 - Bonbon holen
 - Bonbon jonglieren
 - Bonbon werfen

Von einer HDL zu Logikgattern



TECHNISCHE
UNIVERSITÄT
DARMSTADT

■ Simulation

- Eingangswerte werden in HDL-Beschreibung eingegeben
 - Beschriebene Schaltung wird **stimuliert**
- Berechnete Ausgangswerte werden auf Korrektheit **geprüft**
- Fehlersuche viel einfacher und billiger als in realer Hardware

■ Synthese

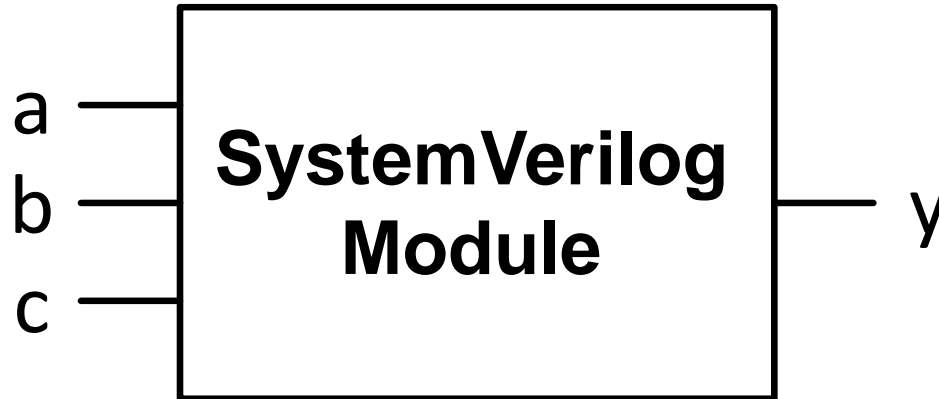
- Übersetzt HDL-Beschreibungen in **Netzlisten**
 - **Logikgatter** (Schaltungselemente)
 - **Verbindungen** (Verbindungsknoten)

WICHTIG:

Beim Verfassen von HDL-Beschreibungen ist es essentiell wichtig, immer die vom Programm beschriebene **Hardware** im Auge zu behalten!



SystemVerilog Module



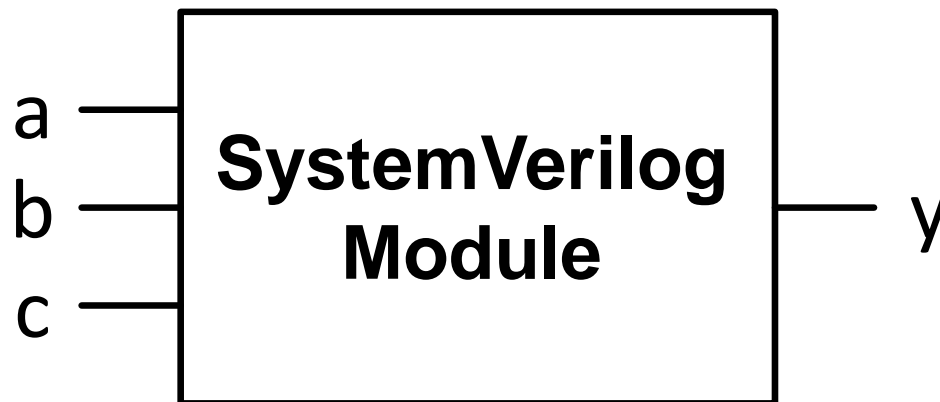
Zwei Arten von Beschreibungen in Modulen:

- **Verhalten:** Was tut die Schaltung?
- **Struktur:** Wie ist die Schaltung aus Untermodulen aufgebaut?

Beispiel für Verhaltensbeschreibung

SystemVerilog:

```
module example(input  logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;  
endmodule
```



Beispiel für Verhaltensbeschreibung

SystemVerilog:

```
module example(input  logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

$\overline{a}\overline{b}\overline{c} + a\overline{b}\overline{c} + a\overline{b}c$

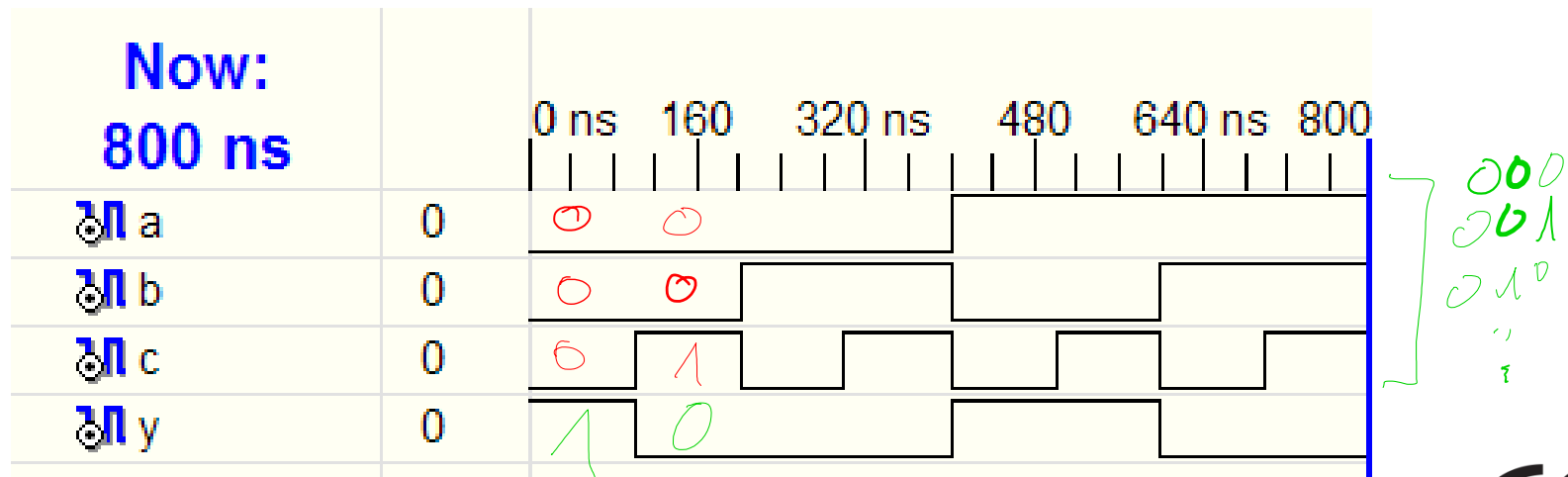
- module/endmodule: Anfang/Ende der Module
- example: Namen der Module
- Operatoren:
 - ~: NOT (nicht)
 - &: AND (und)
 - |: OR (oder)

Simulation von Verhaltensbeschreibungen

SystemVerilog:

```
module example(input logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

Signalverlaufdiagramm (waves):

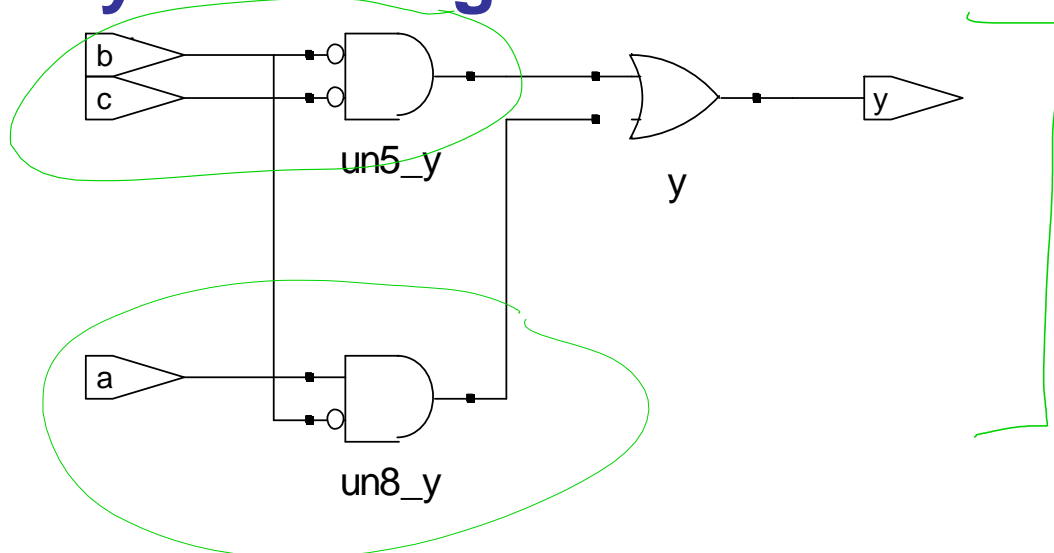


Synthese von Verhaltensbeschreibungen

SystemVerilog:

```
module example(input logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

Syntheseergebnis:



$$\overline{b}c + a\overline{b}$$

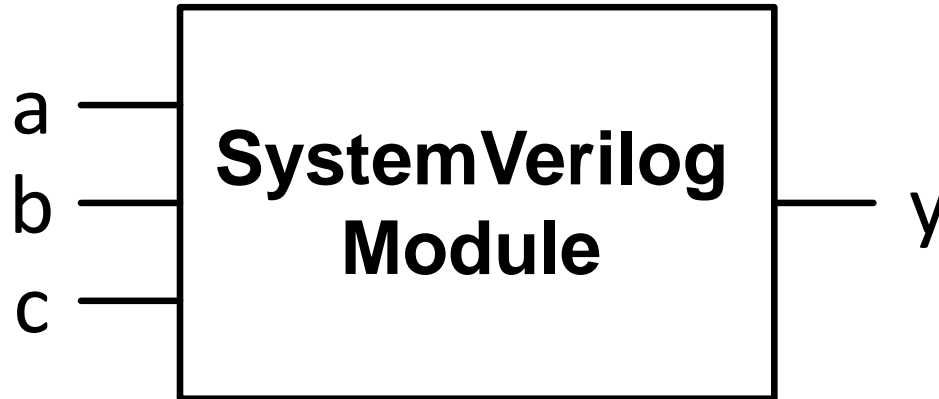
SystemVerilog Syntax

- Unterscheidet Groß- und Kleinschreibung
 - **Beispiel:** `reset` und `Reset` sind **nicht** das gleiche Signal
- Namen dürfen **nicht** mit Ziffern anfangen
 - **Beispiel:** `2mux` ist ein ungültiger Name
- Anzahl von Leerzeichen, Leerzeilen und Tabulatoren **irrelevant**
- Kommentare:
 - `//` bis zum **Ende** der Zeile
 - `/*` über **mehrere**
Zeilen `*/`

SystemVerilog Syntax

- **Unterscheidet** Groß- und Kleinschreibung
 - **Beispiel:** `reset` und `Reset` sind **nicht** das gleiche Signal
- Namen dürfen **nicht** mit Ziffern anfangen
 - **Beispiel:** `2mux` ist ein ungültiger Name
- Anzahl von Leerzeichen, Leerzeilen und Tabulatoren **irrelevant**
- Kommentare: **Sehr ähnlich zu C und Java!**
 - `//` bis zum **Ende** der Zeile
 - `/*` über **mehrere**
Zeilen `*/`

SystemVerilog Module



Zwei Arten von Beschreibungen in Modulen:

- **Verhalten:** Was tut die Schaltung?
- **Struktur:** Wie ist die Schaltung aus Untermodulen aufgebaut?

Strukturelle Beschreibung: Modulhierarchie



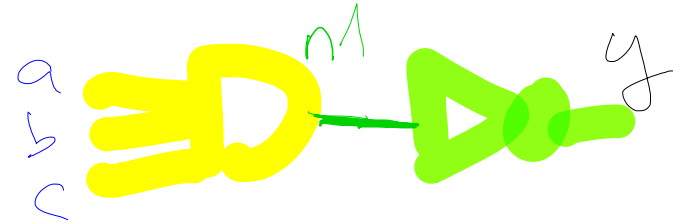
TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module and3(input  logic a, b, c,  
            output logic y);  
    assign y = a & b & c;  
endmodule
```

```
module inv(input  logic a,  
          output logic y);  
    assign y = ~a;  
endmodule
```

```
module nand3 (input  logic a, b, c  
             output logic y);  
    logic n1; // internes Signal (Verbindungsknoten)
```

```
    and3 andgate(a, b, c, n1); // Instanz von and3 namens andgate  
    inv inverter(n1, y); // Instanz von inv namens inverter  
endmodule
```



Strukturelle

Strukturelle Beschreibung: Modulhierarchie



```
module and3(input  logic a, b, c,  
            output logic y);  
    assign y = a & b & c;  
endmodule
```

```
module inv(input  logic a,  
           output logic y);  
    assign y = ~a;  
endmodule
```

```
module nand3 (input  logic d, e, f  
              output logic w);  
    logic n1; // internes Signal (Verbindungsknoten)  
  
    and3 andgate(d, e, f, n1); // Instanz von and3 namens andgate  
    inv  inverter(n1, w); // Instanz von inv namens inverter  
endmodule
```

Strukturelle Beschreibung: Modulhierarchie

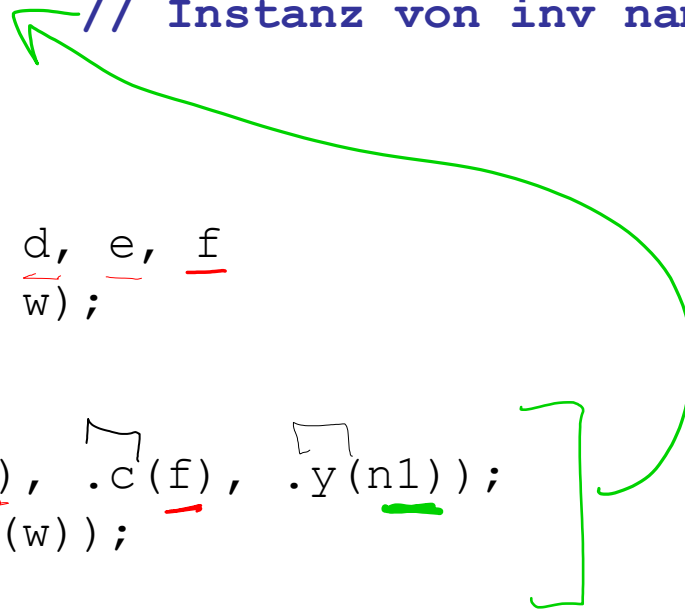


```
module nand3 (input logic d, e, f
              output logic w);
    logic n1; // internes Signal (Verbindungsknoten)

    and3 andgate(d, e, f, n1); // Instanz von and3 namens andgate
    inv inverter(n1, w); // Instanz von inv namens inverter
endmodule
```

```
module nand3 (input logic d, e, f
              output logic w);
    logic n1;

    and3 andgate(.a(d), .b(e), .c(f), .y(n1));
    inv inverter(.a(n1), .y(w));
endmodule
```



Bitweise Verknüpfungsoperatoren

```
module gates (input logic [3:0] a, b,  
             output logic [3:0] y1, y2, y3, y4, y5);
```

```
/* Fünf unterschiedliche Logikgatter
```

```
   mit zwei Eingängen, jeweils 4b Busse */
```

```
assign y1 = a & b; // AND
```

```
assign y2 = a | b; // OR
```

```
assign y3 = a ^ b; // XOR
```

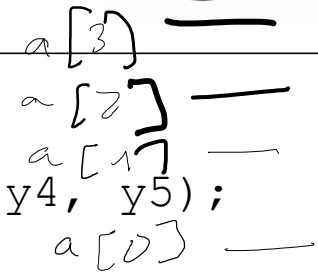
```
assign y4 = ~(a & b); // NAND
```

```
assign y5 = ~(a | b); // NOR
```

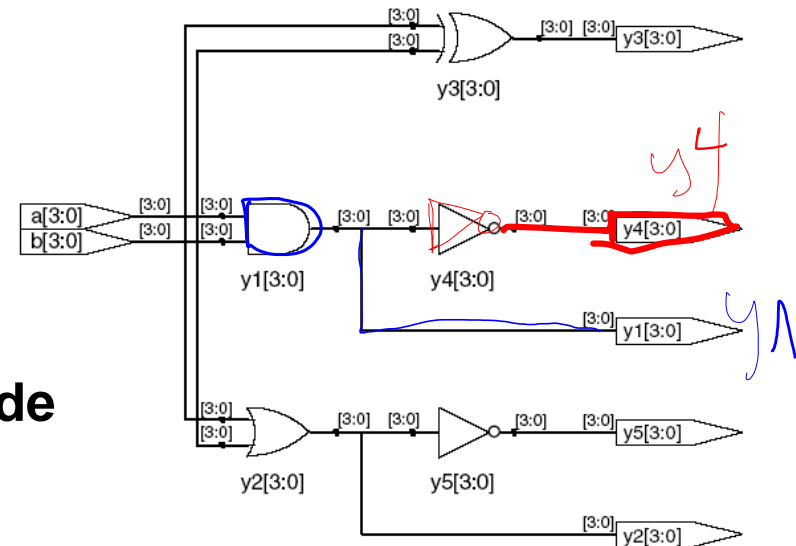
```
endmodule
```

```
//      Kommentar bis zum Zeilenende
```

```
/*...*/  Mehrzeiliger Kommentar
```



Syntheseeergebnis:



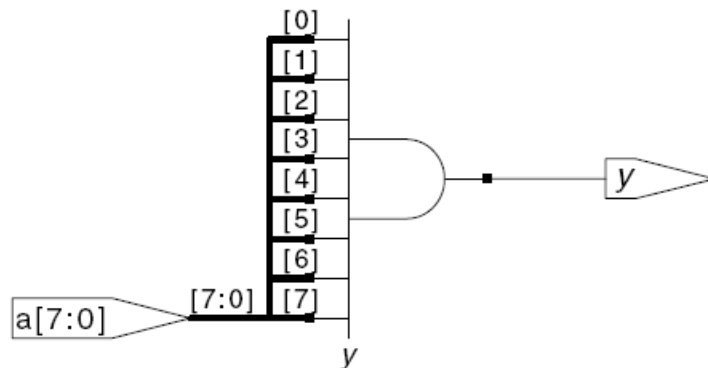
Reduktionsoperatoren



```
module and8 (input logic [7:0] a,  
             output logic      y);  
    assign y = &a; ←  
    // &a ist Abkürzung für  
    // assign y = a[7] & a[6] & a[5] & a[4] &  
    //           a[3] & a[2] & a[1] & a[0];  
endmodule
```

$y = a;$

Syntheseergebnis:



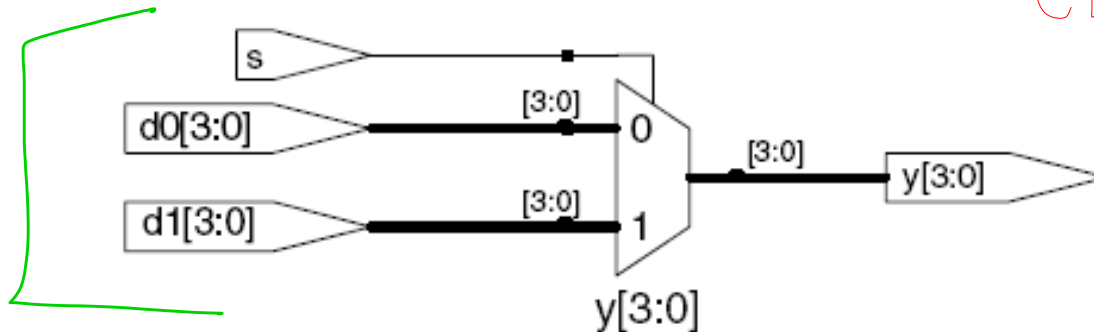
Bedingte Zuweisung

```
module mux2 (input logic[3:0] d0, d1,  
             input logic s,  
             output logic [3:0] y);  
    assign y = s ? d1 : d0;  
endmodule
```

Pseudocode

*if (s=1)
 y = d1
else
 y = d0*

Syntheseergebnis:



? : ist ein ternärer Operator, da er **drei** Operanden miteinander verknüpft: s, d1, und d0.

Interne Verbindungsknoten oder Signale



```
module fulladder(input logic a, b, cin,  
                output logic s, cout);  
  
    logic p, g;
```

p = propagate

g = generate

```
    assign p = a ^ b;  
    assign g = a & b;
```

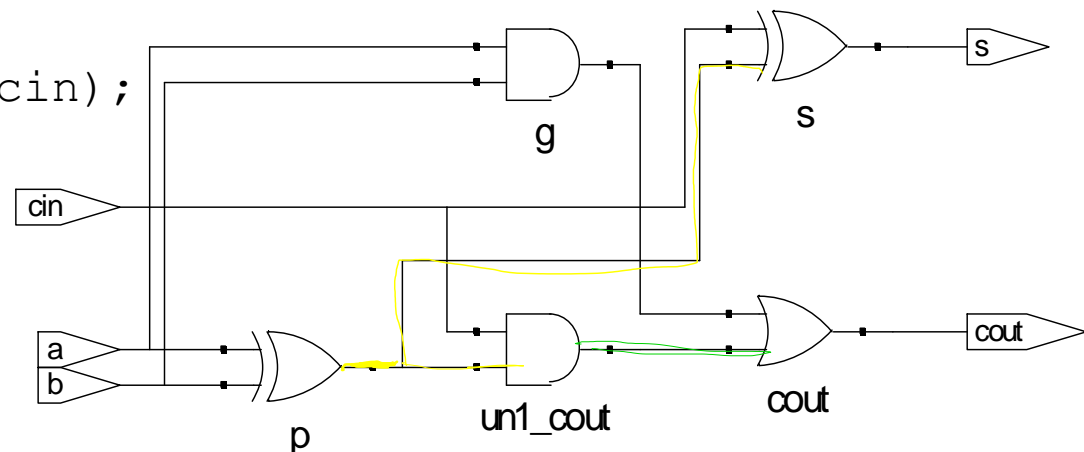
assign p = a | b;

```
    assign s = p ^ cin;  
    assign cout = g | (p & cin);  
endmodule
```

Summe

Übertrag

Syntheseeergebnis:



Bindung von Operatoren (Präzedenz)

Bestimmt Ausführungsreihenfolge

Höchste

~	NOT
<u>*</u> , <u>/</u> , <u>%</u>	Multiplikation, Division, Modulo
+ , -	Addition, Subtraktion
<< , >>	Schieben (logisch)
<<< , >>>	Schieben (arithmetisch)
< , <= , > , >=	Vergleiche
== , !=	gleich, ungleich
& , ~&	<u>AND, NAND</u>
^ , ~^	<u>XOR, XNOR</u>
, ~	<u>OR, NOR</u>
?:	Ternärer Operator

Niedrigste

Zahlen

X 3'10

Syntax: *N'Bwert*

N = Breite in Bits, *B* = Basis

N'B ist optional, sollte der Konsistenz halber aber immer geschrieben werden
wenn weggelassen: Dezimalsystem

Zahl	Bitbreite	Basis	entspricht Dezimal	Darstellung im Speicher
3'b101	3	binär	5	101
'b11	Nicht vorgegeben	binär	3	00...0011
8'b11	8	binär	3	00000011
8'b1010_1011	8	binär	171	10101011
3'd6	3	dezimal	6	110
6'o42	6	oktal	34	100010
8'hAB	8	hexadezimal	171	10101011
42	Nicht vorgegeben	dezimal	42	00...0101010

Operationen auf Bit-Ebene: Beispiel 1

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

// wenn y ein 12-bit Signal ist, hat die
// Anweisung diesen Effekt:

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

Unterstriche (_) in numerischen Konstanten dienen nur der besseren Lesbarkeit, sie werden von SystemVerilog **ignoriert**



Operationen auf Bit-Ebene: Beispiel 2

SystemVerilog:

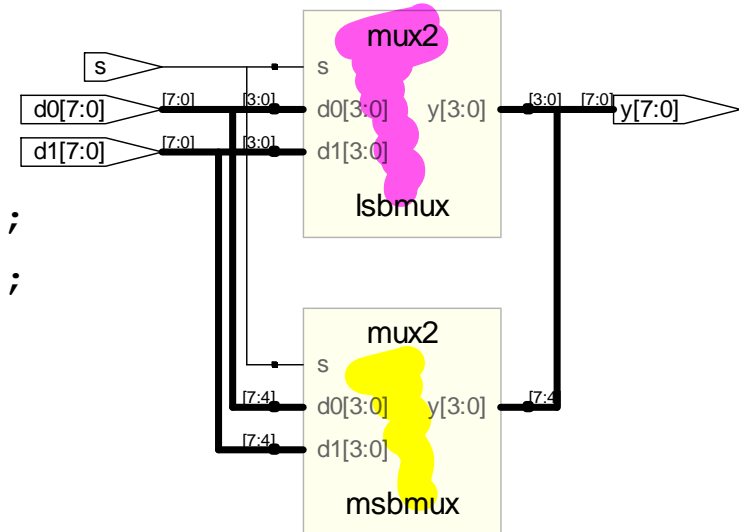
```
module mux2_8(input logic [7:0] d0, d1,  
             input logic s,  
             output logic [7:0] y);
```

```
    mux2 lsbmux (d0[3:0], d1[3:0], s, y[3:0]);
```

```
    mux2 msbmux (d0[7:4], d1[7:4], s, y[7:4]);
```

```
endmodule
```

Syntheseresult:



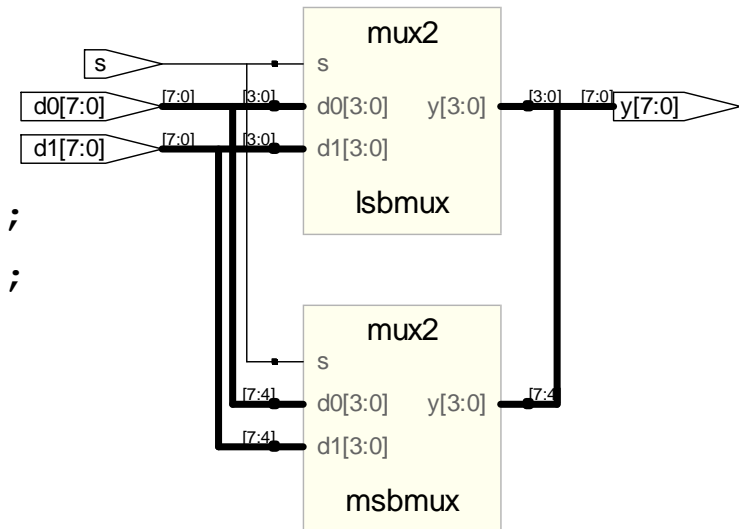
Benutzt mux2
module von
Folie 30.

Operationen auf Bit-Ebene: Beispiel 2

SystemVerilog:

```
module mux2_8(input logic [7:0] d0, d1,  
             input logic      s,  
             output logic [7:0] y);  
  
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);  
endmodule  
  
...  
mux2 lsbmux(.d0(d0[3:0]), .d1(d1[3:0]),  
           .s(s), .y(y[3:0]));  
mux2 msbmux(.d0(d0[7:4]), .d1(d1[7:4]),  
           .s(s), .y(y[7:4]));  
  
...
```

Syntheseresult:



**Benutzt mux2
module von
Folie 30.**

Operationen auf Bit-Ebene: Beispiel 2



```
module mux2_8(input logic [7:0] d0, d1,  
             input logic      s,  
             output logic [7:0] y);  
  mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
  mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);  
endmodule
```

```
module mux2_8(input logic [7:0] d0, d1,  
             input logic      s,  
             output logic [7:0] y);  
  mux2 lsbmux(.d0(d0[3:0]), .d1(d1[3:0]),  
             .s(s), .y(y[3:0]));  
  mux2 msbmux(.d0(d0[7:4]), .d1(d1[7:4]),  
             .s(s), .y(y[7:4]));  
endmodule
```

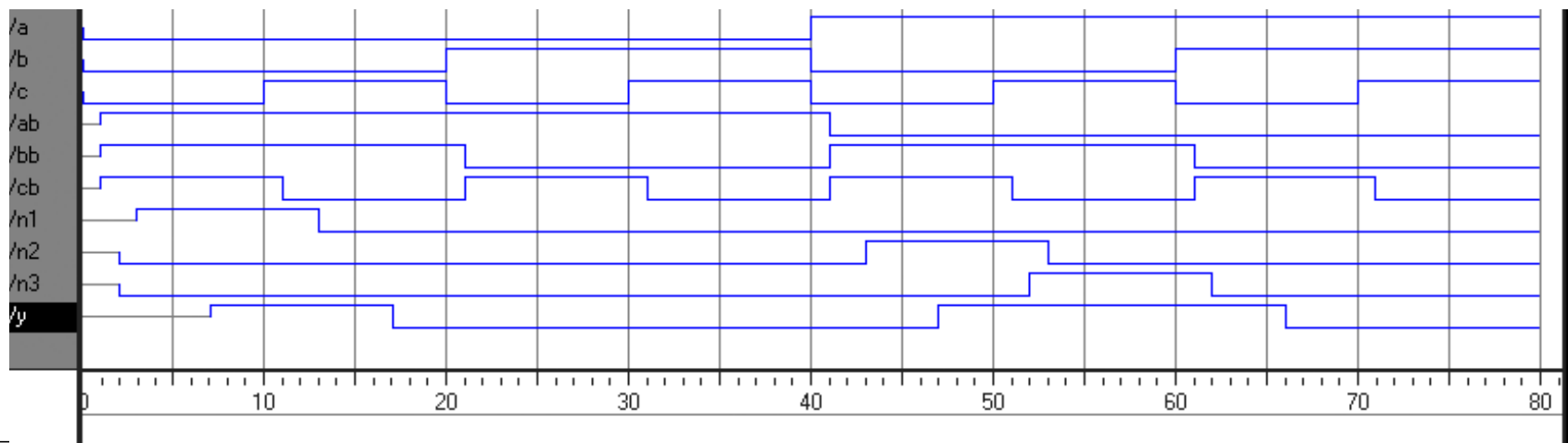
```
module mux2(input logic[3:0] d0, d1,  
           input logic      s,  
           output logic [3:0] y);  
  assign y = s ? d1 : d0;  
endmodule
```


Verzögerungen: # Zeiteinheiten



```
module example(input logic a, b, c,  
               output logic y);  
    logic ab, bb, cb, n1, n2, n3;  
    assign #1 {ab, bb, cb} =  
            ~{a, b, c};  
    assign #2 n1 = ab & bb & cb;  
    assign #2 n2 = a & bb & cb;  
    assign #2 n3 = a & bb & c;  
    assign #4 y = n1 | n2 | n3;  
endmodule
```

Simulation

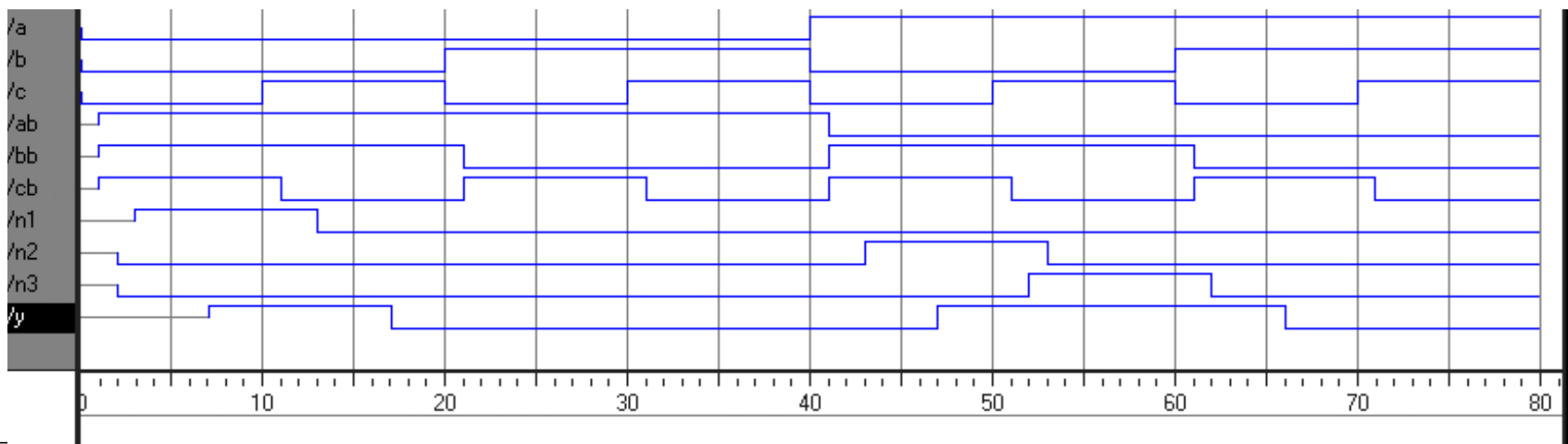


Verzögerungen: # Zeiteinheiten



```
module example(input logic a, b, c,  
               output logic y);  
    logic ab, bb, cb, n1, n2, n3;  
    assign #1 {ab, bb, cb} =  
            ~{a, b, c};  
    assign #2 n1 = ab & bb & cb;  
    assign #2 n2 = a & bb & cb;  
    assign #2 n3 = a & bb & c;  
    assign #4 y = n1 | n2 | n3;  
endmodule
```

Nur für die Simulation,
#n werden für die
Synthese ignoriert!



Verzögerungen

$$ab = \bar{a}$$

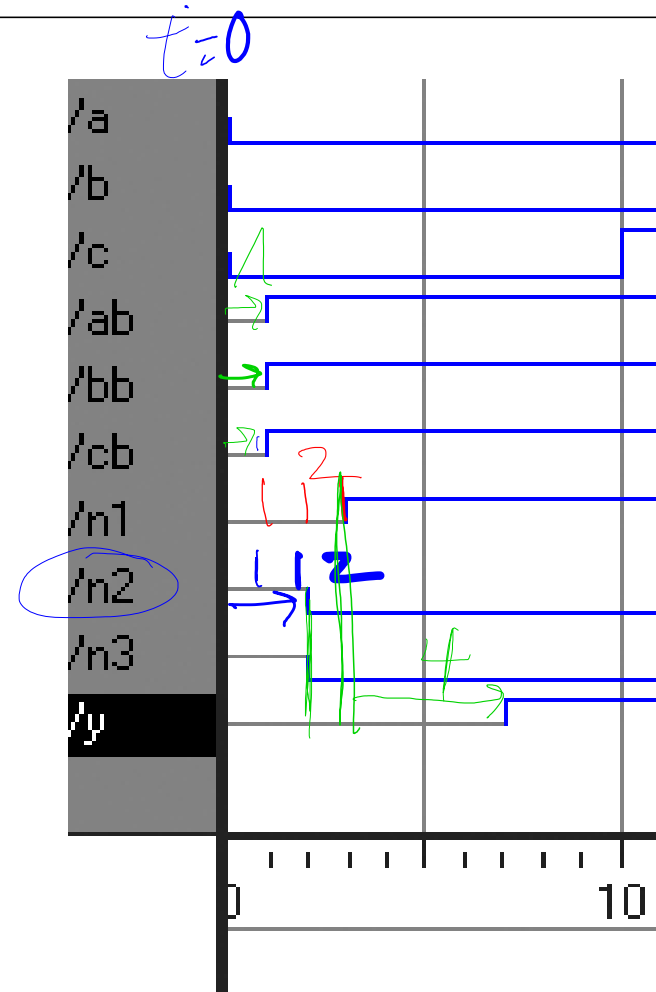
#Nnummer



```

module example(input logic a, b, c,
                output logic y);
    logic ab, bb, cb, n1, n2, n3;
    assign #1 {ab, bb, cb} =
                ~{a, b, c};

    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
    
```

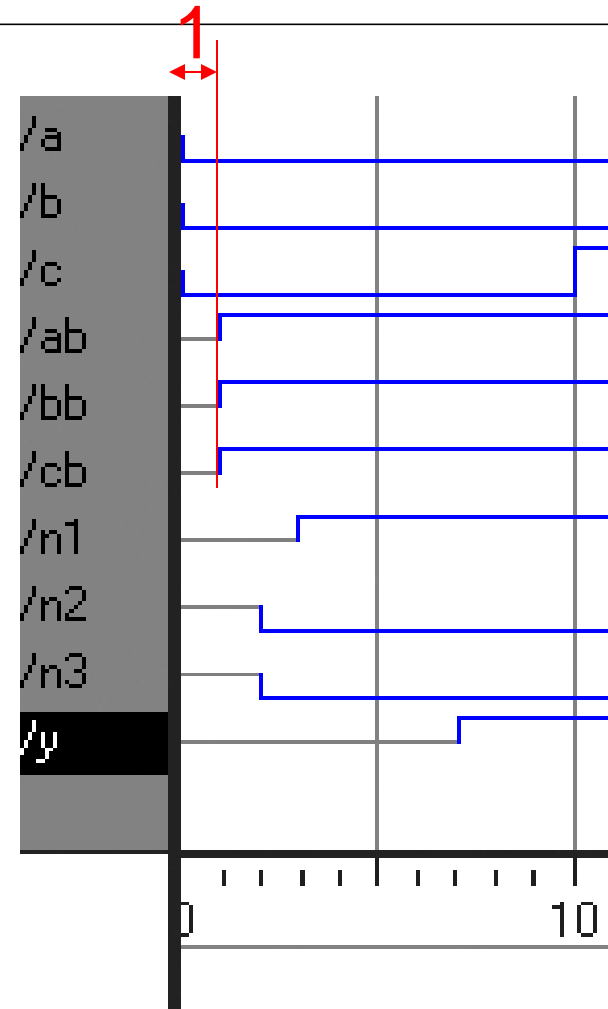


assign #1 ab = \bar{a} ;
 assign #1 bb = \bar{b} ;
 assign #1 cc = \bar{c} ;

Verzögerungen

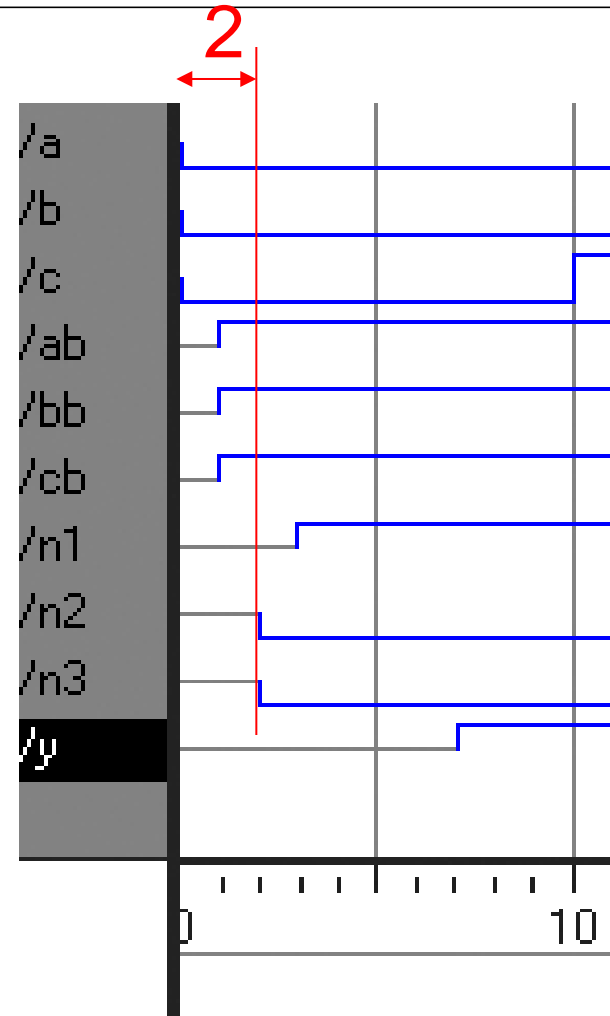


```
module example(input logic a, b, c,  
               output logic y);  
  logic ab, bb, cb, n1, n2, n3;  
  assign #1 {ab, bb, cb} =  
           ~{a, b, c};  
  assign #2 n1 = ab & bb & cb;  
  assign #2 n2 = a & bb & cb;  
  assign #2 n3 = a & bb & c;  
  assign #4 y = n1 | n2 | n3;  
endmodule
```



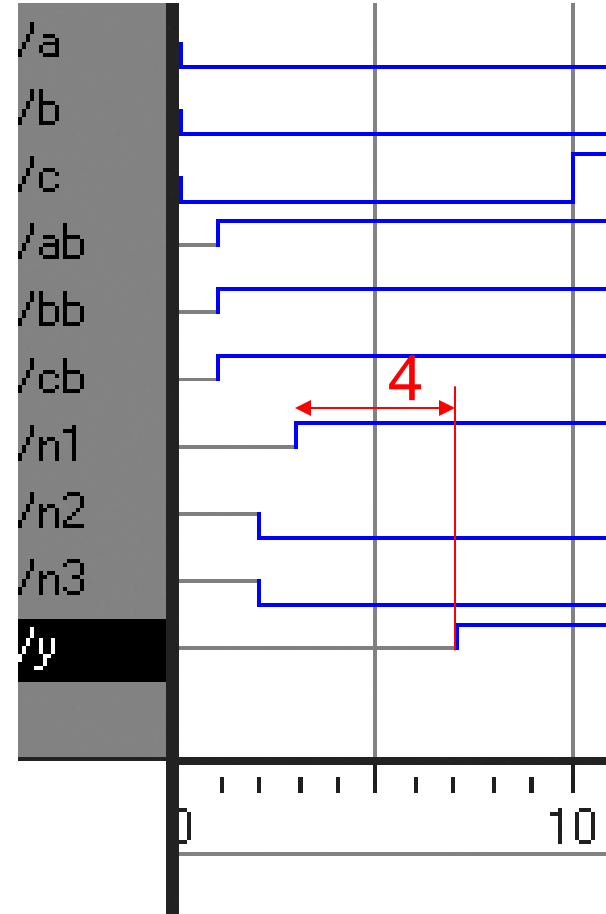
Verzögerungen

```
module example(input logic a, b, c,  
               output logic y);  
  logic ab, bb, cb, n1, n2, n3;  
  assign #1 {ab, bb, cb} =  
           ~{a, b, c};  
  assign #2 n1 = ab & bb & cb;  
  assign #2 n2 = a & bb & cb;  
  assign #2 n3 = a & bb & c;  
  assign #4 y = n1 | n2 | n3;  
endmodule
```



Verzögerungen

```
module example(input logic a, b, c,  
               output logic y);  
  logic ab, bb, cb, n1, n2, n3;  
  assign #1 {ab, bb, cb} =  
           ~{a, b, c};  
  assign #2 n1 = ab & bb & cb;  
  assign #2 n2 = a & bb & cb;  
  assign #2 n3 = a & bb & c;  
  assign #4 y = n1 | n2 | n3;  
endmodule
```



Sequentielle Schaltungen

- Beschreibung basiert auf Verwendung fester “Redewendungen”
 - Idiome
 - **Feststehende** Idiome für
 - Latches
 - Flip-Flops
 - Endliche Zustandsautomaten (FSM)
 - Vorsicht beim Abweichen von Idiomen
 - Wird möglicherweise noch richtig simuliert
 - Könnte aber fehlerhaft synthetisiert werden

Sequentielle Schaltungen

- Beschreibung basiert auf Verwendung fester “Redewendungen”
 - Idiome
- **Feststehende** Idiome für
 - Latches
 - Flip-Flops
 - Endliche Zustandsautomaten (FSM)
- Vorsicht beim **Abweichen** von Idiomen
 - Wird möglicherweise noch richtig simuliert
 - Könnte aber fehlerhaft synthetisiert werden

→ Halten Sie sich an die Konventionen!

always-Anweisung

Allgemeiner Aufbau:

```
always @ (sensitivity list)
    statement;
```

Handwritten annotations: A green checkmark is above the '@' symbol. 'a, b, c' is written above 'sensitivity list'. A green arrow points from the semicolon to the right.

Interpretation:

Wenn sich die in der sensitivity list aufgezählten Werte **ändern**, wird die Anweisung `statement` **ausgeführt**.

Werte: In der Regel Signale, manchmal noch erweitert

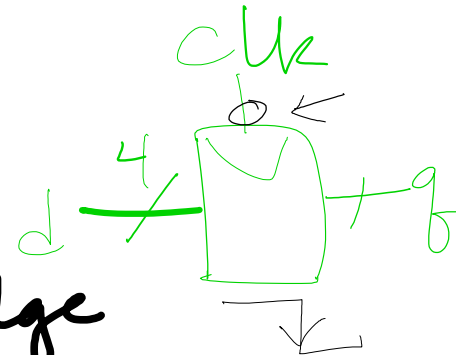
D Flip-Flop

```
module flop(input logic clk,  
            input logic [3:0] d,  
            output logic [3:0] q);
```

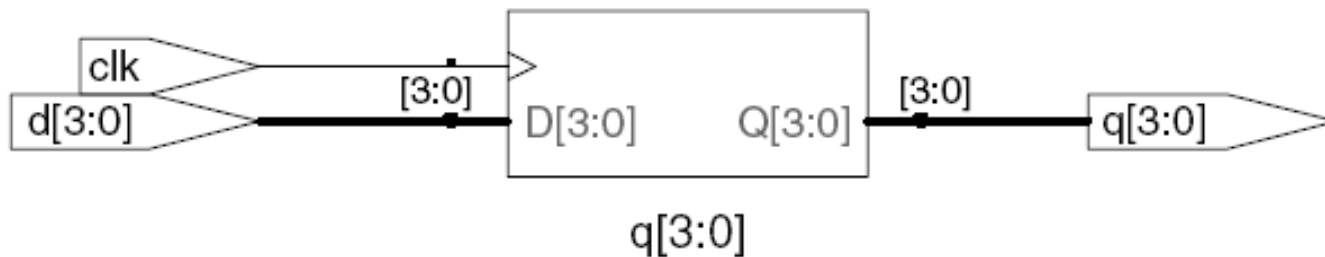
```
always_ff @(posedge clk)  
q <= d; // gelesen als "q übernimmt d"  
          // auf Englisch: "q gets d"
```

```
endmodule
```

negedge



Syntheseeergebnis:



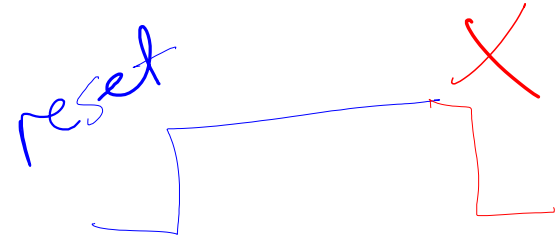
Rücksetzbares D Flip-Flop

```
module flopr (input logic clk,  
             input logic reset,  
             input logic [3:0] d,  
             output logic [3:0] q);
```

```
// synchroner Reset
```

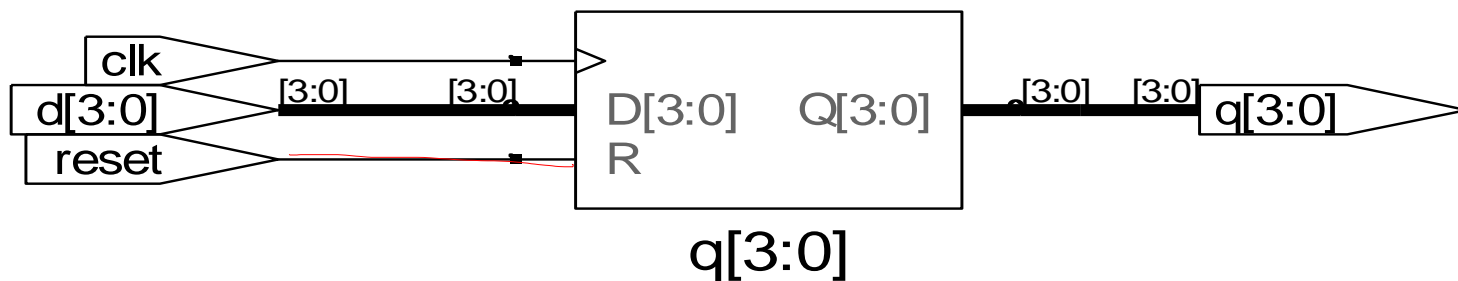
```
always_ff @(posedge clk)  
  if (reset) q <= 4'b0;  
  else      q <= d;
```

posedge reset



```
endmodule
```

Syntheseergebnis:



Rücksetzbares D Flip-Flop

```
module flopr(input logic clk,  
            input logic reset,  
            input logic [3:0] d,  
            output logic [3:0] q);
```

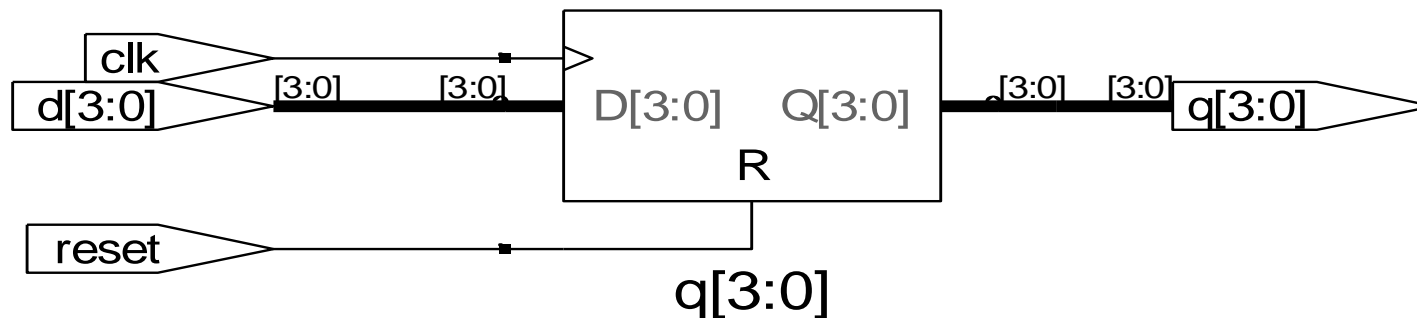
```
// asynchroner Reset
```

```
always_ff @(posedge clk, posedge reset)
```

```
→ if (reset) q <= 4'b0;  
   else      q <= d;
```

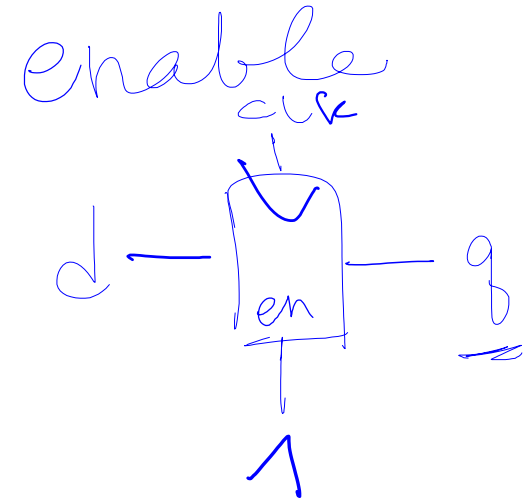
```
endmodule
```

Syntheseergebnis:



Rücksetzbares D Flip-Flop mit Taktfreigabe

```
module flopren(input logic clk,  
              input logic reset,  
              input logic en,  
              input logic [3:0] d,  
              output logic [3:0] q);
```

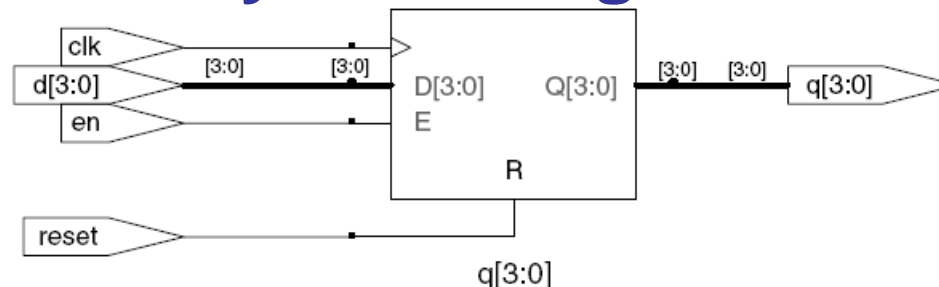


```
// asynchroner Reset mit Clock Enable
```

```
always @ (posedge clk, posedge reset) ←  
  if (reset) q <= 4'b0;  
  else if (en) q <= d;
```

```
endmodule
```

Syntheseresult:

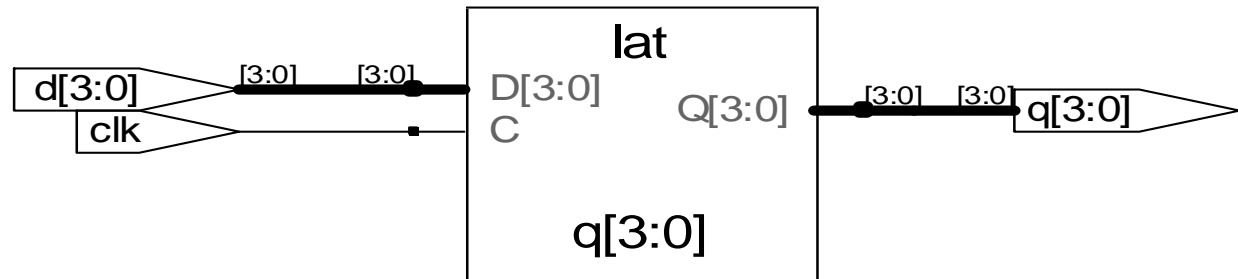


Latch

```
module latch (input logic clk,  
             input logic [3:0] d,  
             output logic [3:0] q);
```

```
    always_latch  
        if (clk) q <= d;  
endmodule
```

Syntheseergebnis:



Achtung: In dieser Veranstaltung werden Latches nur **selten** (wenn überhaupt) gebraucht werden.

Sollten sie dennoch in einem Syntheseergebnis auftauchen, ist das in der Regel auf **Fehler** in Ihrer HDL-Beschreibung zurückzuführen (z.B. Abweichen von Idiomen)!

Wiederholung

Allgemeiner Aufbau:

```
always @(sensitivity list)
    statement;
```

- **Flip-flop:** always_ff
- **Latch:** always_latch **(nicht benutzen)** ~~X~~

Weitere Anweisungen zur Verhaltensbeschreibung



- Dürfen nur **innerhalb** von always-Anweisungen benutzt werden
 - `if / else` ←
 - `case, casez` ←

Kombinatorische Logik als always-Block

```
module gates (input logic [3:0] a, b,  
              output logic [3:0] y1, y2, y3, y4, y5);
```

```
X (always_comb // wann immer sich irgendein gelesenes Signal ändert  
    begin      // bei mehr als einer Anweisung: begin/end  
        y1 = a & b;           // AND  
        y2 = a | b;          // OR  
        y3 = a ^ b;          // XOR  
        y4 = ~(a & b);       // NAND  
        y5 = ~(a | b);       // NOR  
    end  
endmodule
```

assign y1 = a & b;

In diesem Fall, wäre es besser assign-Anweisungen zu benutzen.

Hätte einfacher durch fünf `assign`-Anweisungen beschrieben werden können.

Kombinatorische Logik mit

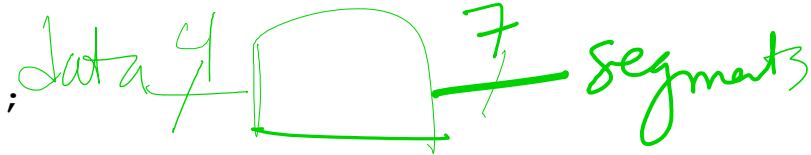
case



TECHNISCHE
UNIVERSITÄT
DARMSTADT

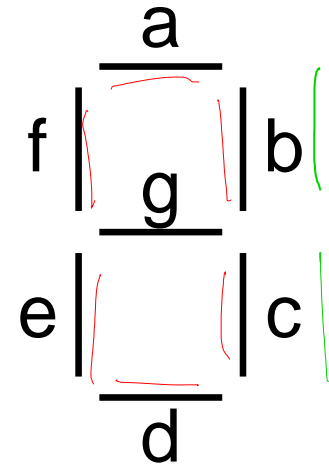
0-15

```
module sevenseg(input  logic [3:0] data,  
                output logic [6:0] segments);
```



...

```
endmodule
```



Kombinatorische Logik mit case



```
module sevenseg(input  logic [3:0] data,  
                 output logic [6:0] segments);
```

```
    always_comb // kombinatorische Logik ...
```

```
        case (data)
```

```
            0: segments = 7'b111_1110;
```

```
            1: segments = 7'b011_0000;
```

```
            2: segments = 7'b110_1101;
```

```
            3: segments = 7'b111_1001;
```

```
            4: segments = 7'b011_0011;
```

```
            5: segments = 7'b101_1011;
```

```
            6: segments = 7'b101_1111;
```

```
            7: segments = 7'b111_0000;
```

```
            8: segments = 7'b111_1111;
```

```
            9: segments = 7'b111_1011;
```

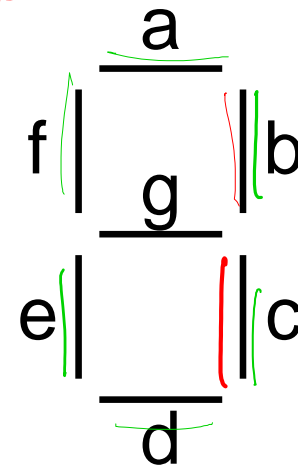
```
            default: segments = 7'b000_0000; // alle Fälle abgedeckt!
```

```
        endcase
```

```
    endmodule
```

*Ausgänge
abc_defg*

Wahrheitstabelle



Eingänge

Kombinatorische Logik mit case



```
module sevenseg(input  logic [3:0] data,  
                output logic [6:0] segments);  
  
    always_comb // kombinatorische Logik ...  
        case (data)  
            //                abc_defg  
            0: segments = 7'b111_1110;  
            1: segments = 7'b011_0000;  
            2: segments = 7'b110_1101;  
            3: segments = 7'b111_1001;  
            4: segments = 7'b011_0011;  
            5: segments = 7'b101_1011;  
            6: segments = 7'b101_1111;  
            7: segments = 7'b111_0000;  
            8: segments = 7'b111_1111;  
            9: segments = 7'b111_1011;  
            default: segments = 7'b000_0000; // alle Fälle abgedeckt!  
        endcase  
    endmodule
```

So einfach nicht als
assign formulierbar

Kombinatorische Logik mit

case



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- Um kombinatorische Logik zu beschreiben, muss ein `case`-Block **alle** Möglichkeiten abdecken
 - Entweder **explizit** angeben
 - Oder einen **default-Fall** angeben
 - Tritt in Kraft, wenn sonst keine andere Alternative passt
 - Im Beispiel verwendet

Kombinatorische Logik mit

casez

? = don't care



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Syntheseresultat:

```
module priority_casez (input logic [3:0] a,  
                      output logic [3:0] y);
```

```
  always_comb // kombinatorische Logik ...
```

```
    casez(a)
```

```
      4'b1???: y = 4'b1000; // ? = don't care
```

```
      4'b01??: y = 4'b0100;
```

```
      4'b001?: y = 4'b0010;
```

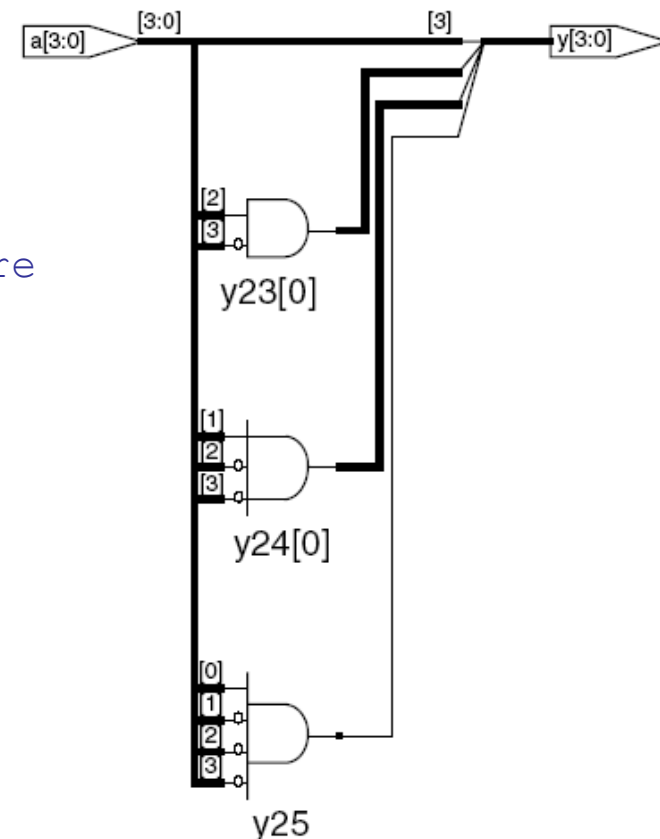
```
      4'b0001: y = 4'b0001;
```

```
      default: y = 4'b0000; // alle Fälle
```

```
                          // abgedeckt
```

```
    endcase
```

```
endmodule
```



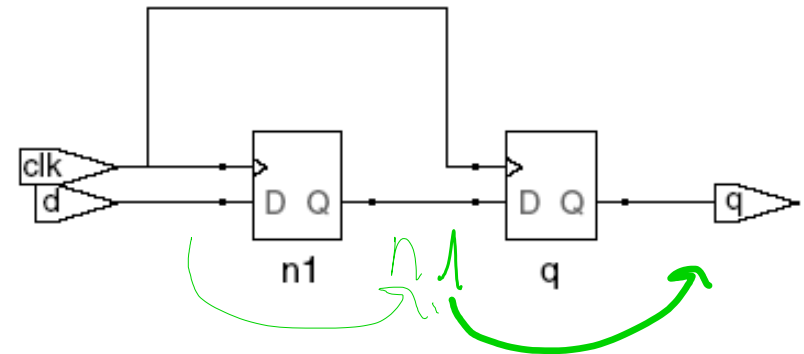
Nicht-blockende Zuweisung



- **<=** steht für eine **„nicht-blockende Zuweisung“**
- Wird **parallel** mit allen anderen nicht-blockenden Zuweisungen ausgeführt
 - 1. Schritt: Alle „rechten Seiten“ werden **berechnet**
 - 2. Schritt: Alle Berechnungsergebnisse werden an „linke Seiten“ **zugewiesen**

```
// Synchronisierer mit nicht-blockenden  
// Zuweisungen  
module syncgood(input logic clk,  
                 input logic d,  
                 output logic q);  
  
    logic n1;  
    always_ff(posedge clk)  
    begin  
        n1 <= d; // nicht-blockend  
        q <= n1; // nicht-blockend  
    end  
endmodule
```

Syntheseeergebnis:



Blockende Zuweisung

statt ←



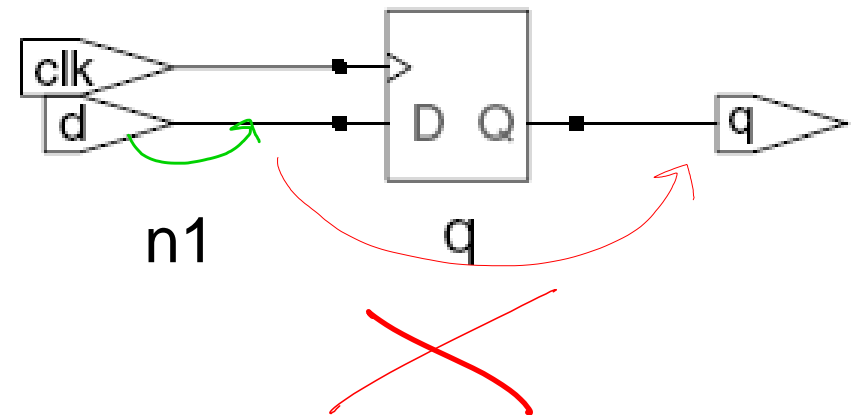
TECHNISCHE
UNIVERSITÄT
DARMSTADT

- = steht für eine **“blockende Zuweisung”**
- Wird **hintereinander** (seriell) in Reihenfolge im Programmtext ausgeführt
 - Solange eine blockende Zuweisung abläuft
 - ... werden andere Anweisungen **blockiert**
 - Jede Anweisung **für sich** berechnet „rechte Seite“ und weist an „linke Seite“ zu

```
// Fehlerhafter Synchronisierer
// mit blockenden Zuweisungen
module syncbad(input logic clk,
               /          input logic d,
               output logic q);

logic n1;
always_ff(posedge clk)
begin
  → n1 = d; // blockend
  → q = n1; // blockend
end
endmodule
```

Syntheseergebnis:



Regeln für Zuweisungen von Signalen

- Um synchrone sequentielle Logik zu beschreiben, benutzen Sie immer `always_ff @(posedge clk)` und nicht-blockende Zuweisungen (`<=`)

```
always_ff @(posedge clk)
    q <= d; // nicht-blockend
```

Regeln für Zuweisungen von Signalen

- Um **synchrone sequentielle** Logik zu beschreiben, benutzen Sie immer `always_ff @(posedge clk)` und nicht-blockende Zuweisungen (`<=`)

```
always_ff @(posedge clk)
    q <= d; // nicht-blockend
```

- Um **einfache kombinatorische** Logik zu beschreiben, benutzen Sie immer ständige Zuweisung (*continuous assignment*) (`assign ...`)

```
assign y = a & b;
```

Regeln für Zuweisungen von Signalen

- Um **synchrone sequentielle** Logik zu beschreiben, benutzen Sie immer `always_ff @(posedge clk)` und nicht-blockende Zuweisungen (`<=`)

```
always_ff @(posedge clk)
    q <= d; // nicht-blockend
```

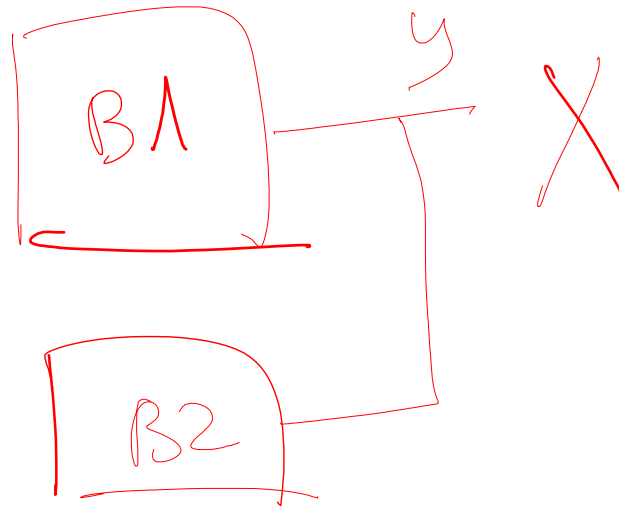
- Um **einfache kombinatorische** Logik zu beschreiben, benutzen Sie immer ständige Zuweisung (*continuous assignment*) (`assign ...`)

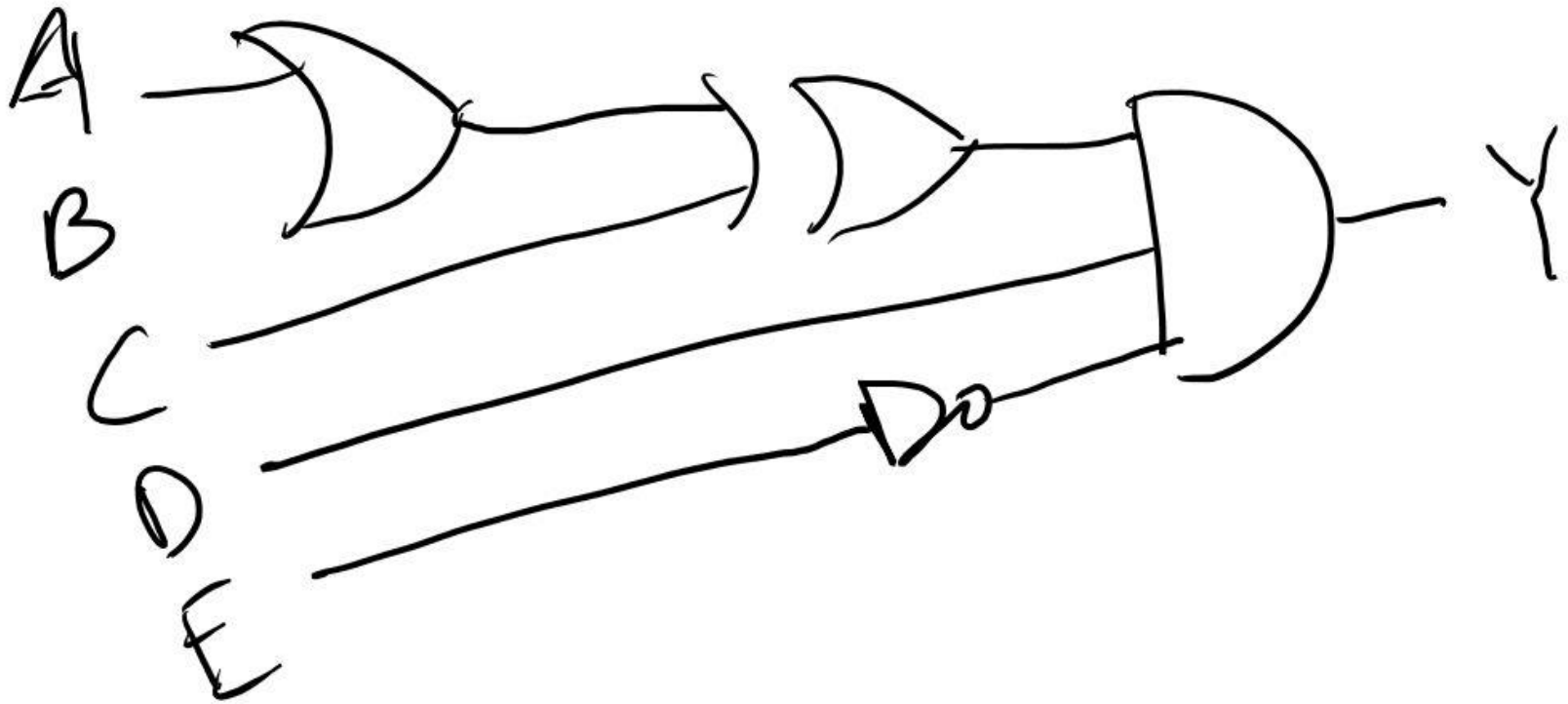
```
assign y = a & b;
```

- Um **komplexere kombinatorische** Logik zu beschreiben, benutzen Sie immer `always_comb` und blockende Zuweisungen (`=`)

Regeln für Zuweisungen von Signalen

- Weisen Sie **nicht** an ein Signal
 - ... in **mehreren** `always`-Blöcken zu
 - ... in einem `always`-Block **gemischt** mit `=` und `<=` zu
 - ... in einem `always`-Block **und** in einer ständigen Zuweisung (*continuous assignment*) (`assign ...`)





```
module gatter(input logic a,b,c,d,e,  
              output logic y);
```

```
    assign y = ~e & d & (c ^ (a | b));  
endmodule
```

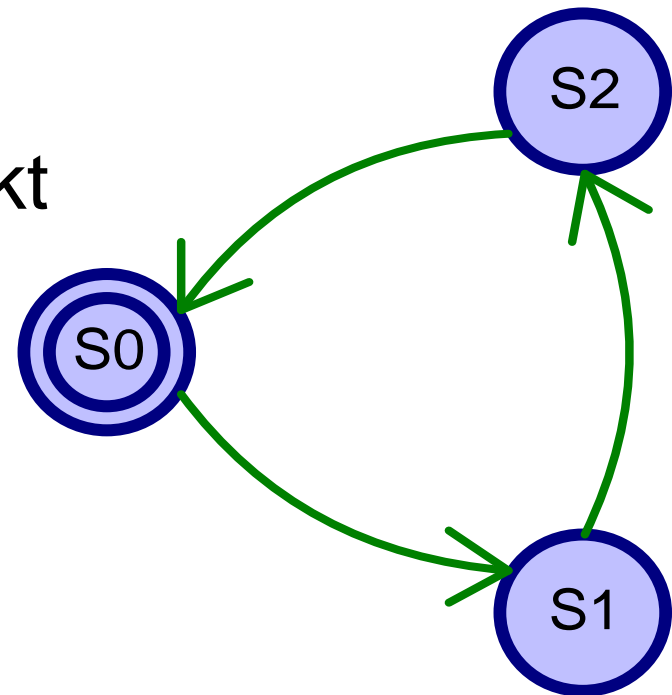

Beispiel-FSM: Dritteln der Taktfrequenz

■ Eingabe:

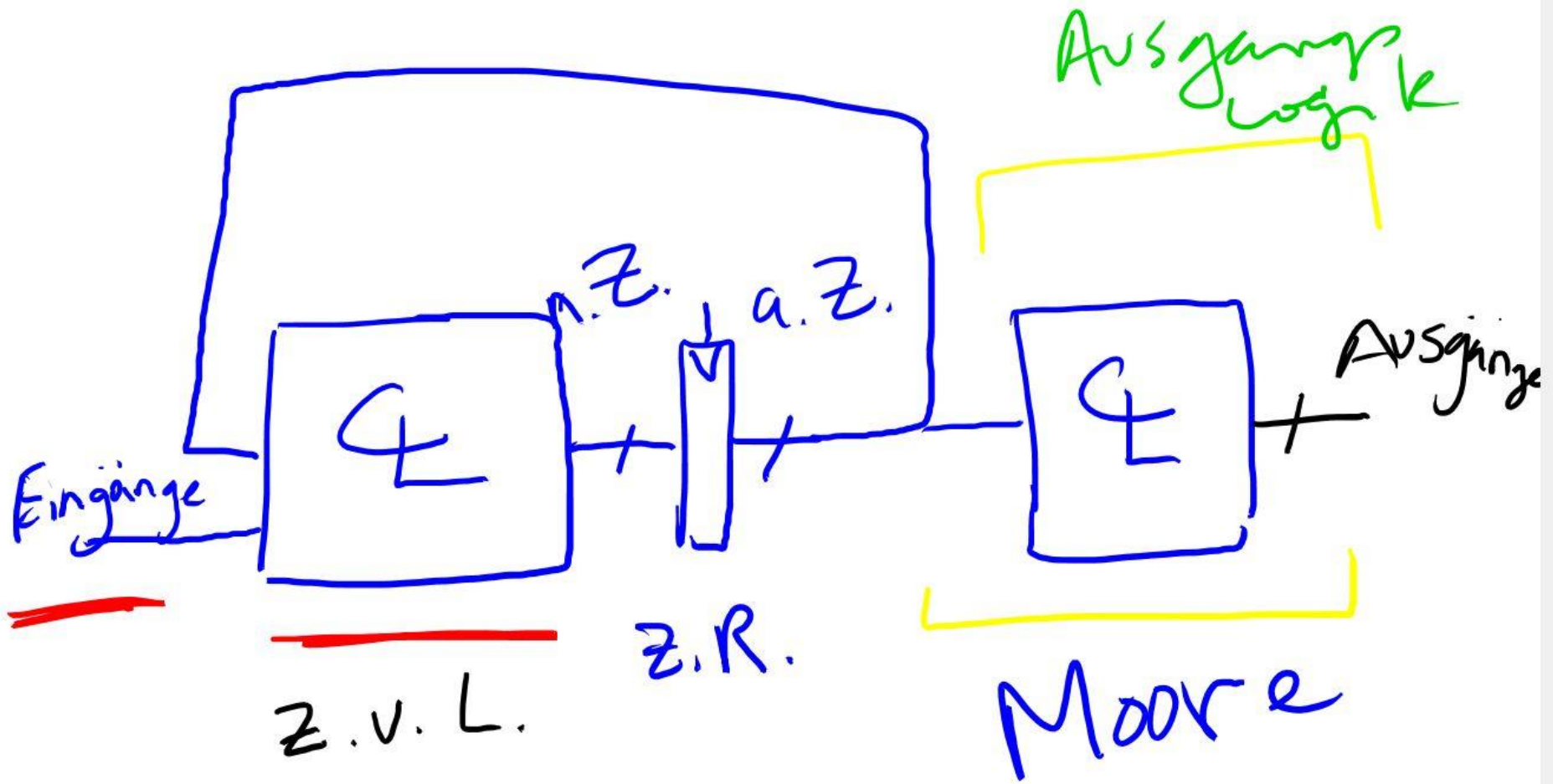
- Explizit kein Signal
- Implizit den Schaltungstakt
 - Mit Frequenz f

■ Ausgabe:


- Signal q_1 mit Frequenz $f/3$



Hier: alternative Schreibweise
für Resetzustand (doppelter Kreis)

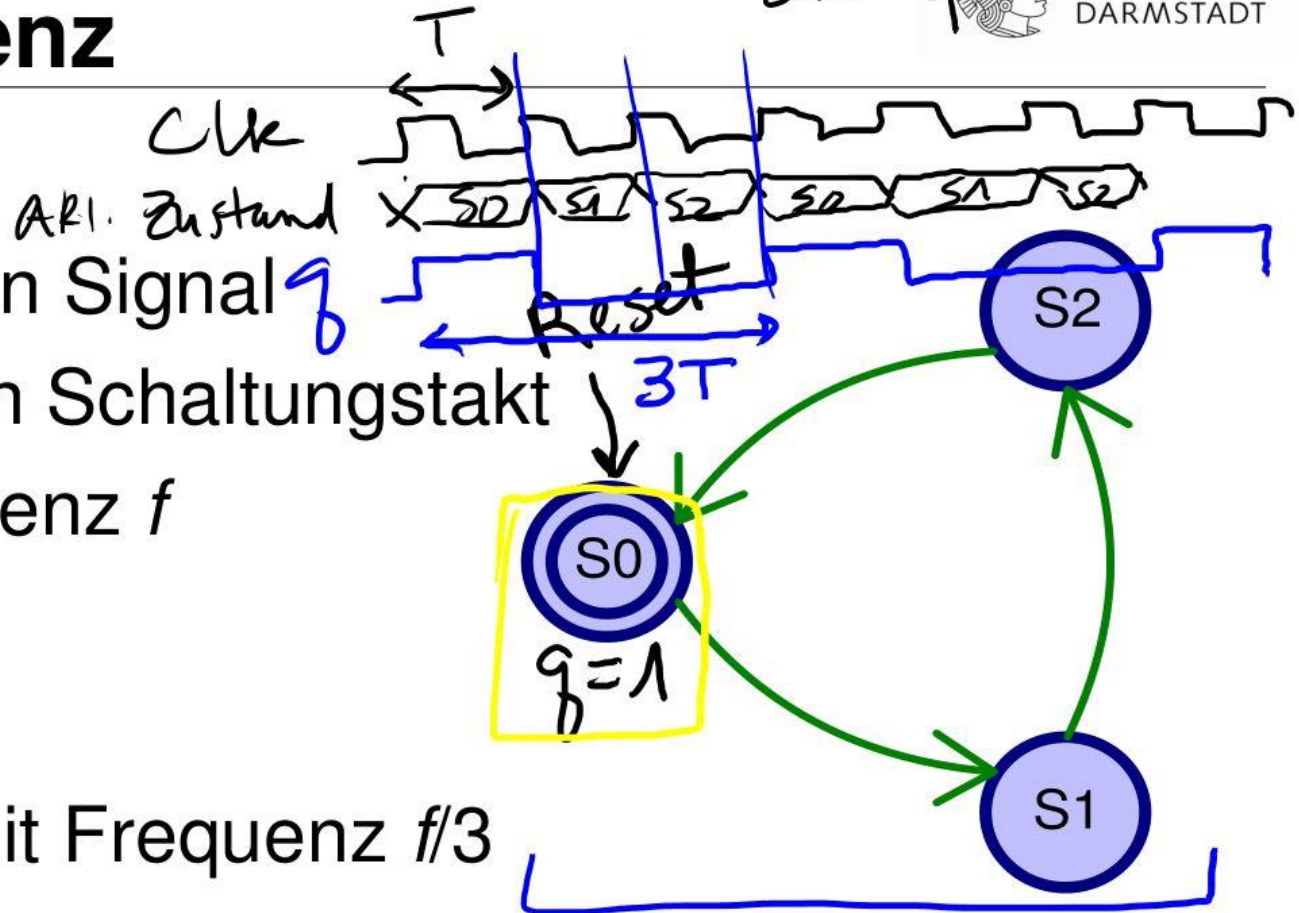


Beispiel-FSM: Dritteln der Taktfrequenz

$$f_{clk} = \frac{1}{T}$$


■ Eingabe:

- Explizit kein Signal g
- Implizit den Schaltungstakt
 - Mit Frequenz f



■ Ausgabe:

- Signal q mit Frequenz $f/3$

$$f_q = \frac{1}{3T}$$

Hier: alternative Schreibweise
für Resetzustand (doppelter Kreis)
(b.z.w. Startzustand)

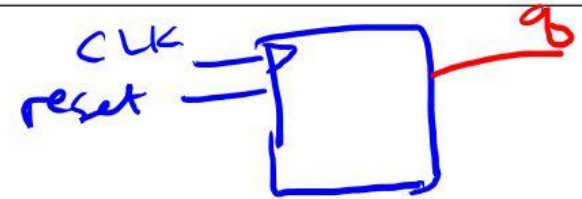
FSM in SystemVerilog



```
module divideby3FSM(input  logic clk,  
                    input  logic reset,  
                    output logic q);  
  
    typedef enum logic [1:0] {S0, S1, S2} statetype;  
    statetype state, nextstate;  
  
    always_ff @(posedge clk, posedge reset) // Zustandsregister  
        if (reset) state <= S0;  
        else      state <= nextstate;  
  
    always_comb // Zustandsübergangslogik  
        case (state)  
            S0:      nextstate = S1;  
            S1:      nextstate = S2;  
            S2:      nextstate = S0;  
            default: nextstate = S0;  
        endcase  
  
    assign q = (state == S0); // Ausgangslogik  
  
endmodule
```

System FSM in Verilog

```
module divideby3FSM(input logic clk,  
                    input logic reset,  
                    output logic q);
```



```
typedef enum logic [1:0] {S0, S1, S2} statetype;  
statetype state, nextstate;
```

```
always_ff @(posedge clk, posedge reset) // Zustandsregister
```

```
    if (reset) state <= S0; ←
```

```
    else state <= nextstate;
```

```
always_comb
```

```
    case (state)
```

```
        S0: nextstate = S1;
```

```
        S1: nextstate = S2;
```

```
        S2: nextstate = S0;
```

```
        default: nextstate = S0;
```

```
    endcase
```

```
    assign q = (state == S0);
```

```
endmodule
```

```
// Zustandsübergangslogik
```

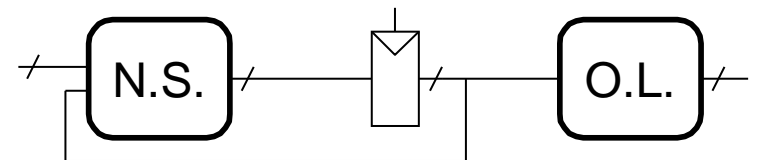
z.B.

S0	2'b00
S1	2'b01
S2	2'b10

```
// Ausgangslogik
```

FSM in Verilog

```
module divideby3FSM(input  logic clk,  
                    input  logic reset,  
                    output logic q);  
  
    typedef enum logic [1:0] {S0, S1, S2} statetype;  
    statetype state, nextstate;  
  
    always_ff @(posedge clk, posedge reset) // Zustandsregister  
        if (reset) state <= S0;  
        else      state <= nextstate;  
  
    always_comb // Zustandsübergangslogik  
        case (state)  
            S0:      nextstate = S1;  
            S1:      nextstate = S2;  
            S2:      nextstate = S0;  
            default: nextstate = S0;  
        endcase  
  
    assign q = (state == S0); // Ausgangslogik  
  
endmodule
```



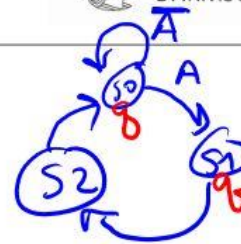
FSM in Verilog

```

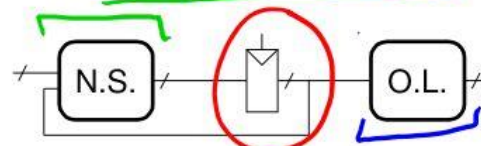
module divideby3FSM(input  logic clk, a,
                   input  logic reset,
                   output logic q);
    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype state, nextstate;

    always_ff @(posedge clk, posedge reset) // Zustandsregister
        if (reset) state <= S0;
        else     state <= nextstate;

    always_comb
        case (state)
            S0: nextstate = S1;
            S1: nextstate = S2;
            S2: nextstate = S0;
            default: nextstate = S0;
        endcase
        assign q = (state == S0 | S1);
    endmodule
    
```



// Zustandsübergangslogik



// Ausgangslogik

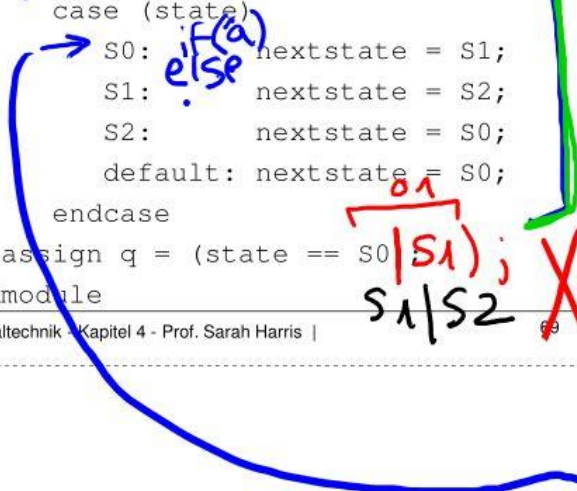
oder

assign q = (state != S2);

assign q = (state == S0 | state == S1);

$01 + 10 = 11$

~~nicht richtig~~



if(a)
else

01
| S1
S1/S2

FSM in Verilog



```
module divideby3FSM(input  logic clk,  
                    input  logic reset,  
                    output logic q);  
  
    parameter S0 = 2'b00;  
    parameter S1 = 2'b01;  
    parameter S2 = 2'b10;  
  
    logic [1:0] state, nextstate;  
  
    always_ff @(posedge clk, posedge reset) // Zustandsregister  
        if (reset) state <= S0;  
        else      state <= nextstate;  
  
    always_comb // Zustandsübergangslogik  
        case (state)  
            S0:      nextstate = S1;  
            S1:      nextstate = S2;  
            ...  
        endcase  
endmodule
```


FSM in Verilog

```
module divideby3FSM(input  logic clk,  
                    input  logic reset,  
                    output logic q);
```

```
parameter S0 = 2'b00;  
parameter S1 = 2'b01;  
parameter S2 = 2'b10;
```

```
logic [1:0] state, nextstate;
```

```
always_ff @(posedge clk, posedge reset) // Zustandsregister
```

```
    if (reset) state <= S0;
```

```
    else      state <= nextstate;
```

```
always_comb
```

```
    case (state)
```

```
        S0:      nextstate = S1;
```

```
        S1:      nextstate = S2;
```

```
        ...
```

*andere Art der
Zustandskodierung
(statt: typedef enum...)
Statetype*

```
// Zustandsübergangslogik
```

FSM in Verilog

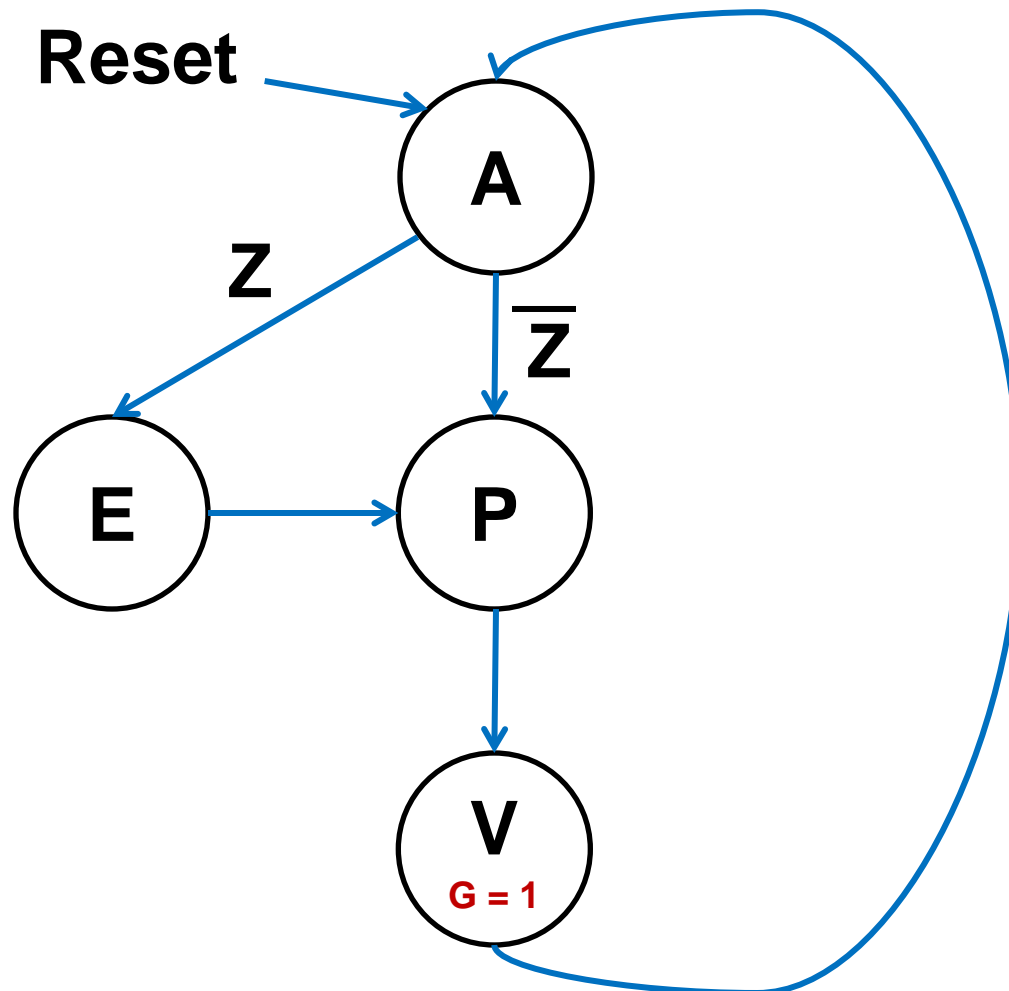


```
module divideby3FSM(input  logic clk,  
                    input  logic reset,  
                    output logic q);  
  
    typedef enum logic [1:0] {S0, S1, S2} statetype;  
    statetype state, nextstate;  
  
    always_ff @(posedge clk, posedge reset) // Zustandsregister  
        if (reset) state <= S0;  
        else      state <= nextstate;  
  
    always_comb // Zustandsübergangslogik  
        case (state)  
            S0:      nextstate = S1;  
            S1:      nextstate = S2;  
            S2:      nextstate = S0;  
            default: nextstate = S0;  
        endcase  
  
    assign q = (state == S0); // Ausgangslogik  
  
endmodule
```

FSM Beispiel 2: Zeitplan FSM



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Zustände

A = aufwachen
E = essen
P = Zähne putzen
V = Vorlesung

Eingang

Z = Zeit

Ausgang

G = glücklich

FSM Beispiel 2: Zeitplan FSM



TECHNISCHE
UNIVERSITÄT
DARMSTADT

SystemVerilog

FSM in Verilog



zeitplanFSM

```
module divideby3FSM(input logic clk,  
                      input logic reset, z,  
                      output logic g);  
    typedef enum logic [1:0] {S0, S1, S2} statetype;  
    statetype state, nextstate;  
  
    always_ff @(posedge clk, posedge reset) // Zustandsregister  
        if (reset) state <= A;  
        else state <= nextstate;  
  
    → always_comb  
        case (state)  
            A: nextstate = S1;  
            S1: nextstate = S2;  
            S2: nextstate = S0;  
            default: nextstate = S0;  
        endcase  
        assign g = (state == S0);  
endmodule
```

z, A, E, P, V

// Zustandsübergangslogik
if (**z**) nextstate = **E**;
else nextstate = **P**;
... (u.s.w.)

// Ausgangslogik

FSM Beispiel 2: Zeitplan FSM



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module zeitplanFSM(input logic clk, reset, z,
                  output logic g);

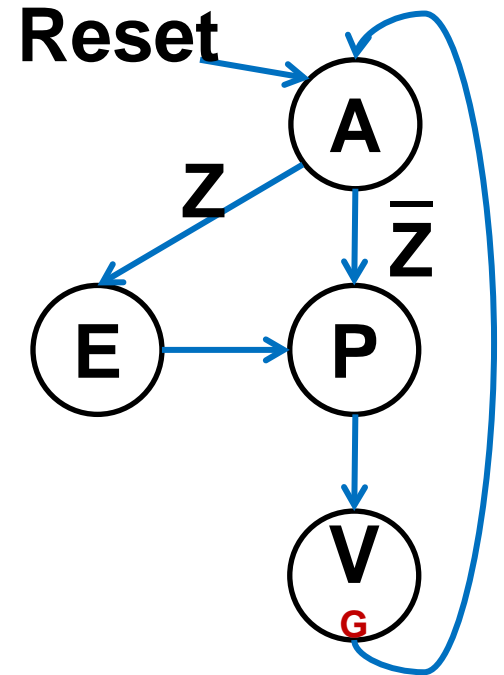
    typedef enum logic [1:0] {A, E, P, V} statetype;
    statetype state, nextstate;

    always_ff @(posedge clk, posedge reset)
        if (reset) state <= A;
        else      state <= nextstate;

    always_comb
        case (state)
            A:      if (z) nextstate = E;
                   else  nextstate = P;
            E:      nextstate = P;
            P:      nextstate = V;
            default: nextstate = A;
        endcase

    assign g = (state == V);

endmodule
```



FSM Beispiel 2: Zeitplan FSM



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module zeitplanFSM(input  logic clk, reset, z,
                   output logic g);

    typedef enum logic [1:0] {A, E, P, V} statetype;
    statetype state, nextstate;

    always_ff @(posedge clk, posedge reset) // Zustandsregister
        if (reset) state <= A;
        else      state <= nextstate;

    always_comb // Zustandsübergangslogik
        case (state)
            A:      if (z) nextstate = E;
                   else  nextstate = P;
            E:      nextstate = P;
            P:      nextstate = V;
            default: nextstate = A;
        endcase

    assign g = (state == V); // Ausgangslogik

endmodule
```

Parametrisierte Module

2:1 Multiplexer:

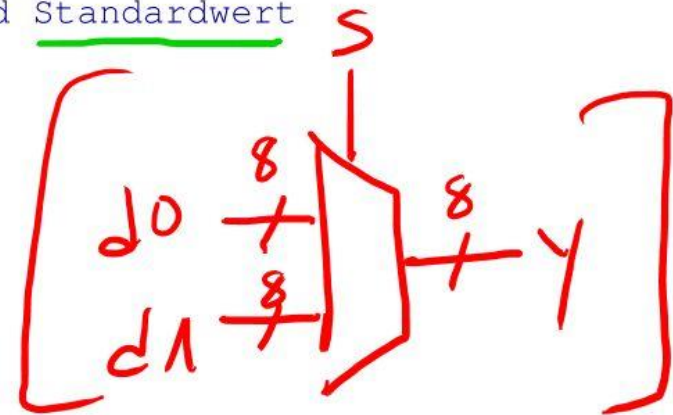
```
module mux2
  #(parameter WIDTH = 8) // Parameter: Name und Standardwert
  (input logic [WIDTH-1:0] d0, d1,
   input logic s,
   output logic [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```


Parametrisierte Module



2:1 Multiplexer:

```
module mux2
  #(parameter WIDTH = 8) // Parameter: Name und Standardwert
  (input logic [WIDTH-1:0] d0, d1,
   input logic s,
   output logic [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```



```
if (s == 1) y = d1
else y = d0
```

pseudo code

always s_comb ...
if (s) y = d1;
else y = d0;



Parametrisierte Module

2:1 Multiplexer:

```
module mux2
    #(parameter WIDTH = 8) // Parameter: Name und Standardwert
    (input logic [WIDTH-1:0] d0, d1,
     input logic          s,
     output logic [WIDTH-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
mux2 mymux(data0, data1, select, out);
```

Parametrisierte Module

2:1 Multiplexer:

```
module mux2
    #(parameter WIDTH = 8) // Parameter: Name und Standardwert
    (input logic [WIDTH-1:0] d0, d1,
     input logic          s,
     output logic [WIDTH-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
mux2 mymux(data0, data1, select, out);
```

Instanz mit 12-bit Busbreite:

```
mux2 #(12) lowmux(data0, data1, sel, out);
```

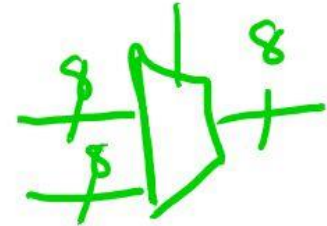
Parametrisierte Module

2:1 Multiplexer:

```
module mux2
  #(parameter WIDTH = 8) // Parameter: Name und Standardwert
  (input logic [WIDTH-1:0] d0, d1,
   input logic          s,
   output logic [WIDTH-1:0] y);
  assign y = s ? d1 : d0;
endmodule
```

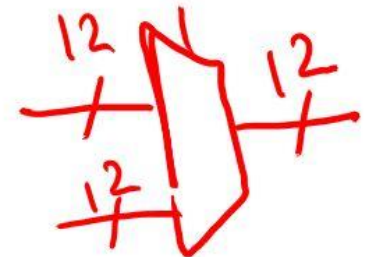
Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
mux2 mymux(data0, data1, select, out);
```



Instanz mit 12-bit Busbreite:

```
mux2 #(12) lowmux(data0, data1, sel, out);
```



auch für Verzögerung benutzt
(hängt vom Kontext ab)

Parametrisierte Module

2:1 Multiplexer:

```
module mux2
    #(parameter WIDTH = 8) // Parameter: Name und Standardwert
    (input logic [WIDTH-1:0] d0, d1,
     input logic          s,
     output logic [WIDTH-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
mux2 mymux(data0, data1, select, out);
```

Instanz mit 12-bit Busbreite:

```
mux2 #(12) lowmux(data0, data1, sel, out);
```

Aber **besser** (falls mehrere Parameter auftreten sollten):

```
mux2 #(.WIDTH(12)) lowmux(data0, data1, sel, out);
```

Testrahmen



- **HDL-Programm zum Testen eines anderen HDL-Moduls**
 - Im Hardware-Entwurf schon lange üblich
 - ... seit einigen Jahren auch im Software-Bereich (**JUnit** etc.)
- **Getestetes Modul**
 - *Device under test (DUT), Unit under test (UUT)*
- **Testrahmen werden nicht synthetisiert**
 - Nur für **Simulation** benutzt
- **Arten von Testrahmen**
 - **Einfach:** Legt nur feste Testdaten an und zeigt Ausgaben an
 - **Selbstprüfend:** Prüft auch noch, ob Ausgaben den Erwartungen entsprechen
 - **Selbstprüfend mit Testvektoren:** Auch noch mit variablen Testdaten

Testrahmen

- **HDL-Programm zum Testen eines anderen HDL-Moduls**

- Im Hardware-Entwurf schon lange üblich
- ... seit einigen Jahren auch im Software-Bereich (JUnit etc.)

- **Getestetes Modul**

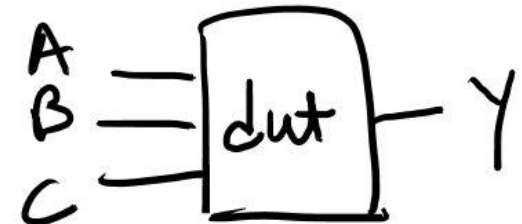
- *Device under test (DUT), Unit under test (UUT)*

- **Testrahmen werden nicht synthetisiert**

- Nur für Simulation benutzt

- **Arten von Testrahmen**

- **Einfach:** Legt nur feste Testdaten an und zeigt Ausgaben an
- **Selbstprüfend:** Prüft auch noch, ob Ausgaben den Erwartungen entsprechen
- **Selbstprüfend mit Testvektoren:** Auch noch mit variablen Testdaten



ABC 000
 001
 010
 011

111

Beispiel

Verfasse Verilog-Code um die folgende Funktion in Hardware zu berechnen:

$$y = \overline{b}\overline{c} + a\overline{b}$$

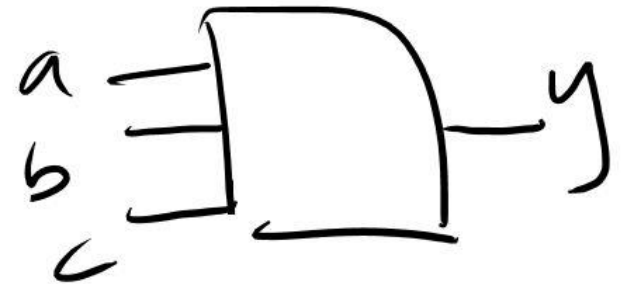
Der Modulname sei `sillyfunction`

Beispiel

Verfasse Verilog-Code um die folgende Funktion in Hardware zu berechnen:

$$y = \overline{bc} + a\overline{b}$$

Der Modulname sei `sillyfunction`



SystemVerilog

```
module sillyfunction(input  logic a, b, c,  
                    output logic y);  
    assign y = ~b & ~c | a & ~b;  
endmodule
```

Beispiel

Verfasse Verilog-Code um die folgende Funktion in Hardware zu berechnen:

$$y = \overline{b}\overline{c} + a\overline{b}$$

Der Modulname sei `sillyfunction`

SystemVerilog

```
module sillyfunction(input  logic a, b, c,  
                    output logic y);  
    assign y = ~b & ~c | a & ~b;  
endmodule
```

Einfacher Testrahmen für Beispiel



```
module testbench1();  
    logic a, b, c;  
    logic y;  
  
    // Instanz des zu testenden Moduls erzeugen  
    sillyfunction dut(a, b, c, y);  
  
    // Eingangswerte anlegen und warten  
    initial begin  
        a = 0; b = 0; c = 0; #10;  
        c = 1; #10;  
        b = 1; c = 0; #10;  
        c = 1; #10;  
        a = 1; b = 0; c = 0; #10;  
        c = 1; #10;  
        b = 1; c = 0; #10;  
        c = 1; #10;  
    end  
endmodule
```

Einfacher Testrahmen für Beispiel

```
module testbench1 ();  
  logic a, b, c;  
  logic y;
```

Keine Inputs/Outputs

d, r, f, w

```
// Instanz des zu testenden Moduls erzeugen
```

```
sillyfunction dut(a, b, c, y);
```

```
// Eingangswerte anlegen und warten
```

```
initial begin
```

```
a = 0; b = 0; c = 0; #10;
```

```
c = 1; #10;
```

```
b = 1; c = 0; #10;
```

```
c = 1; #10;
```

```
a = 1; b = 0; c = 0; #10;
```

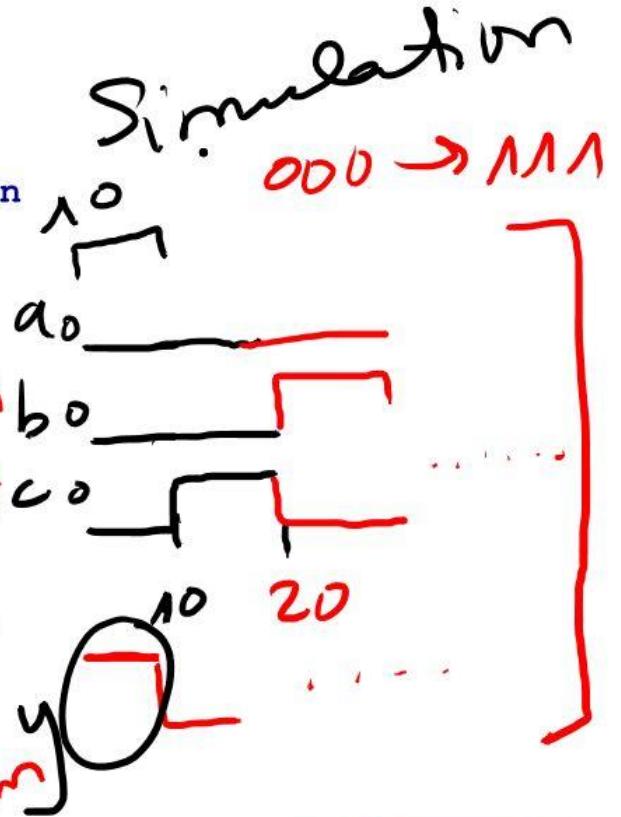
```
c = 1; #10;
```

```
b = 1; c = 0; #10;
```

```
c = 1; #10;
```

```
end
```

```
endmodule
```



Selbstprüfender Testrahmen



```
module testbench2();
  logic a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y); // Instanz des zu testenden Module erzeugen
  initial begin // Eingangswerte anlegen, warten
                // Ausgang mit erwartetem Wert überprüfen

    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

Selbstprüfender Testrahmen

```
module testbench2();
  logic a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y); // Instanz des zu testenden Module erzeugen
  initial begin                // Eingangswerte anlegen, warten
                                // Ausgang mit erwartetem Wert überprüfen
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

← interne Signale

← New

← New

← New

←

Jetzt kann
man auf den
Rechner nach
diesen Aussagen
schauen.

Selbstprüfender Testrahmen mit Testvektoren



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Trennen von **HDL-Programm** und **Testdaten**

- Eingaben
- Erwartete Ausgaben
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

Selbstprüfender Testrahmen mit Testvektoren



Trennen von HDL-Programm und Testdaten

- Eingaben *abc*
- Erwartete Ausgaben *expected*
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

abc ↖ y
000 - 1
001 - 0

Selbstprüfender Testrahmen mit Testvektoren



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Trennen von **HDL-Programm** und **Testdaten**

- Eingaben
- Erwartete Ausgaben
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

Eigene **Datei** für Vektoren

Selbstprüfender Testrahmen mit Testvektoren

Trennen von **HDL-Programm** und **Testdaten**

- Eingaben
- Erwartete Ausgaben
- Organisiere beides als Vektoren von zusammenhängenden Signalen/Werten

Eigene **Datei** für Vektoren

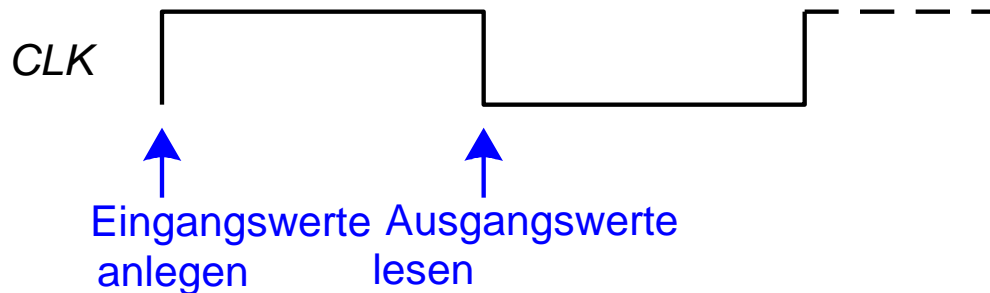
Dann HDL-Programm für **universellen** Testrahmen

1. Erzeuge **Takt** zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. **Lese** Vektordatei in SystemVerilog Array
3. **Lege** Eingangsdaten an
4. **Warte** auf Ausgabedaten, **werte** Ausgabedaten aus
5. **Vergleiche** aktuelle mit erwarteten Ausgabedaten, **melde** Fehler bei Differenz
6. Noch **weitere** Testvektoren abzuarbeiten?

Selbstprüfender Testrahmen mit Testvektoren

Im Testrahmen erzeugter Takt legt **zeitlichen** Ablauf fest

- **Steigende Flanke:** Eingabewerte aus Testvektor an **Eingänge** anlegen
- **Fallende Flanke:** Aktuelle Werte an **Ausgängen** lesen

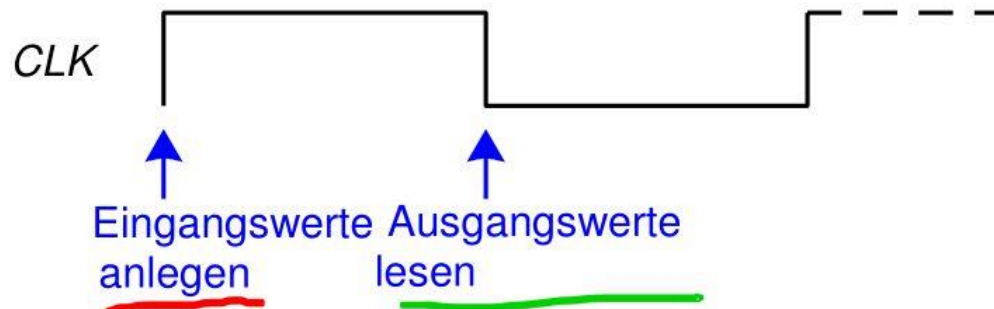


Selbstprüfender Testrahmen mit Testvektoren



Im Testrahmen erzeugter Takt legt **zeitlichen** Ablauf fest

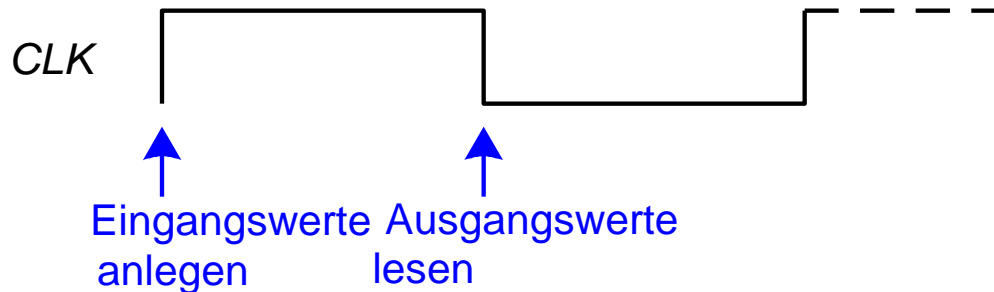
- **Steigende** Flanke: Eingabewerte aus Testvektor an **Eingänge** anlegen
- **Fallende** Flanke: Aktuelle Werte an **Ausgängen** lesen



Selbstprüfender Testrahmen mit Testvektoren

Im Testrahmen erzeugter Takt legt **zeitlichen** Ablauf fest

- **Steigende** Flanke: Eingabewerte aus Testvektor an **Eingänge** anlegen
- **Fallende** Flanke: Aktuelle Werte an **Ausgängen** lesen



Takt kann auch als Takt für **sequentielle synchrone Schaltungen** verwendet werden

Einfaches Textformat für Testvektordateien



Datei: `example.tv`

`000_1`

`001_0`

`010_0`

`011_0`

`100_1`

`101_1`

`110_0`

`111_0`

Aufbau:

Eingangsdaten “_” erwartete Ausgangsdaten

Einfaches Textformat für Testvektordateien



Datei: example.tv

abc y .txt

000_1	}
001_0	
010_0	
011_0	
100_1	
101_1	
110_0	
111_0	

binär

Aufbau:

Eingangsdaten “_” erwartete Ausgangsdaten

Selbstprüfender Testrahmen mit Testvektoren



TECHNISCHE
UNIVERSITÄT
DARMSTADT

HDL-Programm für **universellen** Testrahmen

1. Erzeuge **Takt** zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. **Lese** Vektordatei in SystemVerilog Array
3. **Lege** Eingangsdaten an
4. **Warte** auf Ausgabedaten, **werte** Ausgabedaten aus
5. **Vergleiche** aktuelle mit erwarteten Ausgabedaten, **melde** Fehler bei Differenz
6. Noch **weitere** Testvektoren abzuarbeiten?

Testrahmen: 1. Erzeuge Takt



```
module testbench3();  
    logic        clk, reset;  
    logic        a, b, c, yexpected;  
    logic        y;  
    logic [31:0] vectornum, errors;    // Verwaltungsdaten  
    logic [3:0]  testvectors[10000:0]; // Array für Testvektoren
```

// Instanz der Testschaltung erzeugen

```
sillyfunction dut (a, b, c, y);
```

// Takterzeugung

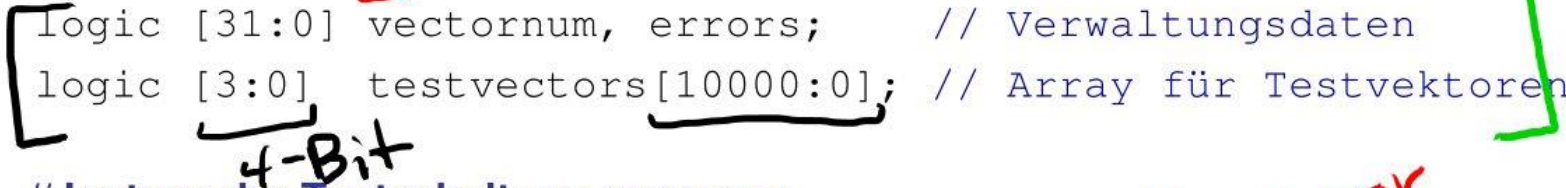
```
always    // Hängt von keinen anderen Signalen ab:  
          // Wird immer ausgeführt!
```

```
begin  
    clk = 1; #5; clk = 0; #5;  
end
```

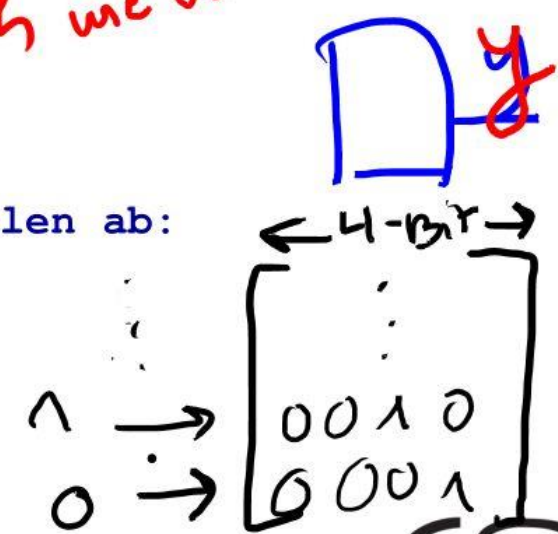
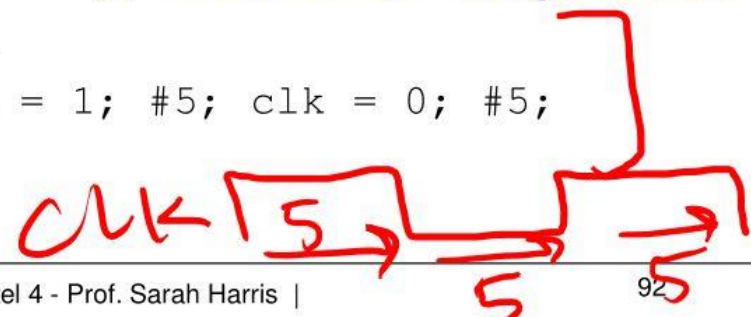
Testrahmen: 1. Erzeuge Takt



```
module testbench3();  
  logic      clk, reset;   
  logic      a, b, c, yexpected;   
  logic      y;   
  logic [31:0] vectornum, errors; // Verwaltungsdaten  
  logic [3:0] testvectors[10000:0]; // Array für Testvektoren  
  // Instanz der Testschaltung erzeugen  
  sillyfunction dut (a, b, c, y);  
  
  // Takterzeugung  
  always // Hängt von keinen anderen Signalen ab:  
         // Wird immer ausgeführt!  
  begin  
    clk = 1; #5; clk = 0; #5;  
  end
```



] gleich wie vor



2. Lese Testvektordatei in Array ein

...

// Zu Beginn der Simulation:

// Testdaten einlesen und einen Reset-Impuls erzeugen

```
initial // Block wird genau einmal ausgeführt
begin
    $readmemb("example.tv", testvectors);
    vectornum = 0; errors = 0; // Verwaltungsdaten initialisieren
    reset = 1; #27; reset = 0; // Reset-Impuls erzeugen
end
```

...

Hinweis: Falls **hexadezimale** Testvektoren verwendet werden sollen, statt `$readmemb` den Aufruf `$readmembh` verwenden

2. Lese Testvektordatei in Array ein

...
// Zu Beginn der Simulation:
// Testdaten einlesen und einen Reset-Impuls erzeugen

initial // Block wird genau einmal ausgeführt

begin

→ \$readmemb("example.tv", testvectors);

vectornum = 0; errors = 0; // **Verwaltungsdaten initialisieren**

→ reset = 1; #27; reset = 0; // **Reset-Impuls erzeugen**

end

...



Hinweis: Falls **hexadezimale** Testvektoren verwendet werden sollen, statt \$readmemb den Aufruf \$readmemh verwenden

z.B. 128 Bits breit binär: 01010000...
hexadezi: 5 0...

3. Lege Testdaten an Eingänge an



...

// zur steigenden Taktflanke (genauer: kurz danach!)

```
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

...

a, b, c sind **Eingänge** der DUT

$y_{expected}$ ist eine **Hilfsvariable**, die nun den erwarteten Ausgangswert dieses Vektors enthält.

3. Lege Testdaten an Eingänge an

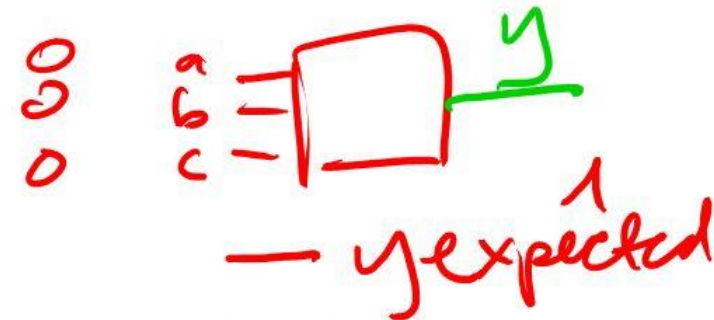
...

// zur steigenden Taktflanke (genauer: kurz danach!)

```
always @(posedge clk)
begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
end
```

...

a, b, c sind **Eingänge** der DUT



yexpected ist eine **Hilfsvariable**, die nun den erwarteten Ausgangswert dieses Vektors enthält.

4. Warte auf Ausgabedaten, lese Ausgabedaten

5. Vergleiche aktuelle Ausgaben mit erwarteten Werten



...

// warte auf fallende Flanke zum Lesen der Ausgabedaten der DUT

```
always @(negedge clk)
```

// nur Prüfen, nachdem Schaltung schon initialisiert

```
if (~reset) begin
```

// vergleiche aktuelle Ausgabe mit erwartetem Wert

```
if (y !== yexpected) begin
```

// Fehlermeldung

```
$display("Fehler: Eingänge = %b", {a, b, c});
```

```
$display("  Ausgänge = %b (%b erwartet)", y, yexpected);
```

```
errors = errors + 1; // zähle Fehler
```

```
end
```

...

Hinweis: Um Werte **hexadezimal** auszugeben, Formatkennung %h verwenden

Beispiel: `$display("Error: Eingänge = %h", {a, b, c});`

4. Warte auf Ausgabedaten, lese Ausgabedaten

5. Vergleiche aktuelle Ausgaben mit erwarteten Werten



...

// warte auf fallende Flanke zum Lesen der Ausgabedaten der DUT

always @(negedge clk)

// nur Prüfen, nachdem Schaltung schon initialisiert

→ if (~reset) begin

// vergleiche aktuelle Ausgabe mit erwartetem Wert

if (y != yexpected) begin

// Fehlermeldung

\$display("Fehler: Eingänge = %b", {a, b, c}); ←

\$display(" Ausgänge = %b (%b erwartet)", y, yexpected); ←

errors = errors + 1; **// zähle Fehler**

end

...

Hinweis: Um Werte hexadezimal auszugeben, Formatkennung %h verwenden

Beispiel: \$display("Error: Eingänge = %h", {a, b, c});

6. Sind noch weitere Testvektoren abzuarbeiten?

...

```
// Array-Index zum Zugriff auf nächsten Testvektor erhöhen
```

```
vectornum = vectornum + 1;
```

```
// Ist der nächste schon ein ungültiger Testvektor?
```

```
if (testvectors[vectornum] === 4'bx) begin
```

```
    // Endmeldung ausgeben
```

```
    $display("%d Tests bearbeitet mit %d Fehlern",  
            vectornum, errors);
```

```
    $finish;    // Simulation anhalten
```

```
end
```

```
end
```

```
endmodule
```

Hinweis: Zum Vergleichen auf **X** und **Z** müssen die Operatoren **===** und **!==** benutzt werden

6. Sind noch weitere Testvektoren abzuarbeiten?

...

```
// Array-Index zum Zugriff auf nächsten Testvektor erhöhen
vectornum = vectornum + 1; ^
// Ist der nächste schon ein ungültiger Testvektor?
if (testvectors[vectornum] === 4'bx) begin ←
    // Endmeldung ausgeben
    $display("%d Tests bearbeitet mit %d Fehlern",
             vectornum, errors);
    $finish; // Simulation anhalten
end
end
endmodule
```

Handwritten notes and diagrams:

- A green arrow points to the line `vectornum = vectornum + 1;`.
- A red arrow points to the `===` operator in the `if` statement.
- A red arrow points to the `4'bx` value in the `if` statement.
- A green arrow points to the `$display` function call.
- A green arrow points to the `$finish;` line.
- A red arrow points to the `vectornum` variable in the `$display` call.
- A red arrow points to the `errors` variable in the `$display` call.
- A green box highlights the `$display` and `$finish;` lines.
- A red arrow points to the `vectornum` variable in the `$display` call.
- A green arrow points to the `0` value in the `vectornum` variable in the `$display` call.
- A handwritten diagram shows a vertical list of values: `LEER`, `1111`, `0010`, and `0001`. A red arrow points to the `0010` value, and a green arrow points to the `0001` value. A red arrow points to the `4'bx` value above the list.

Hinweis: Zum Vergleichen auf **X** und **Z** müssen die Operatoren `===` und `!==` benutzt werden

Selbstprüfender Testrahmen mit Testvektoren

HDL-Programm für **universellen** Testrahmen

1. Erzeuge **Takt** zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. **Lese** Vektordatei in SystemVerilog Array
3. **Lege** Eingangsdaten an
4. **Warte** auf Ausgabedaten, **werte** Ausgabedaten aus
5. **Vergleiche** aktuelle mit erwarteten Ausgabedaten, **melde** Fehler bei Differenz
6. Noch **weitere** Testvektoren abzuarbeiten?

Selbstprüfender Testrahmen mit Testvektoren

HDL-Programm für **universellen** Testrahmen

1. Erzeuge **Takt** zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
2. **Lese** Vektordatei in SystemVerilog Array ←
3. **Lege** Eingangsdaten an ↗
4. **Warte** auf Ausgabedaten, **werte** Ausgabedaten aus ↘
5. **Vergleiche** aktuelle mit erwarteten Ausgabedaten, **melde** Fehler bei Differenz
6. Noch **weitere** Testvektoren abzuarbeiten?



SystemVerilog

Sprachkonstrukte in DT



- Vor Testrahmen alle für die Beschreibung von **echter** Hardware relevanten eingeführt
 - **Schaltungssynthese**
- SystemVerilog kann viel **mehr**
 - Angedeutet beim Testrahmen (Dateioperationen, Ein/Ausgabe, ...)
 - Aber in der Regel **nicht** mehr in Hardware synthetisierbar
 - Nicht Schwerpunkt **dieser** Veranstaltung
- Mehr Details in Kanonik Computer Microsystems
 - Im Sommersemester, dann SystemVerilog und BlueSpec-Erweiterungen
- In Digitaltechnik soll dieser Kurzüberblick reichen
 - Bei akutem Bedarf werden noch weitere Konstrukte eingeführt

Verilog

- Verwendet in 1. Auflage des Lehrbuchs von Harris & Harris
- Verilog ist im Rahmen der Veranstaltung **sehr ähnlich** zu SystemVerilog
 - Teilweise **komplizierter**
 - Verwendet separate Datentypen **wire** und **reg** statt **logic** Typ
 - Benutzt keine **always** Variationen (stattdessen **always @ ...**)
 - **Flip-Flops:** **always @(posedge clk)** statt **always_ff @(posedge clk)**
 - **Latches:** **always @(clk, d)** statt **always_latch**
 - **Kombinatorische Logik:** **always @(*)** statt **always_comb**
 - Übersicht der Unterschiede im Buch (2. Auflage) im Kapitel 4.7.1
 - Für Übungen und Klausuren relevant: **SystemVerilog** (gezeigt auf Folien!)

- Verwendet in 1. Auflage des Lehrbuchs von Harris & Harris
- Verilog ist im Rahmen der Veranstaltung **sehr ähnlich** zu SystemVerilog
 - Teilweise **komplizierter**
 - Verwendet separate Datentypen **wire** und **reg** statt **logic** Typ
 - Benutzt keine `always` Variationen (stattdessen `always @ ...`)
 - **Flip-Flops:** `always @(posedge clk)` statt `always_ff @(posedge clk)`
 - **Latches:** `always @(clk, d)` statt `always_latch`
 - **Kombinatorische Logik:** `always @(*)` statt `always_comb`
 - Übersicht der Unterschiede im Buch (2. Auflage) im Kapitel 4.7.1
 - Für Übungen und Klausuren relevant: **SystemVerilog** (gezeigt auf Folien!)