

Technische Grundlagen der Informatik – Kapitel 4



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Prof. Dr. Andreas Koch
Fachbereich Informatik
TU Darmstadt



Kapitel 4: Themen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

- **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

- Fast alle **kommerziellen** Hardware-Entwürfe mit HDLs realisiert
- **Zwei** HDLs haben sich durchgesetzt
- Sie werden **beide** lernen müssen!
 - Es gibt keinen klaren Gewinner

- 1984 von der Fa. 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 an den Informatik-Fachgebieten
 - **Rechnerarchitektur** (RA, Prof. Hoffmann)
 - **Eingebettete Systeme und ihre Anwendungen** (ESA, Prof. Koch)

- *Very High-Speed Integrated Circuit 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 am Informatik-Fachgebiet
 - Integrierte Schaltungen und Systeme (ISS, Prof. Huss)

In dieser Iteration der Vorlesung



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- In den **Vorlesungen** Verilog
 - Häufig kompakter zu schreiben
 - Eher auf Einzelfolien darstellbar
- In den **Übungen** auch VHDL
- 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

▪ 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



TECHNISCHE
UNIVERSITÄT
DARMSTADT

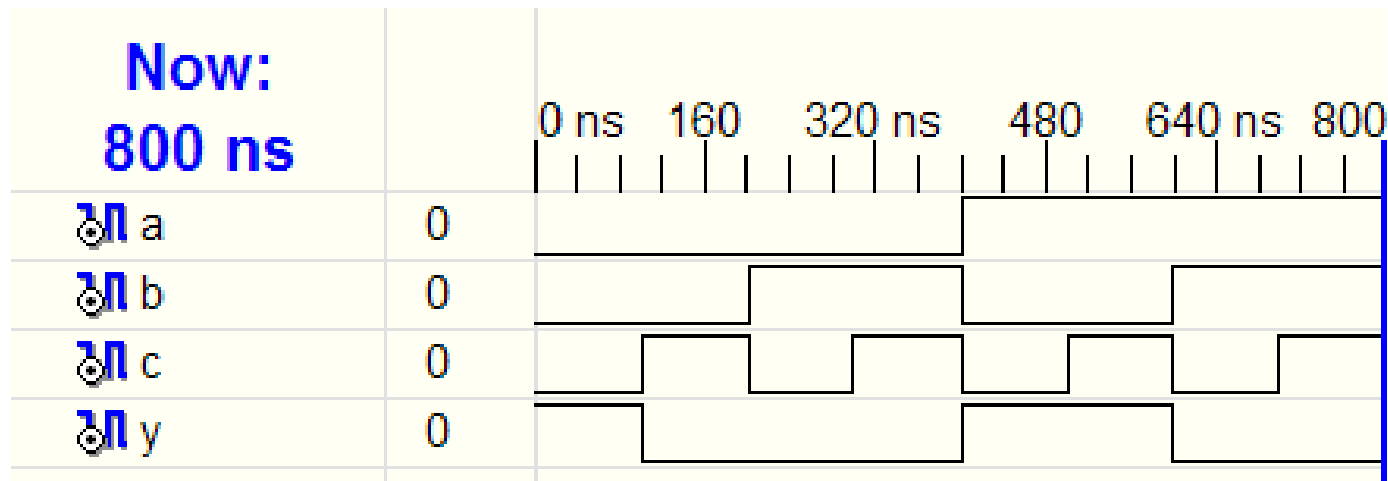
Verilog:

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

Verilog:

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

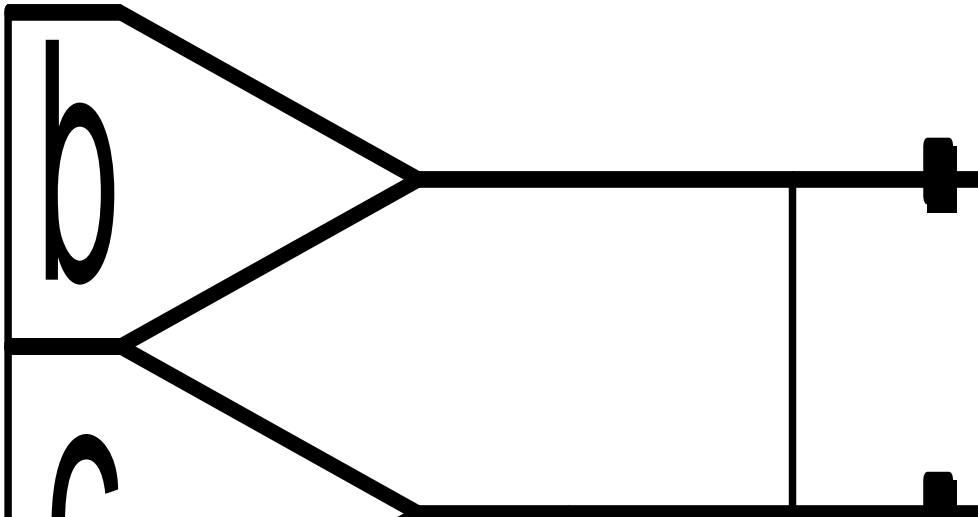
Signalverlaufdiagramm (*waves*)



Verilog:

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

Syntheseergebnis:



- **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 `*/`

Sehr **ähnlich** zu C und Java!

Strukturelle Beschreibung: Modulhierarchie



TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

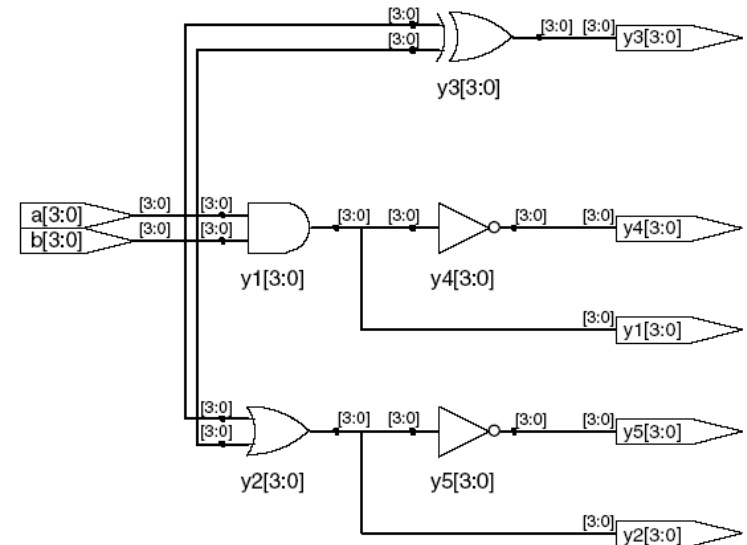
```
module inv (input  a,  
           output y);  
    assign y = ~a;  
endmodule
```

```
module nand3 (input  a, b, c  
             output y);  
    wire 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
```

Bitweise Verknüpfungsoperatoren

```
module gates (input  [3:0]  a, b,  
               output [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
```

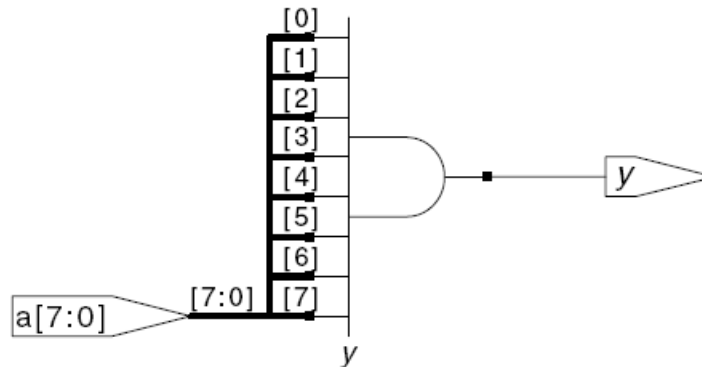
```
//      Kommentar bis zum Zeilenende  
/*...*/ Mehrzeiliger Kommentar
```



Reduktionsoperatoren

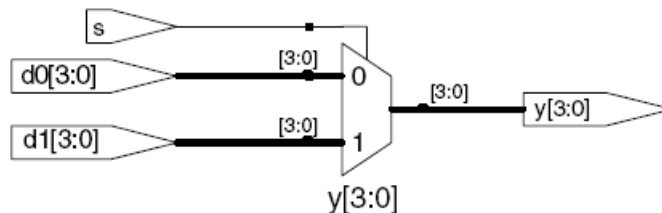


```
module and8 (input  [7:0] a,  
              output          y);  
    assign y = &a;  
    // &a ist Abkürzung für  
    // assign y = a[7] & a[6] & a[5] & a[4] &  
    //          a[3] & a[2] & a[1] & a[0];  
endmodule
```



Bedingte Zuweisung

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



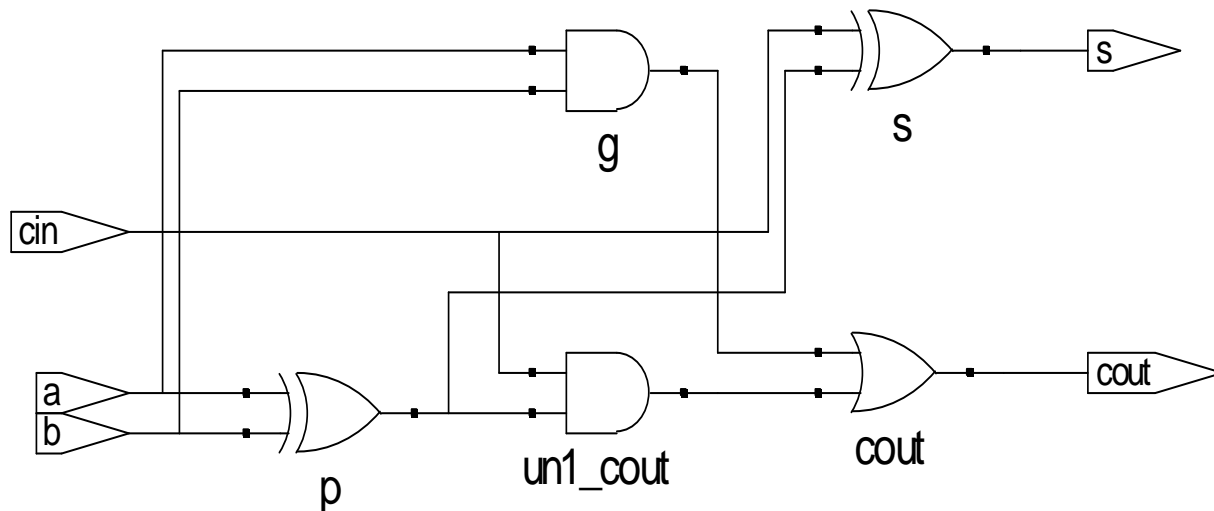
? : ist ein **ternärer** Operator, da er **drei** Operanden miteinander verknüpft: s , $d1$, und $d0$.

Interne Verbindungsknoten oder Signale



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module fulladder (input  a, b, cin, output s, cout);  
  wire p, g;          // interne Verbindungsknoten ("Drähte")  
  
  assign p = a ^ b;  
  assign g = a & b;  
  
  assign s = p ^ cin;  
  assign cout = g | (p & cin);  
endmodule
```



Bindung von Operatoren (Präzedenz)

Bestimmt Ausführungsreihenfolge

Höchste

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

Niedrigste

Syntax: $N'B_{\text{wert}}$

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



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
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
```

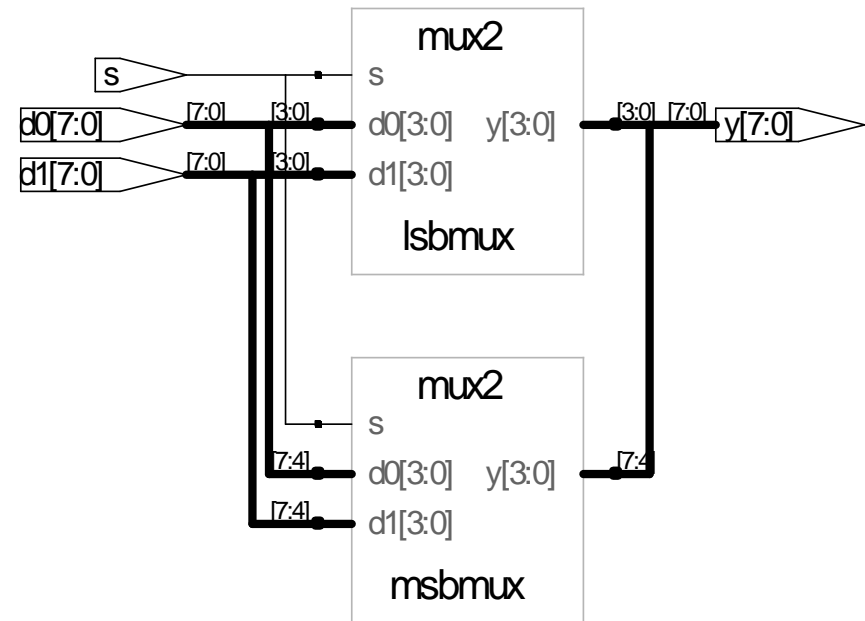
Unterstriche (__) in numerischen Konstanten dienen nur der besseren Lesbarkeit, sie werden von Verilog **ignoriert**

Operationen auf Bit-Ebene: Beispiel 2

Verilog:

```
module mux2_8 (input  [7:0] d0, d1,  
               input      s,  
               output [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
```

Synthese:



Hochohmiger Ausgang: Z

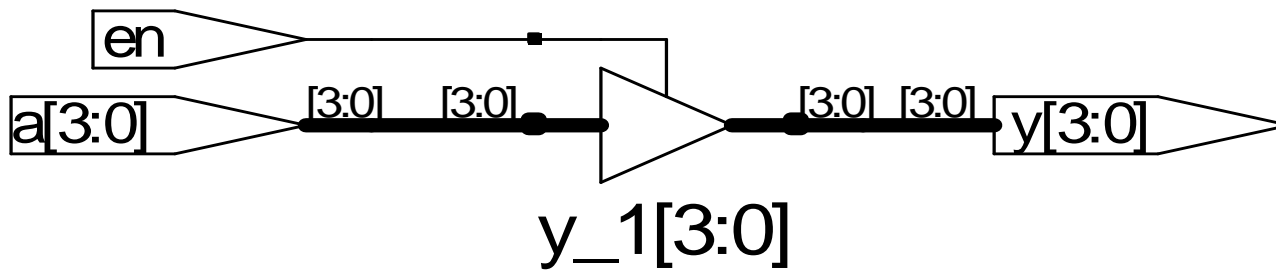


TECHNISCHE
UNIVERSITÄT
DARMSTADT

Verilog:

```
module tristate (input  [3:0] a,  
                 input      en,  
                 output [3:0] y);  
    assign y = en ? a : 4'bz;  
endmodule
```

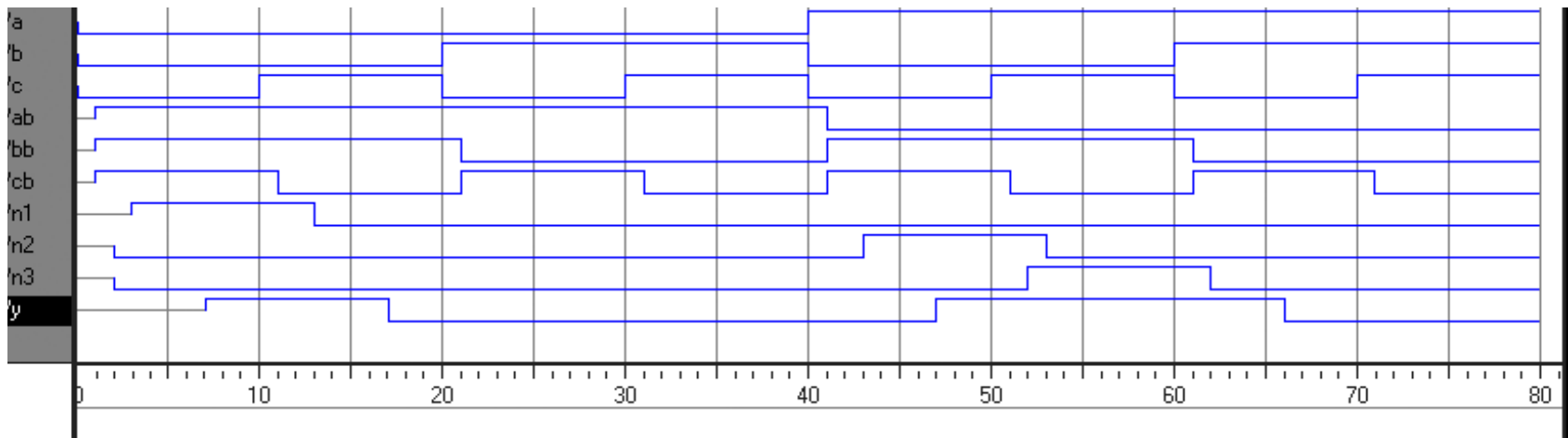
Synthese:



Verzögerungen: # Zeiteinheiten

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

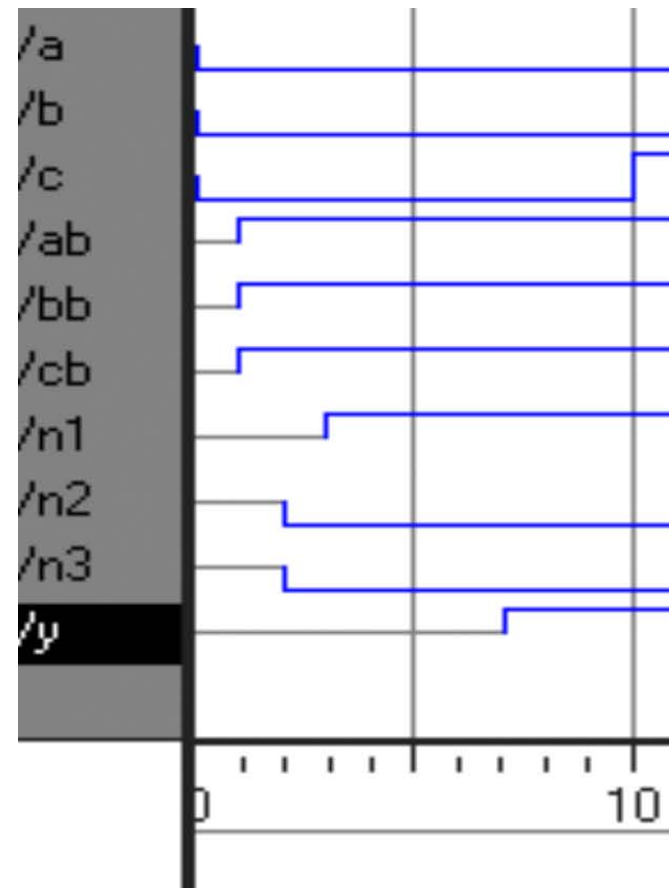


Verzögerungen



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



- 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!

Allgemeiner Aufbau:

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

Interpretation:

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

Werte: In der Regel Signale, manchmal noch erweitert

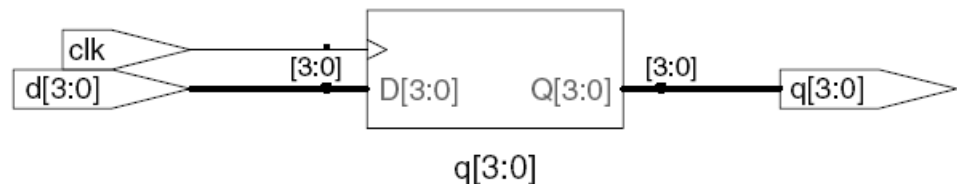
D Flip-Flop

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

```
    always @ (posedge clk)  
        q <= d;
```

// gelesen als "q übernimmt d"

```
endmodule
```



Jedes Signal, an das innerhalb von einer `always`-Anweisung zugewiesen wird,
muss als `reg` deklariert sein
- Im Beispiel: `q`

Wichtig: So ein Signal wird bei der Synthese **nicht** zwangsläufig in ein Hardware-Register abgebildet!

Rücksetzbares D Flip-Flop

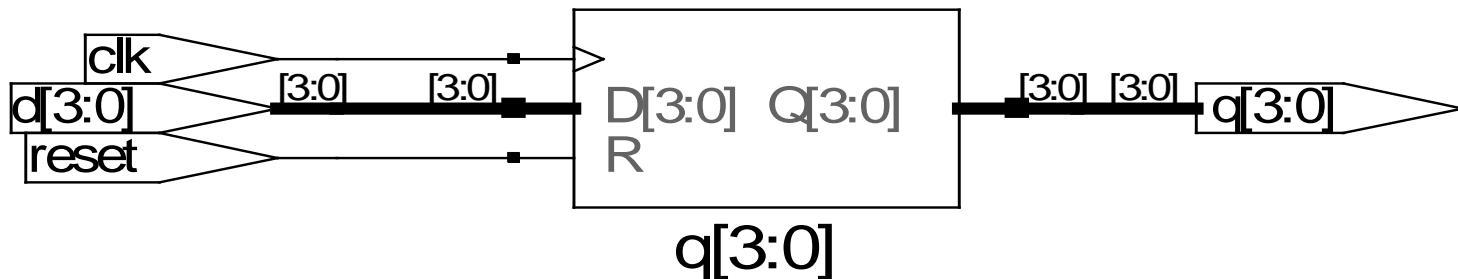


```
module flopr (input          clk,  
              input          reset,  
              input  [3:0] d,  
              output reg [3:0] q);
```

```
// synchroner Reset
```

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

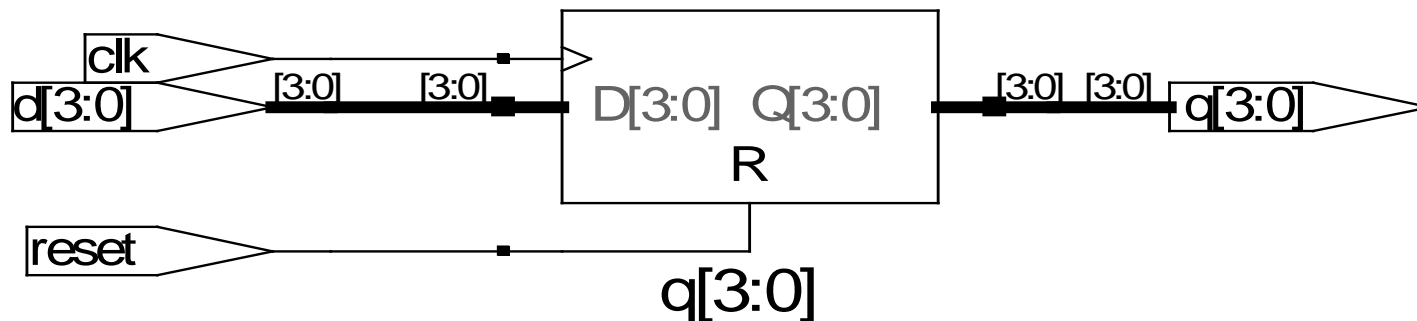
```
endmodule
```



Rücksetzbares D Flip-Flop



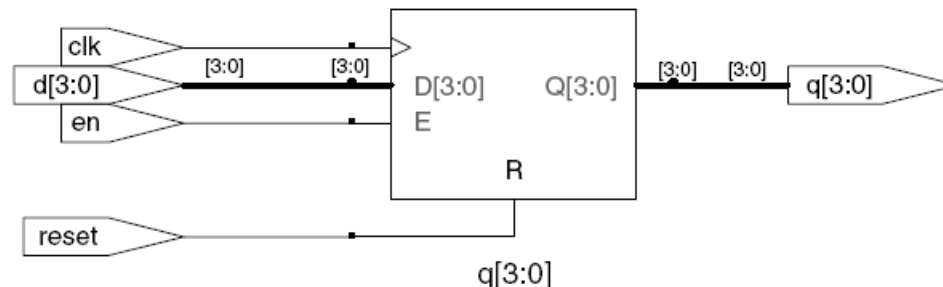
```
module flopr (input          clk,  
              input          reset,  
              input  [3:0] d,  
              output reg [3:0] q);  
  
    // asynchroner Reset  
    always @ (posedge clk, posedge reset)  
        if (reset) q <= 4'b0;  
        else      q <= d;  
  
endmodule
```



Rücksetzbares D Flip-Flop mit Taktfreigabe



```
module flopren (input          clk,  
               input          reset,  
               input          en,  
               input  [3:0] d,  
               output reg [3:0] q);  
  
    // asynchroner Reset mit Clock Enable  
    always @ (posedge clk, posedge reset)  
        if      (reset) q <= 4'b0;  
        else if (en)    q <= d;  
  
endmodule
```

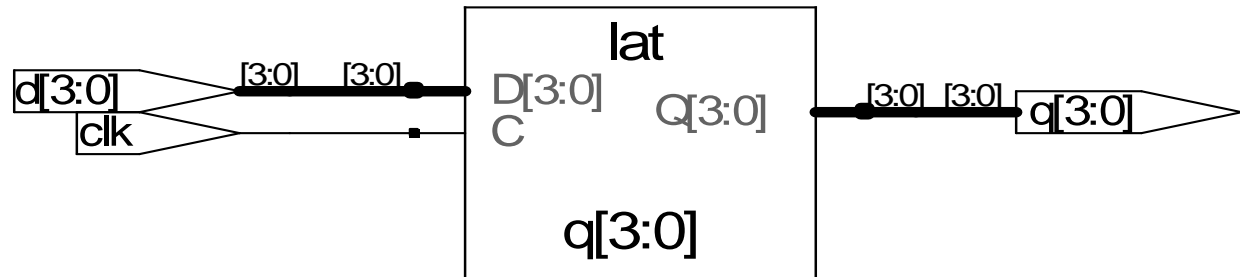


Latch

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

```
    always @ (clk, d)  
        if (clk) q <= d;
```

```
endmodule
```



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

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

Weitere Anweisungen zur Verhaltensbeschreibung

- Dürfen nur **innerhalb** von `always`-Anweisungen benutzt werden
 - `if / else`
 - `case, casez`
- **Erinnerung:**
 - Alle Zuweisungsziele innerhalb einer `always`-Anweisung **müssen** als `reg` deklariert werden!
 - Selbst, wenn sie **keine** echten Hardware-Register beschreiben

Kombinatorische Logik als `always`-Block

```
module gates (input      [3:0] a, b,  
               output reg [3:0] y1, y2, y3, y4, y5);  
  
    always @(*)                // 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
```

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

Kombinatorische Logik mit case



```
module sevensseg (input      [3:0] data,  
                  output reg [6:0] segments);  
  
always @(*) // 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

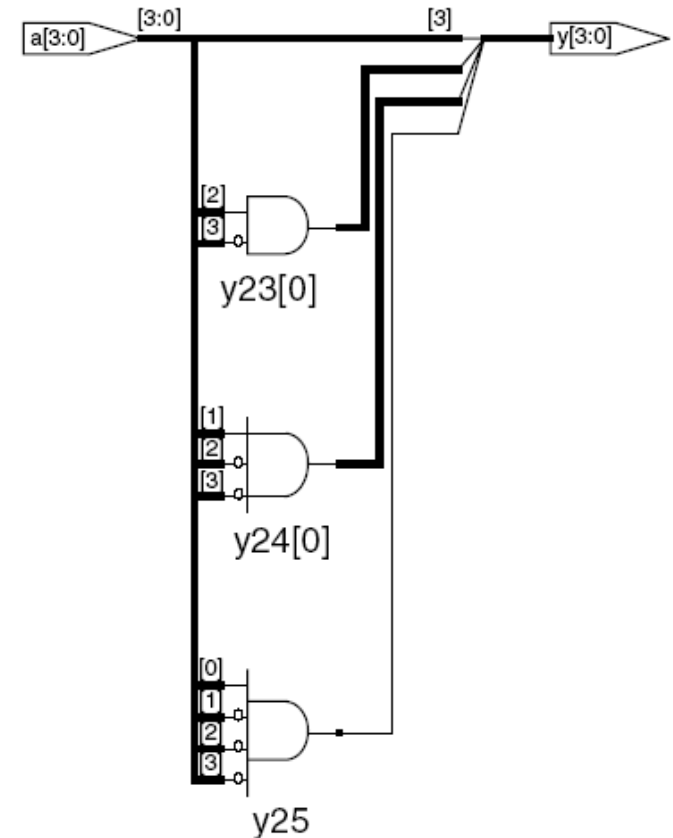
- Um kombinatorische Logik zu beschreiben, muss ein `case`-Block **alle** Möglichkeiten abdecken
 - Entweder **explizit** angeben
 - Oder einen **default-Fall** angeben
 - Tritt in Kraft, wenn sonst keine andere Alternative passt
 - Im Beispiel verwendet

Kombinatorische Logik mit casez



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module priority_casez (input      [3:0] a,  
                       output reg [3:0] y);  
  
  always @(*) // 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
```

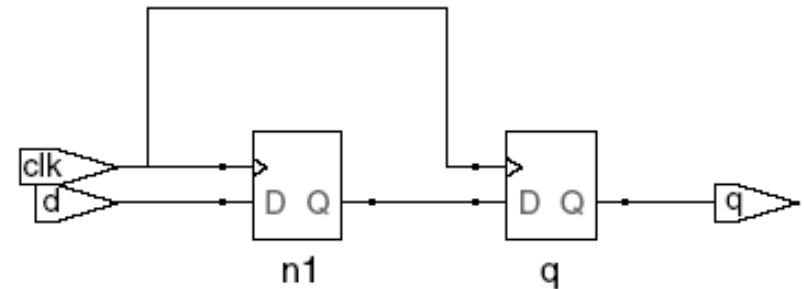


Nicht-blockende Zuweisung

- `<=` 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**
 - Am **Ende** des Blocks

// Synchronisierer mit nicht-blockenden
// Zuweisungen

```
module syncgood (input      clk,  
                 input      d,  
                 output reg q);  
  
    reg n1;  
    always @(posedge clk)  
    begin  
        n1 <= d; // nicht-blockend  
        q  <= n1; // nicht-blockend  
    end  
endmodule
```

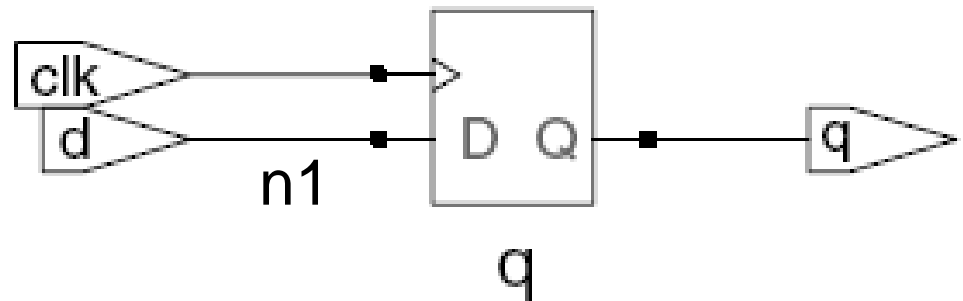


Blockende Zuweisung

- = 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    clk,  
                input    d,  
                output reg q);  
  
    reg n1;  
    always @(posedge clk)  
    begin  
        n1 = d; // blockend  
        q  = n1; // blockend  
    end  
endmodule
```



Regeln für Zuweisungen von Signalen



- Um **synchrone sequentielle** Logik zu beschreiben, benutzen Sie immer

- `always @(posedge clk)`
- **Nicht-blockende** Zuweisungen

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

- Um **einfache kombinatorische** Logik zu beschreiben, benutzen Sie immer

- **Ständige** Zuweisung (*continuous assignment*)

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

- Um **komplexere kombinatorische** Logik zu beschreiben, benutzen Sie immer

- `always @ (*)`
- **Blockende** Zuweisungen

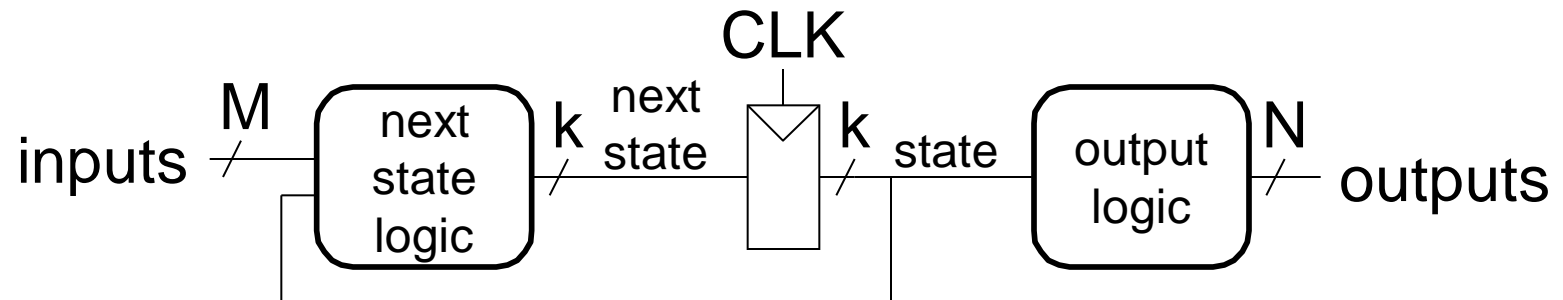
- Weisen Sie **nicht** an ein Signal

- ... in **mehreren** `always`-Blöcken zu
- ... in einem `always`-Block **gemischt** mit `=` und `<=` zu

Endliche Zustandsautomaten (FSM)

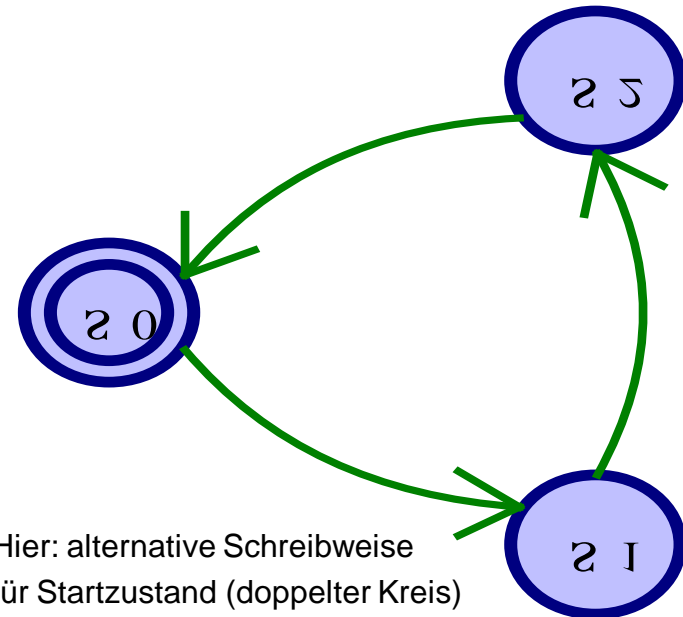
- Drei Blöcke:

- Zustandsübergangslogik (*next state logic*)
- Zustandsregister (*state register*)
- Ausgangslogik (*output logic*)



Beispiel-FSM: Dritteln der Taktfrequenz

- **Eingabe:**
 - Explizit kein Signal
 - Implizit den Schaltungstakt
 - Mit Frequenz f
- **Ausgabe**
 - Signal q_1 mit Frequenz $f/3$



FSM in Verilog



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module divideby3FSM (input  clk,
                    input  reset,
                    output q);
    reg [1:0] state, nextstate;

    parameter S0 = 2'b00;           // Kodierung der Zustände
    parameter S1 = 2'b01;
    parameter S2 = 2'b10;

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

    always @ (*) // Zustandsübergangslogik
        case (state)
            S0:      nextstate = S1;
            S1:      nextstate = S2;
            S2:      nextstate = S0;
            default: nextstate = S0;
        endcase

    assign q = (state == S0);        // Ausgangslogik
endmodule
```

2:1 Multiplexer:

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

Instanz mit 8-bit Busbreite (verwendet Standardwert):

```
mux2 mux1(d0, d1, s, out);
```

Instanz mit 12-bit Busbreite:

```
mux2 #(12) lowmux(d0, d1, s, out);
```

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

```
mux2 #(.WIDTH(12)) lowmux(d0, d1, s, out);
```

- 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 wird **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

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:

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

Der Modulname sei **sillyfunction**

Verilog

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

Einfacher Testrahmen für Beispiel



```
module testbench1 ();  
    reg  a, b, c;  
    wire 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
```


Selbstprüfender Testrahmen



TECHNISCHE
UNIVERSITÄT
DARMSTADT

```
module testbench2 ();
    reg  a, b, c;
    wire y;

    // Instanz des zu testenden Moduls erzeugen
    sillyfunction dut(a, b, c, y);

    // Eingangswerte anlegen, warten,
    // Ausgang mit erwartetem Wert überprüfen
    initial begin
        a = 0; b = 0; c = 0; #10;
        if (y != 1) $display("000 fehlerhaft.");

        c = 1; #10;
        if (y != 0) $display("001 fehlerhaft.");

        b = 1; c = 0; #10;
        if (y != 0) $display("010 fehlerhaft.");

        c = 1; #10;
        if (y != 0) $display("011 fehlerhaft.");

        a = 1; b = 0; c = 0; #10;
        if (y != 1) $display("100 fehlerhaft.");
```

```
        c = 1; #10;
        if (y != 1) $display("101 fehlerhaft.");

        b = 1; c = 0; #10;
        if (y != 0) $display("110 fehlerhaft.");

        c = 1; #10;
        if (y != 0) $display("111 fehlerhaft.");
    end
endmodule
```

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

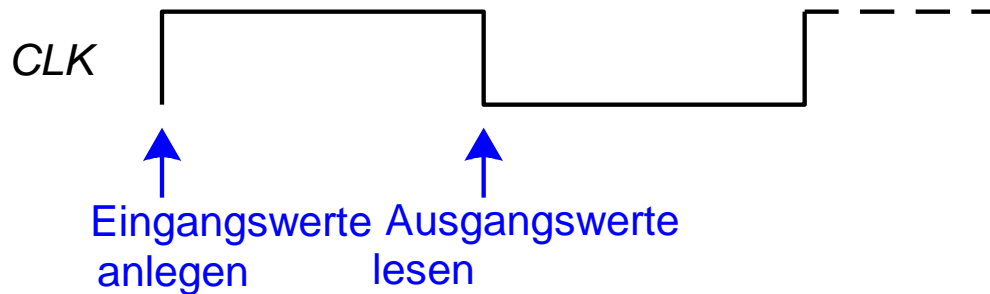
- Dann HDL-Programm für universellen Testrahmen
 1. Erzeuge Takt zum Anlegen von Eingabedaten/Auswerten von Ausgabedaten
 2. Lese Vektordatei in Verilog Array
 3. Lege Eingangsdaten an
 4. Warte auf Ausgabedaten, werte Ausgabedaten aus
 5. Vergleiche aktuelle mit erwarteten Ausgabedaten, melde Fehler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

Selbstprüfender Testrahmen mit Testvektoren



TECHNISCHE
UNIVERSITÄT
DARMSTADT

- 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



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Datei: `example.tv`

000_1

001_0

010_0

011_0

100_1

101_1

110_0

111_0

Aufbau:

Eingangsdaten “_” erwartete Ausgangsdaten

Testrahmen: 1. Erzeuge Takt



```
module testbench3 ();  
    reg          clk, reset;  
    reg          a, b, c, yexpected;  
    wire         y;  
    reg [31:0] vectornum, errors;    // Verwaltungsdaten  
    reg [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  
  
    ...
```

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

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)

    if (~reset) begin          // nur Prüfen, nachdem Schaltung schon initialisiert

        if (y != yexpected) begin // vergleiche aktuelle Ausgabe mit erwartetem Wert

            $display("Fehler: Eingänge = %b", {a, b, c}); // Fehlermeldung
            $display("  Ausgänge = %b (%b erwartet)", y, yexpected);

            errors = errors + 1;          // zähle Fehler
        end
    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  
  
        $display("%d Tests bearbeitet mit %d Fehlern", // Endmeldung ausgeben  
                vectornum, errors);  
  
        $finish; // Simulation anhalten  
    end  
end  
endmodule
```

Hinweis: Zum Vergleichen auf **X** und **Z** müssen die Operatoren

=== und **!==**

benutzt werden



- Vor Testrahmen alle für die Beschreibung von **echter** Hardware relevanten eingeführt
 - **Schaltungssynthese**
- Verilog 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
- In TGD soll dieser Kurzüberblick reichen
 - Bei akutem Bedarf werden noch weitere Konstrukte eingeführt