Die Hardware-Beschreibungssprache

Verilog

Überblick Verilog

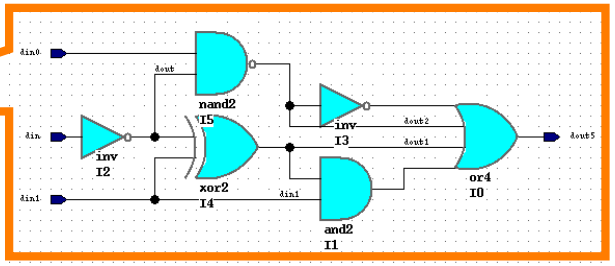
- **Entwickelt 1983/1984** als Werkzeug zur Modellierung, Simulation und Analyse digitaler Schaltungen
- Ursprünglich Eigentum von Cadence
- 1990 gab Cadence die Sprache zur Verwendung durch Dritte frei
- **Standardisierung** im Jahr **1995** durch die IEEE
- Einfach zu lernen
- **Kompakter Code**
- Syntax **ähnelt C**
- Inzwischen auch **Verilog 2001**

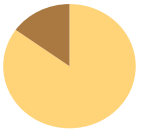
- Stark verbreitet in Nordarmerika und Japan, weniger in Europa
- Stark bei **low-level designs**, hat Schwächen bei system-level designs
- Wird verwendet zur Spezifikation von **Netzlisten** im back-end
- 1993 wurden **85% der ASICs mit Verilog** entworfen (Quelle: EE Times)
- Im Internet unter: www.ovi.org

cadence

IEEE

```
case (sel)
  0: out = in1;
  1: out = in2;
```





Vorsicht !

- ‚The syntax of Verilog is very similar to that of the ‚C' or Java languages although the semantics are quite different. Verilog has many mechanisms for **concurrency** as appropriate for hardware modeling. As a result, 'C' or Java programmers who follow their intuition or syntax matters will be right much of the time, although **not all** of the time!
  (Smith/Franzon: Verilog Styles for Synthesis of Digital Systems, Prentice-Hall, 2000, ISBN 0-201-61870-5)

- Beispiele:

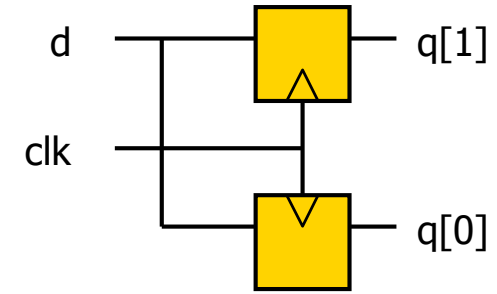
```
...
wire  [N:1]  a, b, s1, s2, out;
assign s1 = a + 1;
assign s2 = b + 1;
assign out = s1+ s2;
...
```

```
...
wire  [N:1]  a, b, s1, s2, out;
assign s1 = a + 1;
assign out = s1+ s2;
assign s2 = b + 1;
...
```

Beide Beschreibungen sind **identisch**!

```
...
integer i;
always @ (posedge clk)
for (i=0; i<N; i=i+1)
  q[i] = d;
...
```

ergibt:



Schleife wirkt nicht temporal, sondern topologisch !

Verilog vs. VHDL

```
module counter (data, clk, clrn, ena, ld, count);

   input [7:0] data;
   input clk, clrn, ena, ld;
   output [7:0] count;

   reg  [7:0] count_tmp;

   always @(posedge clk or posedge clrn)
     begin
        if (!clrn)
            count_tmp = 'b0;
        else if (ld)
            count_tmp = data;
        else if (ena)
            count_tmp = count_tmp + 'b1;
     end

   assign count = count_tmp;

endmodule
```

C-style

```
library ieee;
use ieee.std_logic_1164.all;

entity counter is
    port
    (  data : in integer range 0 to 255;
       clk, clrn, ena, ld : in std_logic;
       count : out integer range 0 to 255 );
end counter;

architecture counter_arch of counter is
    signal countsig : integer range 0 to 255;
begin
    process (clk, clrn)
        begin
            if clrn = '0' then
                countsig <= 0;
            elsif (clk'event and clk = '1') then
                if ld = '1' then
                    countsig <= data;
                    else
                        if ena = '1' then
                            countsig <= countsig + 1;
                            else
                                countsig <= countsig;
                        end if;
                end if;
            end if;
    end process;
    count <= countsig;
end counter_arch;
```

Pascal-style

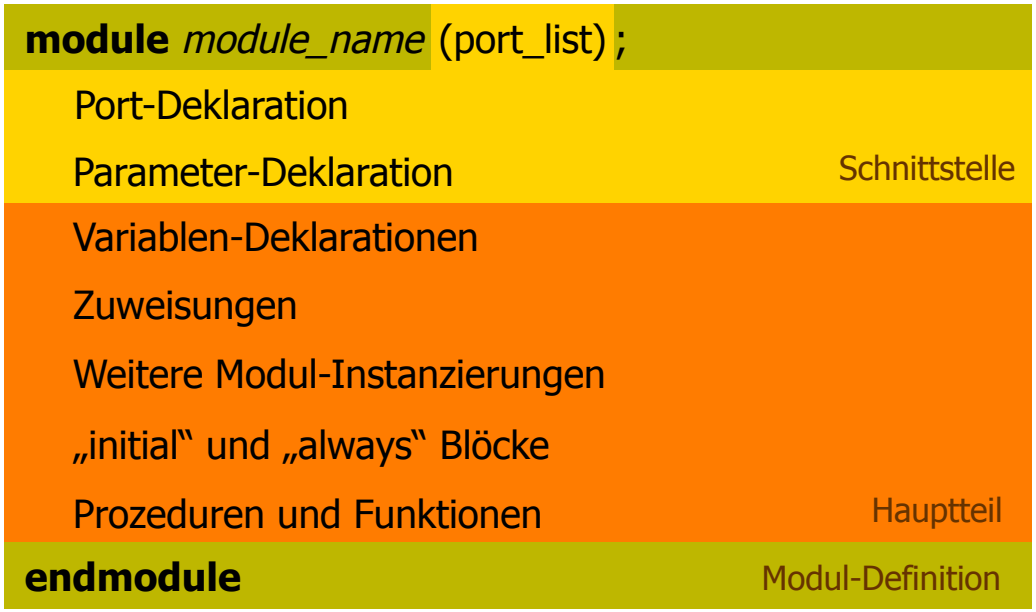
Synthetisierbarkeit und ‚Coding Style'

- Viele Befehle im Sprachumfang sind NICHT synthetisierbar, d.h. die Design-Werkzeuge können daraus keine ‚Gatter'-Netzlisten erstellen (z.B. $print, $stop, $readmemx,… aber auch initial oder UDPs)
- Simulationen sind damit möglich

- Zur besseren Lesbarkeit, Wiederverwertbarkeit und zum Vermeiden von Fehlern sollte man vorgegebene ‚Coding Styles' beachten. Hiervon gibt es verschiedene (z.B. in Firmen vorgegeben).
- Sie legen z.B. fest
  - Formatierung
  - Variablen-Namen
  - Implementierung State-Machines
  - …
  - Buchtitel: Verilog **Styles** for Synthesis of Digital Systems
- Für manche Leute sind ‚Verletzungen' des ‚Coding Style' fast wie Syntax Fehler…

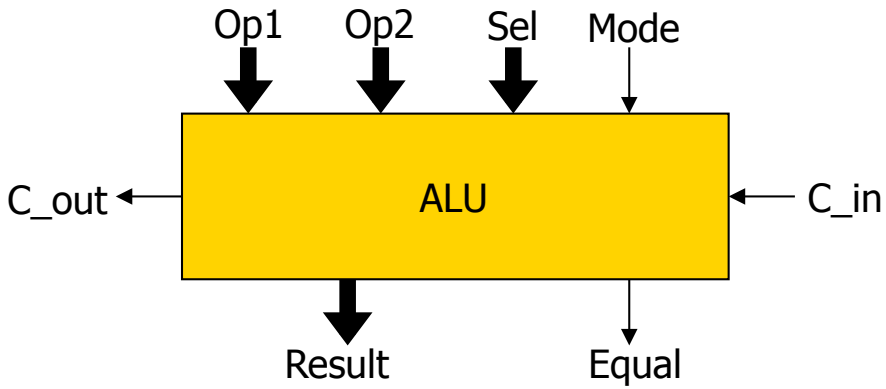
- NB: In Verilog2001 gibt es neue sprachliche Möglichkeiten. Diese müssen aber natürlich von den Tools unterstützt werden. (Siehe Link auf LS Seite.)
- Beispiele:
  - kombinierte Deklarationen `output reg q;`
  - Port Definitionen wie in C: `module xx(input[3:0] wire a, output reg sum)`

Definition eines Moduls



- Module stellen das wichtigste strukturelle Element in Verilog dar
- Beinhalten das eigentliche Design
- Jedes Design besteht aus mindestens einem Modul (sog. top-level Modul)
- Können verschachtelt werden, definieren so die Design-Hierarchie
- Können also strukturell aufgebaut sein durch Instanzen von Gattern, Standard-Zellen oder weiteren Modulen
- Ebenso möglich: Inhalt eines Moduls wird algorithmisch beschrieben durch sog. „always“-Blöcke

Die Modul-Schnittstelle



Port-Deklaration:
Genaue Angabe der Richtung des Signalflusses durch den Port und seine Breite

```
module ALU (Result, Equal, C_out,
            C_in, Op1, Op2, Sel, Mode);

 input  [3:0]  Op1, Op2, Sel;
 input         C_in, Mode;
 output [3:0]  Result;
 output        C_out, Equal;
  .
  .
  .

endmodule
```

Port-Liste:
Angabe der Portnamen

Port Deklaration in Verilog2001

- In Verilog2001 können die ports direkt in der **port liste** deklariert werden:

```
module XXX (
 input        CLK,     // 40 MHz
 input  [7:0] CMD,     // 8 Bit breiter Eingang
 output       ERR,     // Ausgangs (Bit)
 inout  [3:0] PRI      // 4 bit breite bid. Leitung
)
   .
   .
   .

endmodule
```

Signale

Verilog Schlüsselworte für externe Signale: **input**, **output** oder **inout**

```
module ALU (Result, Equal, C_inout,
            C_in, Op1, Op2, Sel, Mode);

 input  [3:0]  Op1, Op2, Sel;
 input         C_in, Mode;
 output [3:0]  Result;
 inout         C_inout, Equal;

 wire   [7:0]  IntSig;
 reg    [3:0]  Result;
    .
    .
    .

endmodule
```

Vektoren: Angabe der Indizes in eckigen Klammern, z.B. **[7:0]**, zwischen Schlüsselwort und Signalnamen

Verilog Schlüsselwort für Signale innerhalb eines Moduls: **wire**

Externe Signale vom Typ „output" und interne Signale können mit „**reg**" als Register gekennzeichnet werden

- Signale in Verilog können nur **vier Werte** annehmen: **0**, **1**, **X** (x) oder **Z** (z)
- **X** steht für „**Unknown**", d.h. das Signal kann 0, 1 oder Z sein oder ändert sich gerade
- **Z** steht für „**High Impedance**", d.h. die Signalquelle ist abgetrennt

- Output-Ports vom Typ „reg" werden in **sequentiellen** Schaltkreisen verwendet

  Damit wird sichergestellt, dass der Wert des Signals bis zum nächsten **Taktzyklus** gehalten wird, statt in den Zustand „High Impedance" zu wechseln

Parameter

Parameter-Konstanten	**parameter** Name **=** Konstante**;**

- Parameter dienen zur Definition von elementaren Konstanten, wie z.B. Verzögerungen (delays), Busbreiten und Schleifendurchläufe
- Werden zwei oder mehr Konstanten deklariert, müssen die einzelnen Name-Werte-Paare mit Kommas getrennt werden

```
parameter BusWidth = 8;
reg [BusWidth-1:0] DataBus;
```

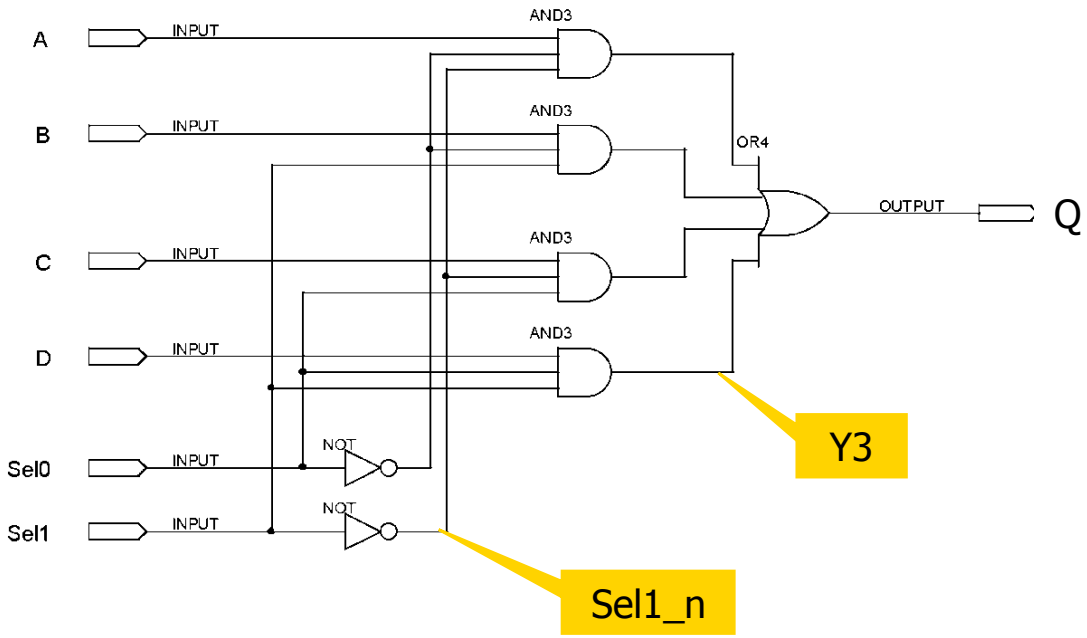
```
parameter Iterations = 5;
for (k=0; k<Iterations; k=k+1);
```

```
parameter PropDel = 3;
assign #PropDel A = B;
```

Strukturelle Beschreibung

Direkte Beschreibung der Schaltung

```
module MUX4(A,B,C,D,Sel0,Sel1,Q);

 input  A,B,C,D,Sel0,Sel1;
 output Q;

 wire   Sel0_n, Sel1_n;
 wire   Y0, Y1, Y2, Y3;

 not (Sel0_n, Sel0);
 not (Sel1_n, Sel1);
 and (Y0, A, Sel0_n, Sel1_n);
 and (Y1, B, Sel0_n, Sel1  );
 and (Y2, C, Sel0,   Sel1_n);
 and (Y3, D, Sel0,   Sel1  );
 or  (Q, Y0, Y1, Y2, Y3);

endmodule
```



- Beschreibung eines Moduls durch **Primitive** (and, or,…) oder weitere **Module** und **Verbindungen**
- **Die Reihenfolge der Instanzen ist irrelevant** (Konzept der Nebenläufigkeit)
- Die ‚**Primitive**' (‚Gatter-Ebene') werden nicht weiter aufgelöst
- Reihenfolge der Argumente hier: (Ausgang, Eingang1, Eingang2,…)

Strukturelle Beschreibung: ‚Primitives`

```
module MUX2(A1,A0,Sel,Q);

 input   A1,A0,Sel;
 output  Q;

 wire    Sel_n;
 wire    Y0, Y1;

 not #3  U1 (Sel_n, Sel);
 and (weak0,strong1) (Y1, A1, Sel);
 and #(2,3) (Y0, A0, Sel_n);
 or  (Q, Y0, Y1);

endmodule
```

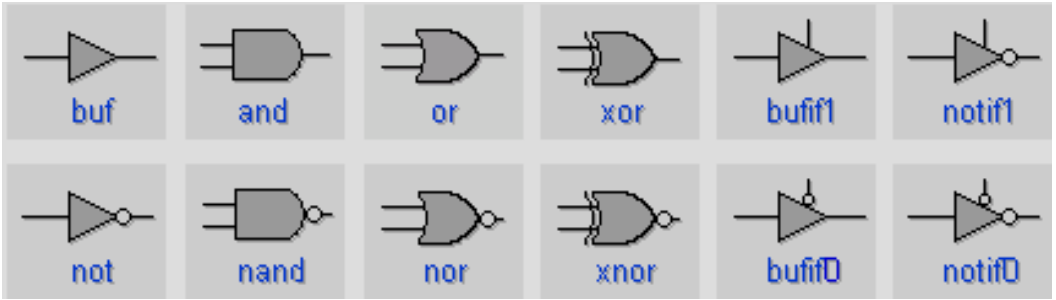
Instanzname

Treiberstärke

Verzögerung

Einfache Gatter etc:

- **single-input gates:** buf, not
- **multiple-input gates:** and, nand, nor, or, xnor, xor
- **tristate gates**: bufif0, bufif1, notif0, notif1
- und einige mehr…



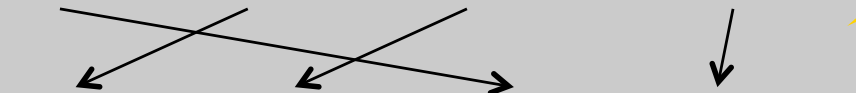
Einer Instanz einer Gatter-Primitive können **zusätzliche Informationen** beigefügt werden:

- Instanz **Name** und eine **Bereichsangabe** für Felder (arrays) von Instanzen (`wire[7:0] y,x; buf AB[7:0] (y,x);`)
- **Treiber-Stärken**, die im Fall von Signal-Konflikten entscheiden ‚wer gewinnt`:
  Es gibt 7 Stärken (supply, strong, weak, highz…), default ist strong.
- **Signal-Verzögerung** (auch getrennt für steigende und fallende Flanke). Die Einheit der Verzögerung (float) wird durch den Befehl ´**timescale <reference_time_unit>/<time_precision>** festgelegt, z.B. `timescale 10ns/1ns
- **Achtung**: Treiber-Stärke und Signal-Verzögerung sind **nicht-synthetisierbare** Konstrukte

Argumente mit Namen

- Um Verwechslungen in der Reihenfolge der Argumente zu vermeiden, **sollten** diese mit NAMEN aufgerufen werden.
- In der Argumentliste werden sie dazu mit **.NAME_IN_DEKLARATION(Netz)** aufgerufen
- Die Reihenfolge der Deklaration im Modulkopf ist **dann** egal

```
module MUX2TO1 (A, B, Sel0, Out);

 input  A, B, Sel0;
 output Out;
 …

endmodule

wire w1, w2, w3, w4

MUX2TO1 I_MUX1 (w1, w2, w3, w4)

MUX2TO1 I_MUX2 (.A(w1), .B(w2), .Sel0(w3), .Out(w4))


MUX2TO1 I_MUX3 (.B(w2), .Sel0(w3), .A(w1), .Out(w4))
```

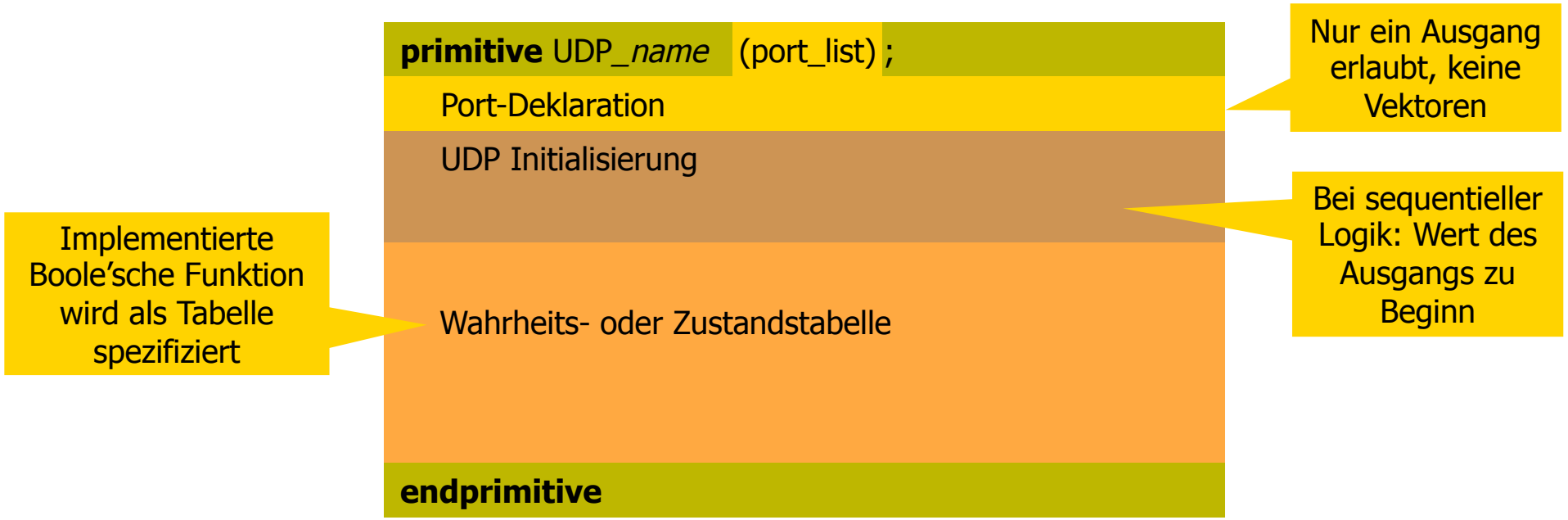


Reihenfolge ist **egal** !

User Defined Primitives (UDP)

Nicht wichtig!

- Erweiterung des Konzepts der Gatter-Primitive durch **User Defined Primitives** (UDP)
- Sowohl **kombinatorische** als auch **sequentielle** Logik kann spezifiziert werden
- Für jede mögliche Belegung der Eingangssignale wird der Wert des Ausgangs in einer Tabelle spezifiziert
- Ist eine Belegung nicht aufgelistet wird der Ausgang auf 'x' gelegt



- **Achtung**: UDPs sind **nicht synthetisierbar**!

User Defined Primitives (Beispiele)

Nicht wichtig!

2 zu 1 Multiplexer

```
primitive MUX2 (Q, Sel, A, B);

 input  A, B, Sel;
 output Q;

 table
 // Sel  A  B  :  Q
     0   0  ?  :  0;
     0   1  ?  :  1;
     1   ?  0  :  0;
     1   ?  1  :  1;
     x   ?  ?  :  x;
 endtable

endprimitive
```

Reihenfolge der Eingänge wie in der Port-**Liste**

? steht für 0, 1 oder x

Toggle-Flipflop mit 'clear'

```
primitive TFF (Q, Clk, Clr);

 input  Clk, Clr;
 output Q;
 reg    Q;

 initial
  Q = 0;

 table
 // Clk Clr : Q :  Q+
     ?   1  : ? :  0;
     r   0  : 0 :  1;
     r   0  : 1 :  0;
     f   0  : ? :  -;
 endtable

endprimitive
```

Gegenwärtiger Wert des Ausgangs

Nächster Wert des Ausgangs

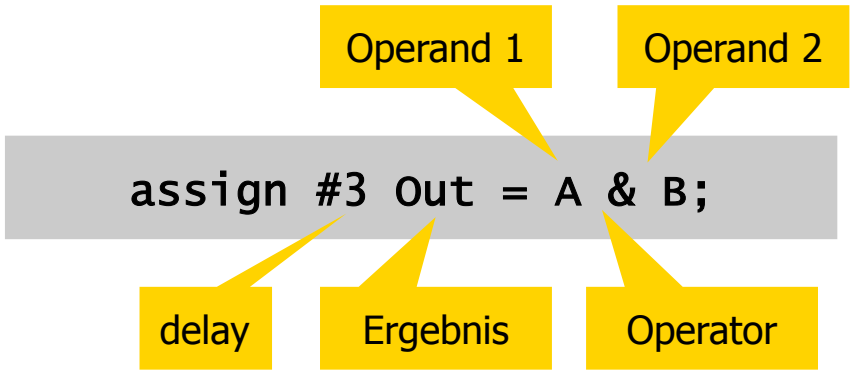
r = steigende Taktflanke

f = fallende Taktflanke

Minus-Zeichen: Keine Änderung

Datenflussbasierte Beschreibung

- Einem Signal wird ein (neuer) Wert zugewiesen (assignment)
- Dieser Wert ergibt sich aus der Transformation eines oder mehrerer Operanden durch einen Operator




```
module MUX (A,B,C,D,S0,S1,Q);

 input  A,B,C,D,S0,S1;
 output Q;

 assign Q = (~S0 & ~S1 & A) |
            (~S0 &  S1 & B) |
            ( S0 & ~S1 & C) |
            ( S0 &  S1 & D);

endmodule
```

- Operanden vom Typ **wire** oder **reg** , **Vektoren**, **Konstanten** und **Funktionsaufrufe**
- Ergebnis immer vom Typ **wire**
- Optional: Modellierung der Verzögerung (delay) **nicht synthetisierbar**!

Operanden (Beispiele)

SEHR wichtig!

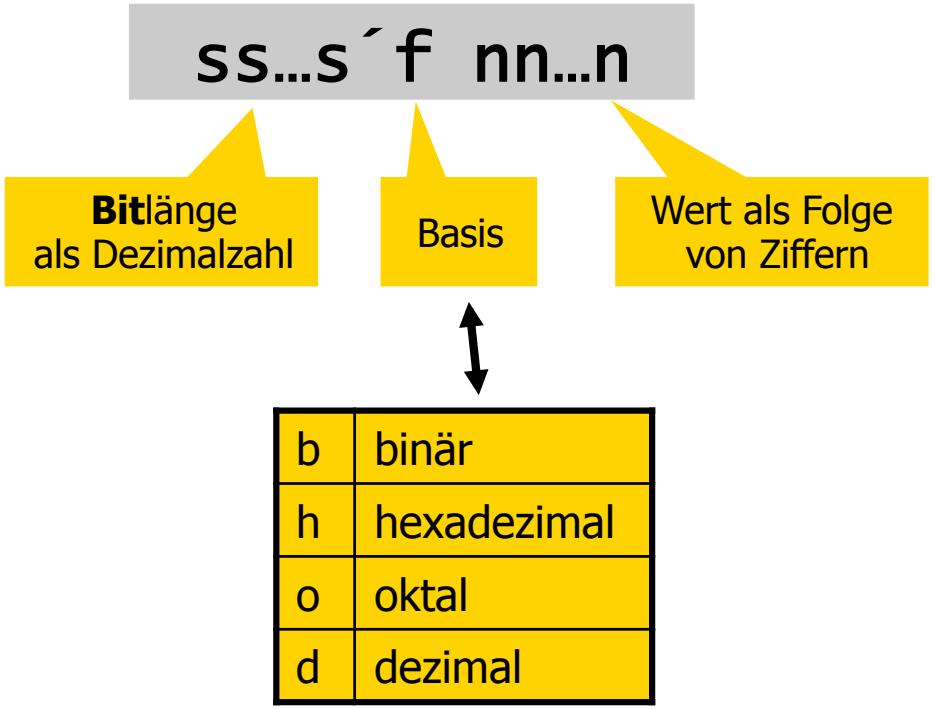
```
wire        signal1;
wire [3:0]  bus1;
real        temperatur;
reg  [7:0]  data;
```

Register vom Typ real (auch integer oder time)

signal1	Einfache Referenz auf ein Netz
temperatur	Einfache Referenz auf ein Register vom Typ real
data[7:4]	„Part-select" Operand, Teil des Vektors data (hier 4 Bit breit)
data[0]	„Bit-select" Operand, **ein** Bit des Vektors data
{ signal1, bus1[2:1], bus1[3] }	**Konkatenation** (hier 4 Bit breit)
{4 {bus1[3:0]}}	**Replikation** (hier 16 Bit breit)
114	Konstante ohne Basis, Verilog nimmt Dezimalzahl an
1.2e7	Konstante vom Typ real in wissenschaftl. Notation
4'b0101	Konstante der Breite 4 Bit
8'hFF	Konstante in Hexadezimal, hier 8 Bit breit
CToF(temperatur)	Funktionsaufruf

Konstante Zahlen

- Konstante Zahlen können **dezimal**, **hexadezimal**, **oktal** oder **binär** angegeben werden.
- Zwei Formen der Angabe sind möglich:
  1. Normale dezimale Schreibweise durch Verwendung der Ziffern 0 bis 9. Die Größe der Zahl ergibt sich aus der max. Größe der anderen Operanden eines Ausdrucks.
  2. Spezielle Schreibweise (Syntax) der Form ss…s´nn…n
- **Default** ist **dezimal**
- Höherwertige Stellen werden mit **Nullen** gefüllt
- Optionale Unterstreichstriche ‚_' zur besseren Lesbarkeit




```
wire[11:0] w;

w = 123;     // dezimal
w = 12´hff; // w = 000011111111
w = 12´bx;  // w = 00000000000x
w = 12´b 0x0x_1101_0010
…
```

Optional zur Verbesserung der Lesbarkeit

Operatoren

Arithmetische Operatoren	
a + b	a plus b
a - b	a minus b
a * b	a mal b
a / b	a geteilt durch b
a % b	a modulo b

- Division bei Integer liefert Integer, Nachkomma-stellen werden abgeschnitten
- Modulo nur bei Integer erlaubt
- x oder z in einem Operanden führt zu x als Ergebnis

Relationale Operatoren	
a < b	a kleiner b
a > b	a größer b
a <= b	a kleiner od. gleich b
a >= b	a größer od. gleich b

- **x oder z in einem Operanden führt zu x als Ergebnis**
- Ist ein Operand kleiner als der andere, wird er von links mit Nullen gefüllt

Gleichheits-Operatoren	
a === b	a gleich b, einschl. x, z
a !== b	a ungleich b, einschl. x, z
a == b	a gleich b, kann x liefern
a != b	a ungleich b, kann x liefern

- Alles wir EXAKT verglichen (auch x und z)
- 1xz0 === 1xz0 liefert also 1
- x oder z im Operanden liefern immer x
- 1xz0 == 1xz0 liefert also x (!)

Operatoren (Forts.)

Bitweise Logik-Operatoren	
&	bitweises UND
|	bitweises ODER
^	bitweises XOR
~^	bitweises XNOR
~	bitweises NOT

- Bitweise logische Verknüpfung
- Ist ein Operand kürzer als der andere, wird er mit **Nullen** gefüllt
- Ergebnis hat immer dieselbe Größe wie der größte Operand

Logische Operatoren	
&&	logisches UND
||	logisches ODER
!	logisches NICHT

- Ist ein Operand Null, wird FALSCH angenommen
- Ist ein Operand ungleich Null, wird WAHR angenommen
- Liefern immer Ergebnis der Länge eins (0, 1 oder x)

Shift-Operatoren	
a << b	Verschiebe Vektor a um b Positionen nach links
a >> b	Verschiebe Vektor a um b Positionen nach rechts

- Der rechte Operand wird immer als Zahl vom Typ unsigned integer behandelt
- Leere Stellen werden mit Nullen aufgefüllt
- x oder z im rechten Operanden führt zu x

Rangfolge der Operatoren

Vorzeichen

Stärkste Bindung

Schwächste Bindung

Unäre Operatoren	**! & ~& | ~| ^ ~^ + -**
Rechenoperatoren	*** / % + - (*/% vor +-)**
Shift-Operatoren	**<< >>**
Vergleichsoperatoren	**< <= > >= == != === !==**
Logikoperatoren (bitweise)	**& ~& ^ ~^ | ~|**
Boolesche Operatoren	**&& ||**
Bedingungsoperator	**? :**

Der Bedingungsoperator

- Wie in der Hochsprache C gibt es eine Kurzschreibweise für folgendes Konstrukt:

```
if (Ausdruck) Register = Zuweisung_wahr;
        else Register = Zuweisung_falsch;
```

Beispiel:

```
if (Sel) Q = A;
   else Q = B;
```

- Dieser Bedingungsoperator (conditional operator) hat die Form:

```
Signal = Ausdruck ? Zuweisung_wahr : Zuweisung_falsch;
```

Beispiel:

```
Q = Sel ? A : B;
```

- Der Bedingungsoperator kann geschachtelt werden:

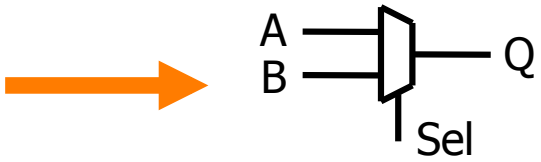
Beispiel:

```
Q = Sel1 ? Sel2 ? A : B : C;
```

- Der Bedingungsoperator kann in einer **verhaltensbasierten** Beschreibung verwendet werden **und** in einer **datenflussbasierten** (assign), d.h. einer strukturellen Beschreibung auf Gatterebene.
- Im Gegensatz dazu kann „if-then-else" **nur** in einer **verhaltensbasierten** Beschreibung (also innerhalb eines „always"-Blockes) verwendet werden.

- Bei der Synthese wird immer ein Multiplexer (MUX) oder ein Tristate-Buffer mit ‚enable' erzeugt:

```
Q = (Sel) ? A : B;
```




```
Q = (enb) ? 1´bz : A;
```

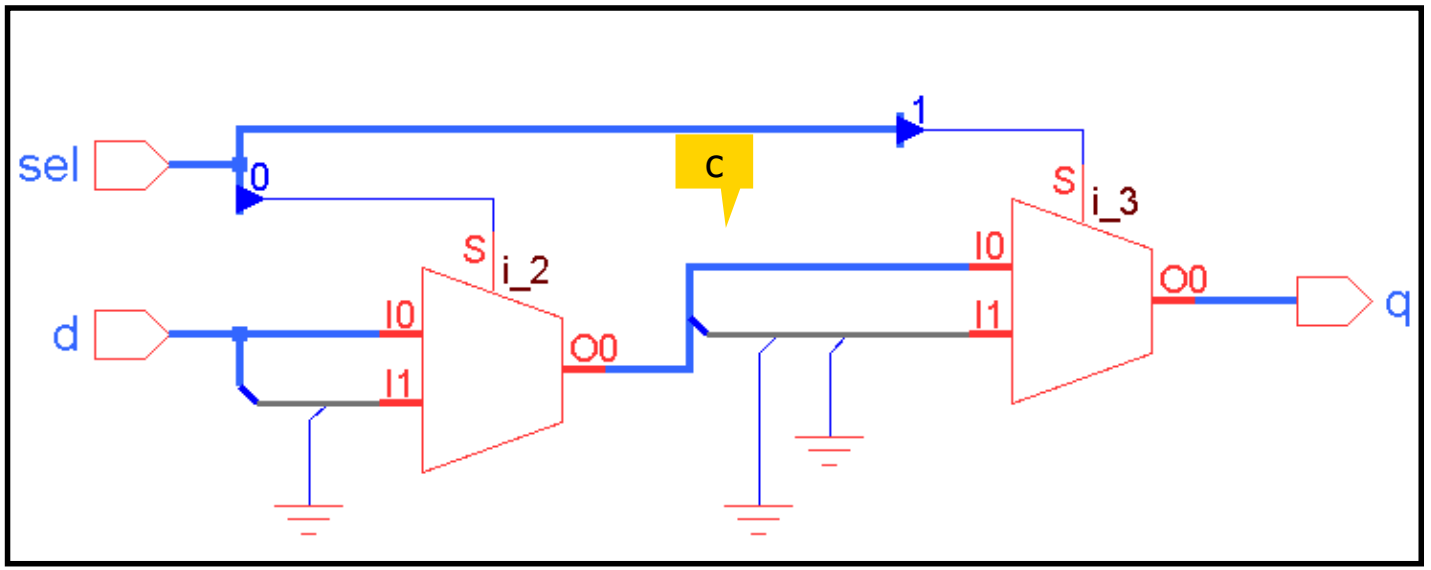
Beispiel1: Barrel Shifter

```
module barrelshift4 (d, q, sel);

  input [3:0] d;
  output[3:0] q;
  input [1:0] sel;

  wire  [3:0] c;
  assign c = sel[0] ? {d[2:0], 1'b0} : d;
  assign q = sel[1] ? {c[1:0], 2'b0} : c;
// assign q = d << sel;  // Alternative!

endmodule
```

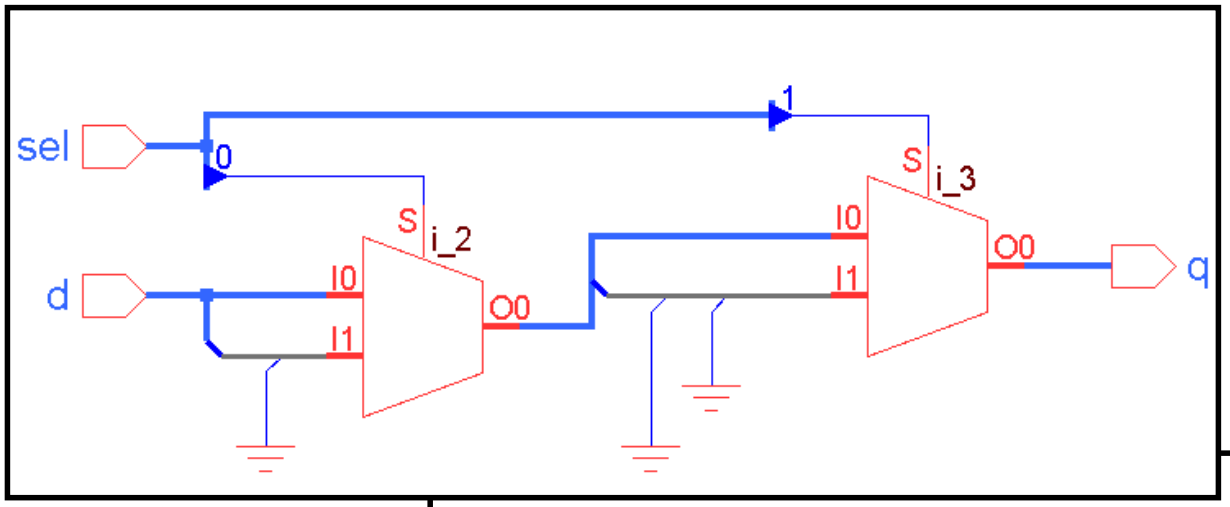
Funktion:

```
d =          abcd
q (sel=0): abcd
q (sel=1): bcd0
q (sel=2): cd00
q (sel=3): d000
```

Synthese

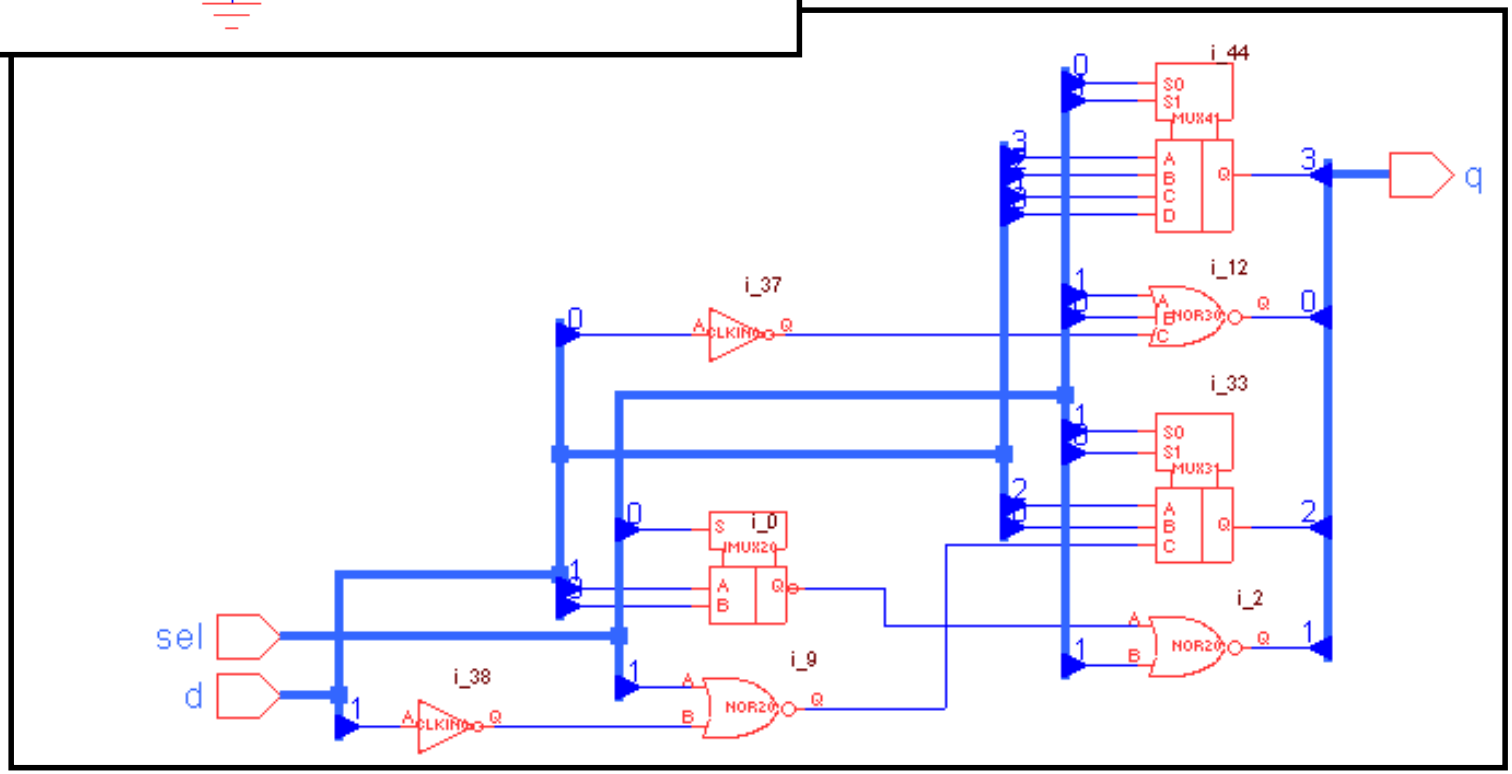


Beispiel1: Barrel Shifter: Technology Mapping



Z.B.:
Q[0] = D[0] & !sel[0] & !sel[1]
= !(!D[0] | sel[0] | sel[1])

Technology Mapping

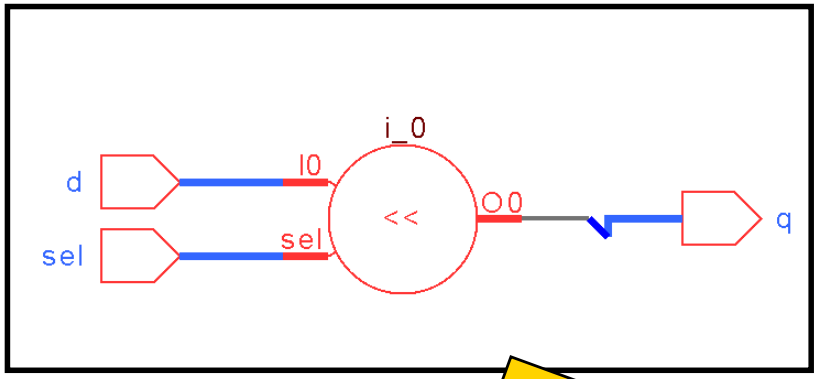


Beispiel2: Rotator

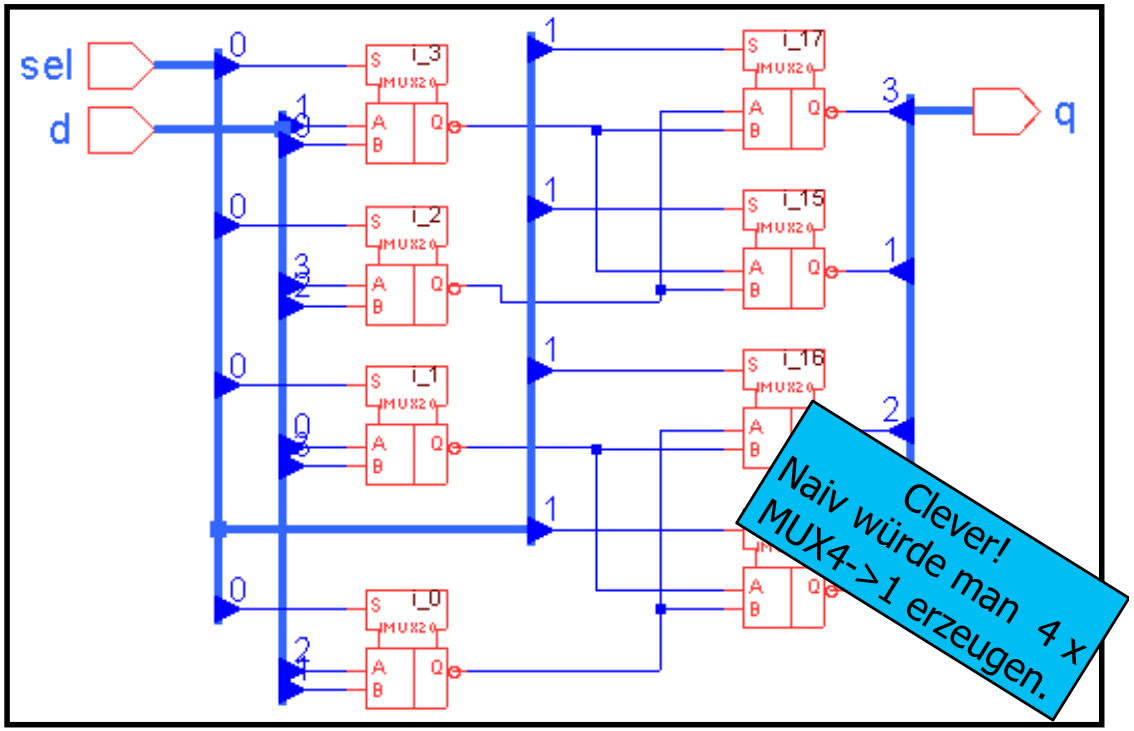
```
module rotate (input [3:0] d, output[3:0] q, input [1:0] sel);

 wire   [3:0] c;
 assign {q[3:0],c[3:0]} = {2{d[3:0]}} << sel;
         // {2{d[3:0]}} = abcdabcd

endmodule
```

```
Funktion:
d =           abcd
q (sel=0): abcd
q (sel=1): bcda
q (sel=2): cdab
q (sel=3): dabc
```

Synthese



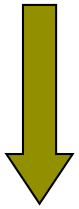
Technology Mapping



Verhaltensbasierte Beschreibung

„initial"- und „always"-Blöcke

- Alle **verhaltensbasierten** Beschreibungen müssen in sog. „initial"- oder „always"-Blöcken eingebettet sein
- Jedes Modul kann **beliebig viele** Blöcke beinhalten
- Die Blöcke werden innerhalb eines Moduls **nebenläufig** (gleichzeitig) ausgeführt – **!!! WICHTIG !!!**
- Blöcke können **nicht** verschachtelt werden
- Anweisungen innerhalb eines Blocks werden mit „**begin**" und „**end**" gruppiert
- Die Anweisungen zwischen „begin" und „end" werden (meist) **sequentiell** ausgeführt



„initial"-Blöcke

- Werden nur zu Beginn (Zeitpunkt 0) ausgeführt
- Werden hauptsächlich benutzt, um Register zu initialisieren
- **Nicht synthetisierbar**

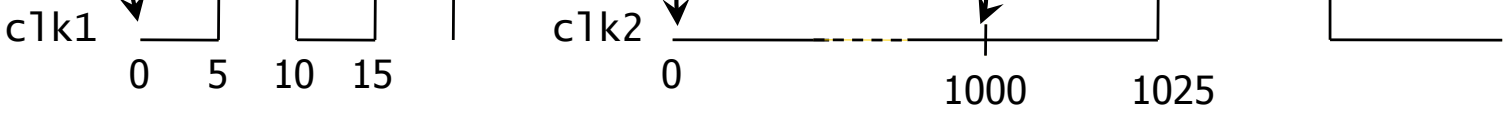
„always"-Blöcke

- Starten ebenfalls zum Zeitpunkt 0, werden jedoch immer wieder ausgeführt
- Sie dienen zur Modellierung des Verhaltens
- **Synthetisierbar**

```
module test
reg clk1, clk2;

initial begin
  clk1 = 0;
end

always begin
  #5 clk1 = ~clk1;
end

initial begin
  clk2 = 0;
  #1000 forever #25 clk2 = ~clk2;
end
endmodule
```

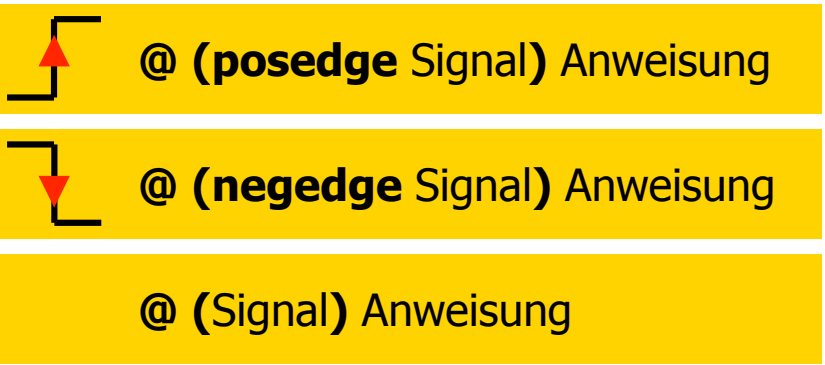
Resultat der Befehle

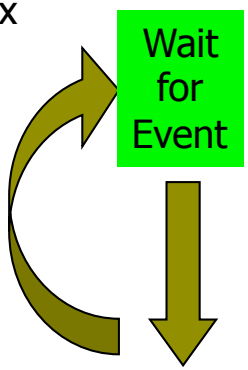
```
module test
reg clk1, clk2;

initial begin
  clk1 = 0;
end

always begin
  #5 clk1 = ~clk1;
end

initial begin
  clk2 = 0;
  #1000 forever #25 clk2 = ~clk2;
end
end
```



Die „sensitivity list“ und „events“

- Neben Spezifikation von „delays“ zur zeitlichen Kontrolle der Ausführung: „Events“
- Events sind z.B. **Änderungen** in einem Netz oder Register
- Events können an denselben Stellen verwendet werden wie delays
- Events werden mit dem **@-Zeichen** spezifiziert
- Wichtigste Events: s. rechts
- Events können auch deklariert werden: **event** xxx
- Wird eine Anweisung durch (mehrere) Signale aktiviert, sprich man von einer „**sensitivity list**“
- Die einzelnen Signale werden durch das Schlüsselwort „**or**“ getrennt
- Sensitivity lists werden hauptsächlich in always-Blöcken verwendet, um deren Ausführung zu triggern



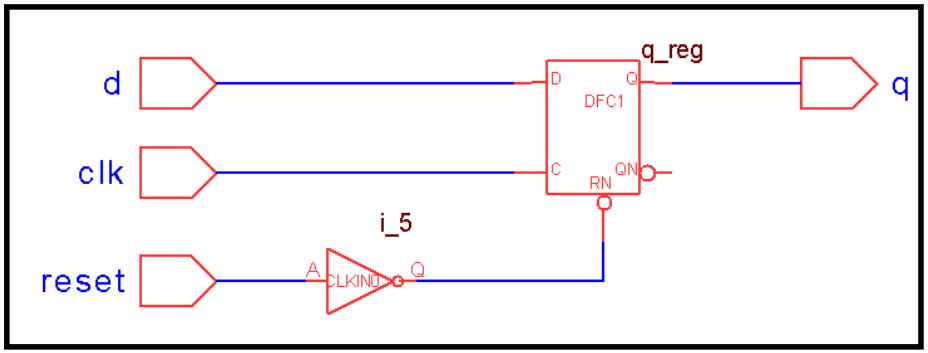



```
always @(posedge res or posedge ck)
  begin
    if (res) Q = 1'b0;
    else Q = D;
  end
```

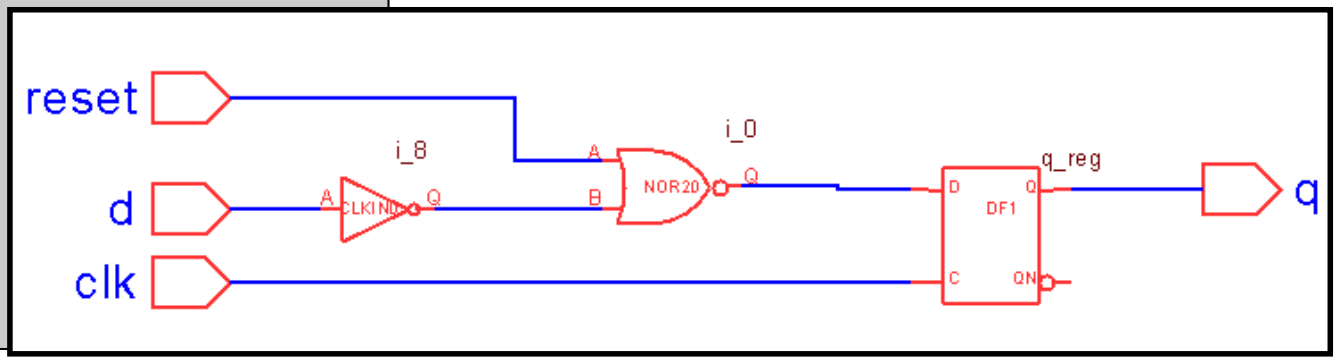
Synchrone und Asynchrone Signale

- Je nach Sensitivity list wirken Signale **synchron** oder **asynchron**:

```
module test_async (clk, reset, d, q);

 input  clk, reset, d;
 output q;
 reg    q;

 always @(posedge clk or posedge reset)
   if (reset) q = 0;
   else q = d;

endmodule
```




```
module test_sync (clk, reset, d, q);

 input  clk, reset, d;
 output q;
 reg    q;

 always @(posedge clk)
  if (reset) q = 0;
  else q = d;

endmodule
```



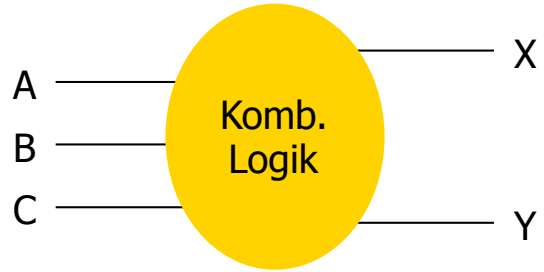
Beispiel: Zähler mit async. reset, load und enable

```
module counter (data, clk, clr, en, ld, count);

 input  [7:0] data;
 input        clk, clr, en, ld;
 output [7:0] count;
 reg    [7:0] count;

 always @(posedge clk or posedge clr) begin
  if (clr)
   count = 'b0;  // is this ok?
  else if (ld)
   count = data;
  else if (en)
   count = count + 1;
 end

endmodule
```

dies macht den Reset **a**synchron!

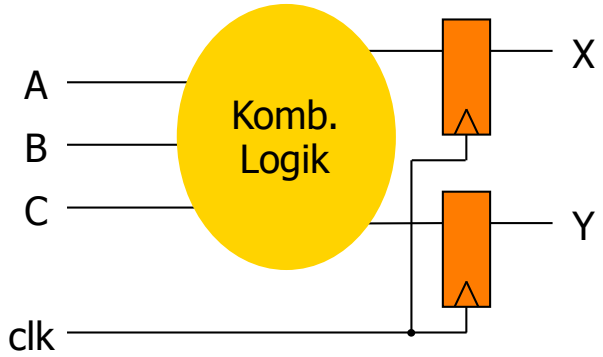
Synthese von Flip-Flops

```
reg X, Y;
always @(A or B or C)
    begin
        X <= ...;
        Y <= ...;
    end
```



- Um Flip-Flops in Verilog zu synthetisieren, genügt die Verwendung von Registern (reg) **nicht**!

```
reg X, Y;
always @(posedge clk)
    begin
        X <= ...;
        Y <= ...;
    end
```



- Anweisungen innerhalb eines **getakteten** always-Blockes führen zu Flip-Flops am Ausgang
- Einzige Bedingung: in der sensitivity list muss die **aktive Taktflanke** spezifiziert werden

Variablen und Verhalten

- Bei **verhaltensbasierten** Beschreibungen ist das Ziel einer Variablenzuweisung immer ein **reg**, kein **wire**
- Das Schlüsselwort „assign" entfällt
- Die Zuweisung muss innerhalb eines „initial" oder „always"-Blockes geschehen
- Modellierung sequentieller Zuweisungen: sog. „**blocking assignments**"
- Modellierung nebenläufiger Zuweisungen: sog. „**non-blocking assignments**"
- Die verschiedenen Typen können in einem Block und mit einer Variable nicht gemischt werden!

blocking assignment

- **Sequentielle** Ausführung
- Zuweisung durch das Symbol „="
- Die Variable ändert ihren Wert zum Zeitpunkt der Ausführung der Zuweisungsoperation innerhalb der Sequenz von Anweisungen in einem Block
- Bei Delays: Spezifizieren die Zeit zwischen den Anweisungen

Achtung!

```
initial
  begin
     regA = #3 0;
     regB = #1 1;
  end
```

Änderung bei t=3

Änderung bei t=**4**

non-blocking assignment

- **Nebenläufige** Ausführung
- Zuweisung durch das Symbol „**<=**"
- Variable ändert ihren Wert gleichzeitig mit allen anderen non-blocking assignments im Block
- Bei Delays: Spezifizieren die Zeit von Beginn des Block zur jeweiligen Anweisung
- **Normalfall** für synthetisierbaren Code!

Standard!

```
initial
  begin
     regA <= #3 0;
     regB <= #1 1;
  end
```

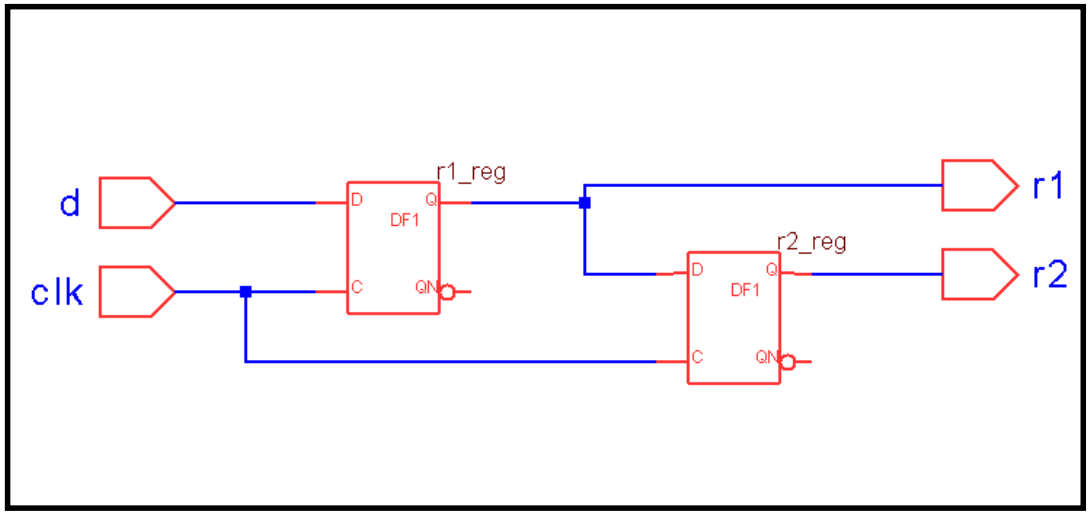
Änderung bei t=3

Änderung bei t=**1**

Beispiel blocking / Non blocking

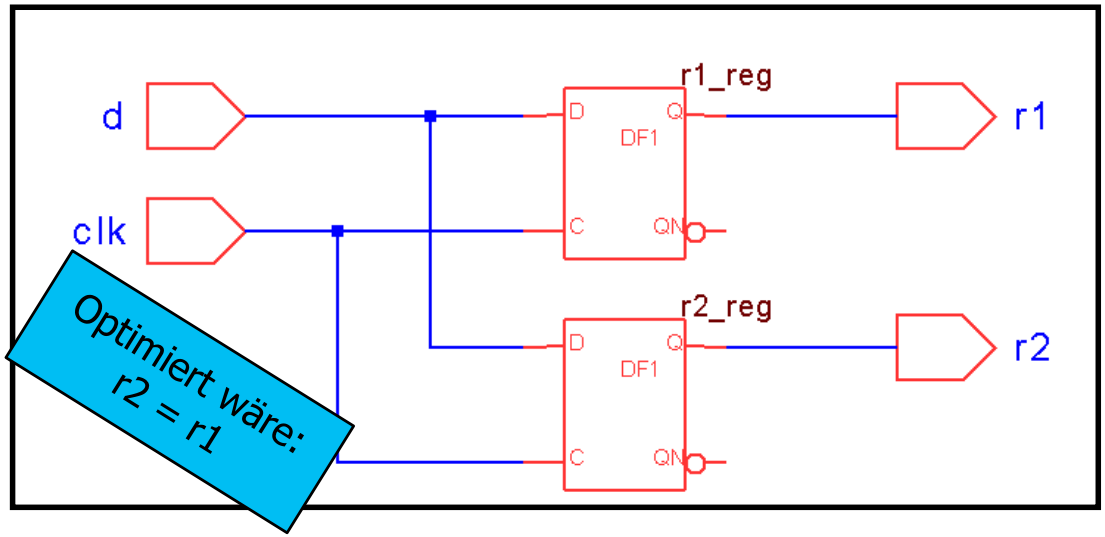
```
module test (output reg r1,
  output reg r2, input d,
  input clk);

always @(posedge clk) begin
  r1 <= d;
  r2 <= r1;
end

endmodule
```

Non Blocking




```
module test (output reg r1,
  output reg r2, input d,
  input clk);

always @(posedge clk) begin
  r1 = d;
  r2 = r1;
end

endmodule
```

Blocking



- ‚Blocking' nicht sehr intuitiv. Besser **vermeiden**!

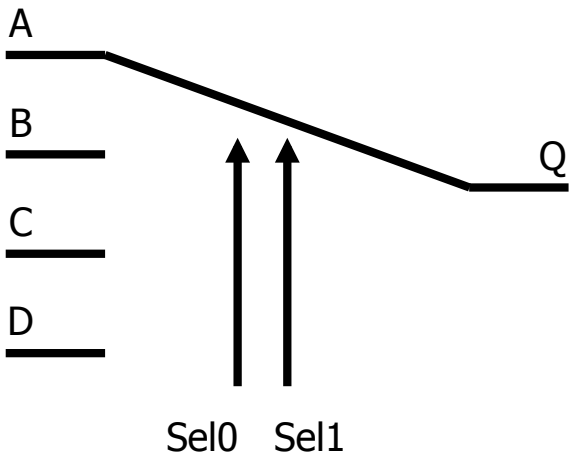
Hochsprachenkonstrukte

Das ‚case' Statement

- Beispiel 4:1 Multiplexer

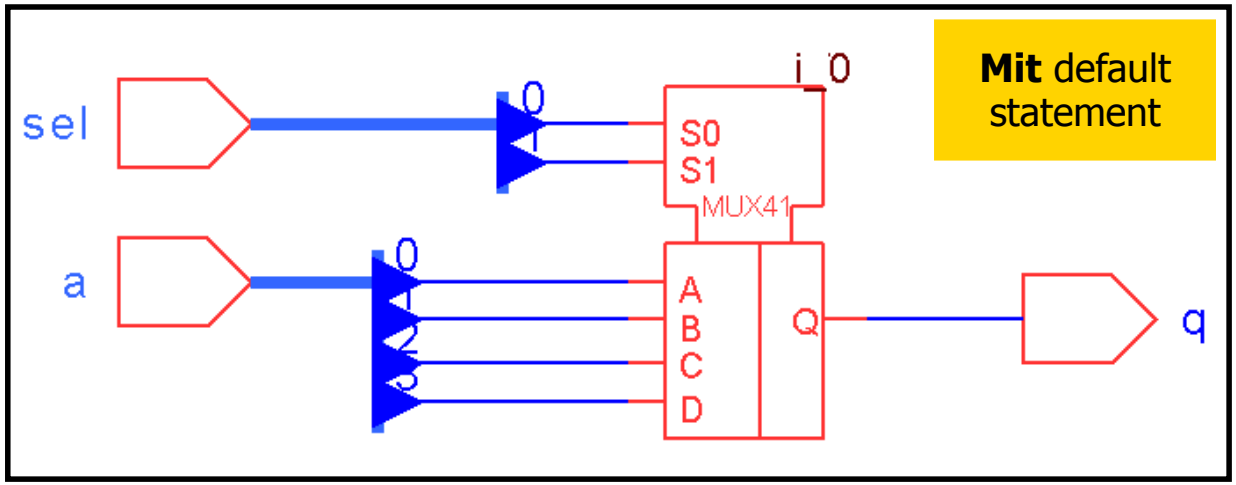
```
module MUX (A,B,C,D,Sel0,Sel1,Q);

 input  A,B,C,D,Sel0,Sel1;
 output Q;

 always @(A or B or C or D or Sel0 or Sel1)
  begin
   case ({Sel1, Sel0})
     2'b00 : Q = A;
     2'b01 : Q = B;
     2'b10 : Q = C;
     2'b11 : Q = D;
   default : Q = 1'bx; // sel could be x
  endcase
 end

endmodule
```

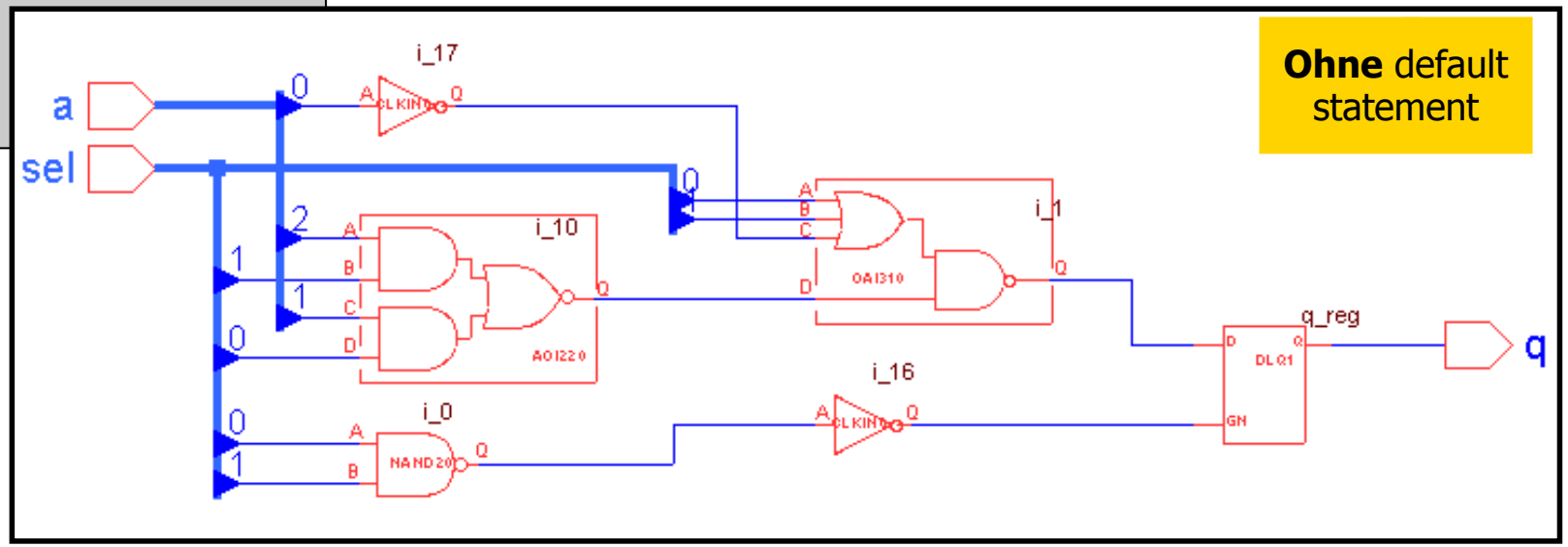
Sensitivity List:
Block wird ausgeführt, wenn sich hier was ändert



Unvollständiges Abfragen der Cases: !!! Vorsicht !!!

```
module test_case(sel,a,q);
input [1:0] sel;
input [3:0] a;
output q;
reg q;
always @(sel or a)
 case (sel)
   0: q <= a[0];
   1: q <= a[1];
   2: q <= a[2];
   default: q <= a[3];
 endcase

endmodule
```





- **Wenn nicht alle cases abgefragt werden, muss Wert gespeichert werden → Latch wird erzeugt**

case / casex / casez

- Es gib **DREI** Varianten
  - **case**: Alle vier Logikwerte (0, 1, x, z) werden berücksichtigt
  - **casez**: z wird als don´t care (‚?') interpretiert
  - **casex**: x und z werden als don´t care interpretiert
- Nach einem Match → **break** (keine weiteren Vergleiche!).
- Der default - Teil ist optional.
- In der Hardware gibt es kein x und meist kein z, daher wird häufig **casex / casez** verwendet.

```
case/casex/casez (Ausdruck)
 Belegung1 : Anweisung1;
 ...
 BelegungN : AnweisungN;
 default   : Anweisung;
endcase
```

1xxx → xx1x →

1000 → zz1z →

```
case ({D0, D1, D2, D3})
 4'b1xxx : Z = 0;
 4'bx1xx : Z = flag;
 4'bxx1x : Z = interrupt;
 4'bxxx1 : Z = 1;
 default : Z = 0;
endcase
```

1xxx → xx1x →

1000 → zz1z →

```
casez ({D0, D1, D2, D3})
 4'b1xxx : Z = 0;
 4'bx1xx : Z = flag;
 4'bxx1x : Z = interrupt;
 4'bxxx1 : Z = 1;
 default : Z = 0;
endcase
```

1xxx → xx1x →

1000 → zz1z →

```
casex ({D0, D1, D2, D3})
 4'b1xxx : Z = 0;
 4'bx1xx : Z = flag;
 4'bxx1x : Z = interrupt;
 4'bxxx1 : Z = 1;
 default : Z = 0;
endcase
```

Synthese von case Statements

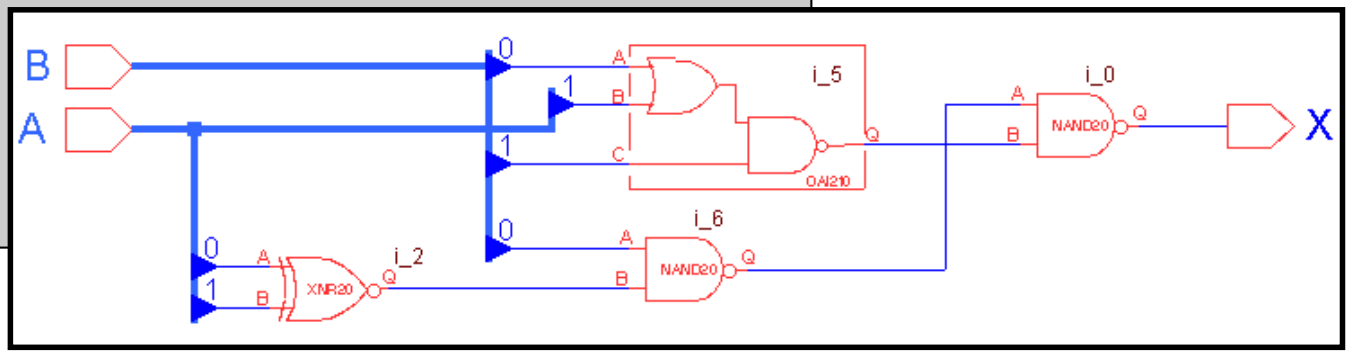
- Case statements werden häufig in Zustandsmaschinen benutzt
- Um eine effiziente Übersetzung zu erreichen, kann man dem Synthese-Tool Direktiven geben
- Diese werden als Kommentar **in der ‚case' Zeile** versteckt.

- case statements sind ‚**full**' wenn alle in der Praxis vorkommenden Eingangsmuster abgefragt werden
  (wenn nicht ‚full' → es werden Latches erzeugt um Zustände zu halten)
  ‚es wird immer ein case ansprechen'
- case statements sind ‚**parallel**', wenn die Kombinationen sich gegenseitig ausschließen.
  (Wenn dies nicht der Fall ist, dann spielt die Reihenfolge der Abfrage eine Rolle und es wird ein Prioritäts-Dekoder erzeugt)
  ‚er werden nie zwei cases ansprechen'

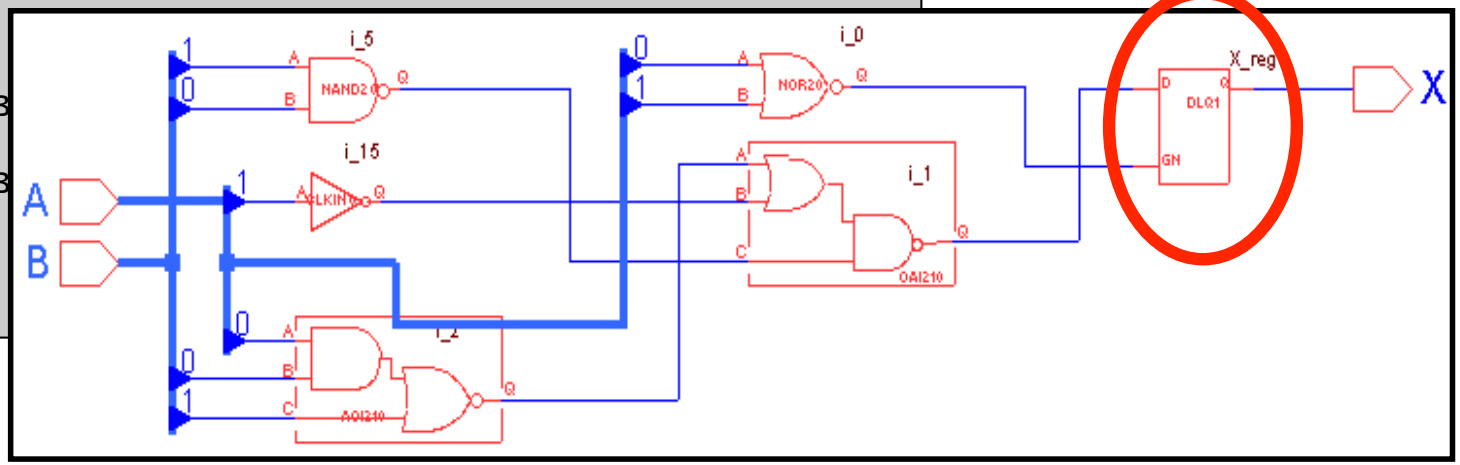
- Weiß man z.B. dass **fehlende** Fälle in einem (**nicht-**full-case) **nie** vorkommen, so kann man die Erzeugung von Latches **vermeiden** mit der Direktive
  // synopsys full_case
- Ähnlich:
  // synopsys parallel_case

Synthese von case Statements

```
module test_fullcase (input [1:0] A, input [1:0] B, output reg X);

always @(*)
  case (A)
    2'b00: X <= B[0];
    2'b01: X <= B[0] & B[1];
    2'b10: X <= B[1];
    2'b11: X <= B[0] | B[1];
  endcase

endmodule
```



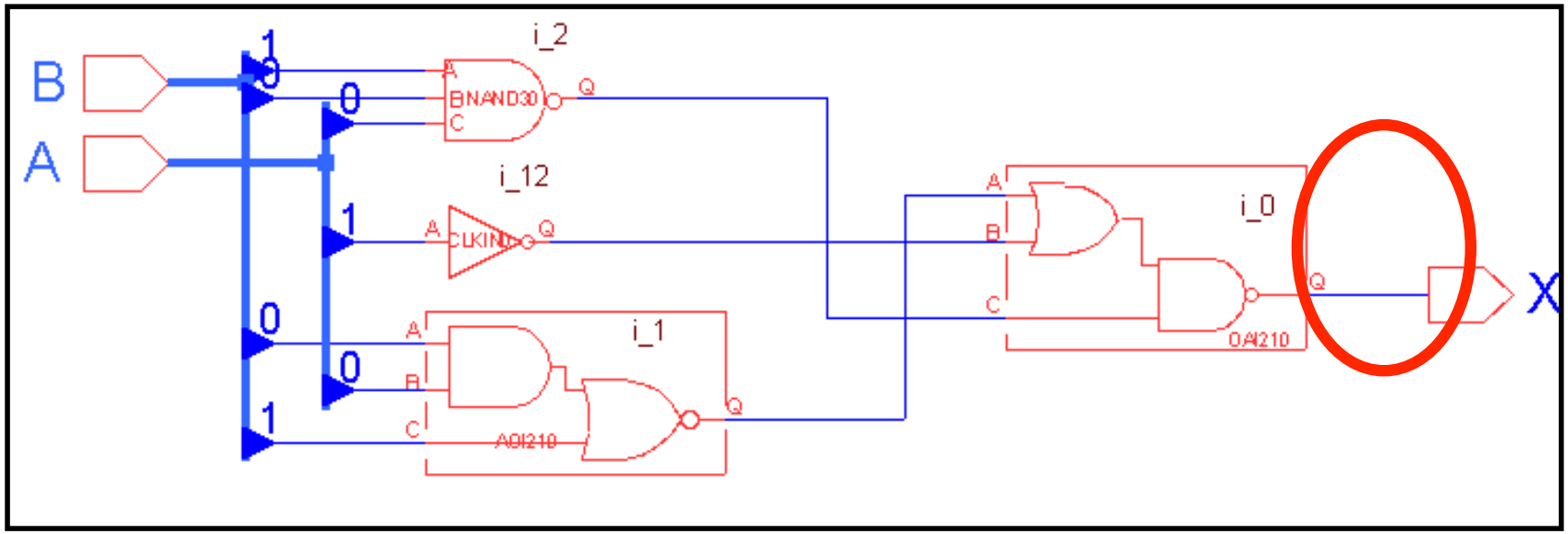

```
module test_nofullcase (input [1:0] A, input [1:0] B, output reg X);

always @(*)
  case (A)
//    2'b00: X <= B[0];
    2'b01: X <= B[0] & B
    2'b10: X <= B[1];
    2'b11: X <= B[0] | B
  endcase

endmodule
```



Synthese von case Statements

- Mit // synopsys full_case

```
module test_nofullcase (input [1:0] A, input [1:0] B, output reg X);

always @(*)
  case (A) // synopsys full_case
//    2'b00: X <= B[0];
    2'b01: X <= B[0] & B[1];
    2'b10: X <= B[1];
    2'b11: X <= B[0] | B[1];
  endcase

endmodule
```



Tasks

```
task Prozedurname;
 input  Signal1;
 output Signal2;
 inout  Signal3;
 begin
  Anweisung1;
  ...
  AnweisungN;
 end
endtask
```

Definition der Argumente, Anzahl beliebig

- **tasks** sind **Unterprogramme ohne Rückgabewert** (Wie Procedures in Pascal)
- Konzept ähnlich den Programmiersprachen: Mehrfachnutzung des Codes (design re-use)
- Aufruf der Prozedur: Prozedurname(Liste der Argumente);
- Argumente werden gemäß ihrer Reihenfolge bei der Definition in der Prozedur interpretiert
- Werden zum Zeitpunkt des Aufrufs ausgewertet

Funktionen

Falls Vektor als Rückgabewert:
Definition der Vektorgröße hier

```
function Funktionsname;
 input Signal;
 begin
  Anweisung1;
  ...
  AnweisungN;
Funktionsname = …;
 end
endfunction
```

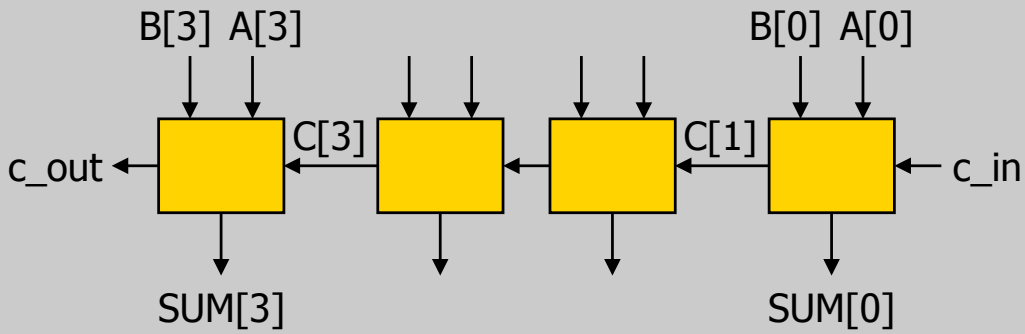
Definition der Eingänge

- Funktionen liefern nur **einen Wert** zurück. Dieser kann auch ein Vektor sein
- Falls nicht anders spezifiziert, ist der Rückgabewert ein 1-Bit Register (reg)
- Keine Definition von Ausgängen (output, inout) erlaubt
- Rückgabe geschieht durch Zuweisung an die durch den Funktionsnamen gegebene lokale Variable
- Kein Spezifikation von delays und keine Ereigniskontrolle in Funktionen erlaubt
- Funktionen werden in **einer** Zeiteinheit der Simulation ausgeführt

Beispiel eines Tasks

- (Umständliche) Definition eines Ripple Addierers mit Breite BITS

```
module Adder4(A, B, C_in, SUM, C_out);

 parameter BITS = 4;

 input  [BITS-1:0] A, B;
 input             C_in;
 output [BITS-1:0] SUM;
 output            C_out;

 wire   [BITS-1:1] C;
 integer           i;

 task FULLADD;
  output c_out, sum;
  input  a, b, c_in;
  {c_out, sum} = a + b + c_in; // Bitbreite ???
 endtask

 FULLADD(C[1], SUM[0], A[0], B[0], C_in);
 for (i=1; i<=BITS-2; i=i+1)
  FULLADD(C[i+1], SUM[i], A[i], B[i], C[i]);
 FULLADD(C_out, SUM[BITS-1], A[BITS-1], B[BITS-1], C[BITS-1]);

endmodule
```



Nachmal: Achtung Latches

Erzeugt Latches

```
always
 begin
  if (Flag)
   Z = A & B;
  else if (!Sel)
   begin
    X = A; Y = B;
   end
  else
   begin
    X = B; Y = A;
   end
 end
```

Keine Aussage über X und Y

Keine Aussage über Z

Keine Aussage über Z

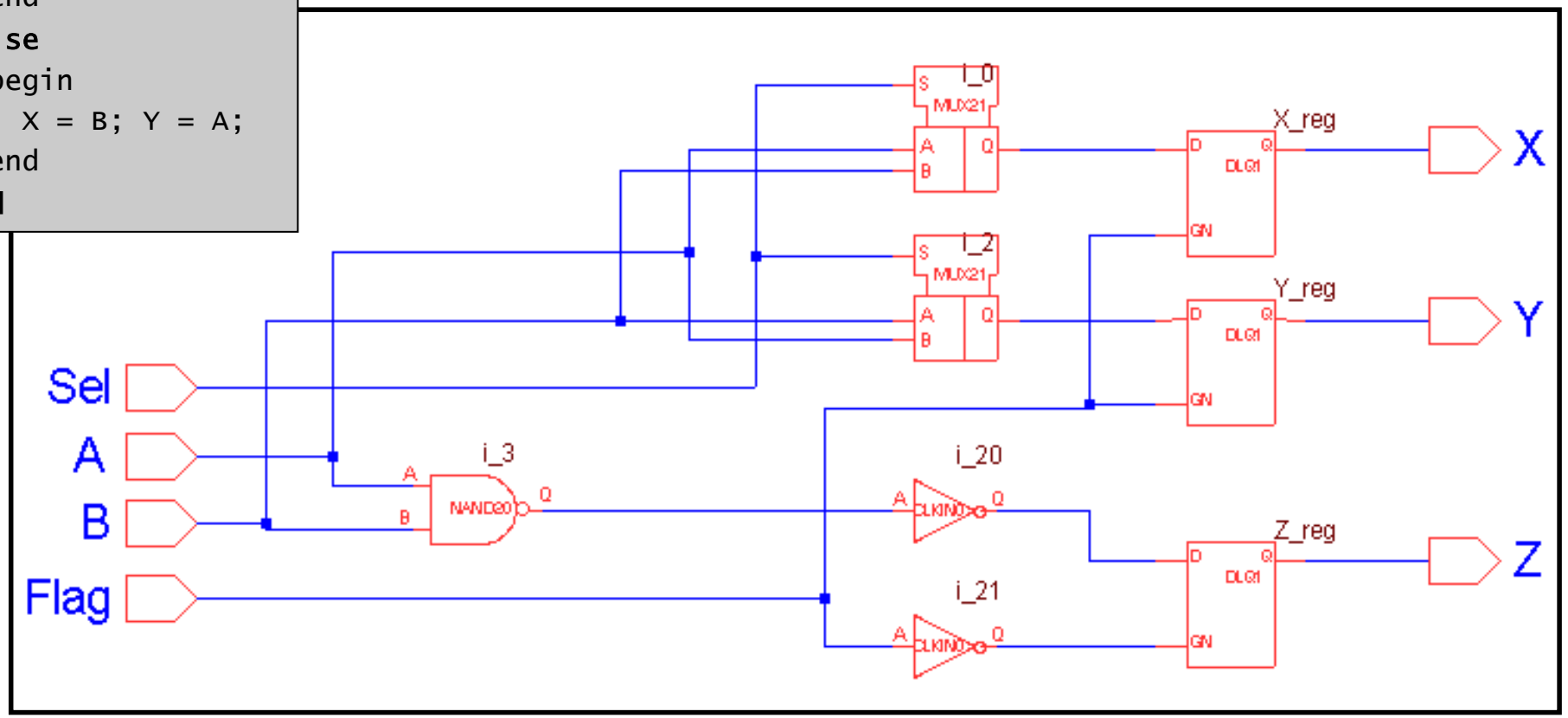
Frei von Latches

```
always
 begin
  if (Flag)
   begin Z = A & B; X = 0; Y = 0; end
  else if (!Sel)
   begin X = A; Y = B; Z = 0; end
  else
   begin X = B; Y = A; Z = 0; end
 end
```

Jeweils Zuweisung an X, Y, Z

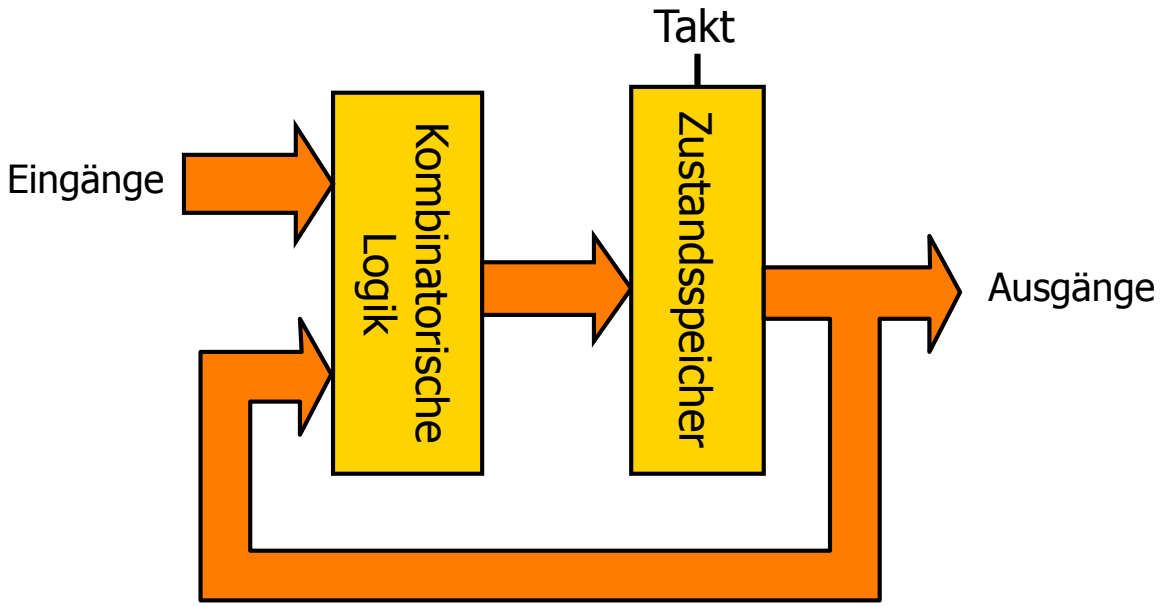
Synthese Ergebnis (erster Fall)

```
always
 begin
  if (Flag)
   Z = A & B;
  else if (!Sel)
   begin
    X = A; Y = B;
   end
  else
   begin
     X = B; Y = A;
   end
 end
```

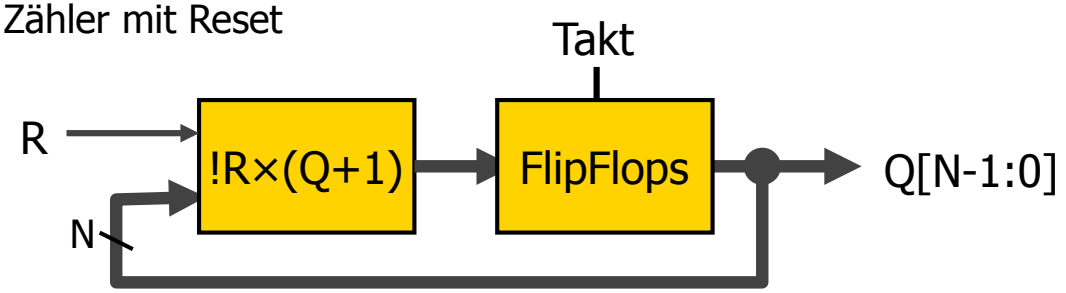


Sequentielle Logik / Zustandsmaschinen

- Zum Speichern des Zustands eines Systems sind **Speicherelemente** notwendig
- Abhängig vom Zustand des Systems und von Eingangsvariablen soll sich der Zustand zu einem bestimmten Zeitpunkt (gegeben durch ein Taktsignal) ändern ⇒ **Zustandsmaschine** ('State machine').
- Beispiel:

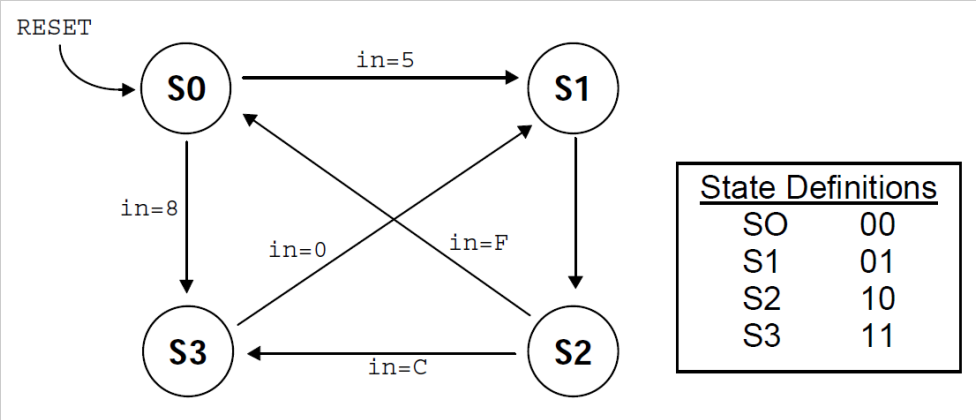


- Einfaches Beispiel: N Bit Zähler mit Reset



Zustandsmaschinen

```
module State_machine (clk, rst, in, state); // from Quicklogic AN17
input clk, rst;
input  [3:0] in;
output [1:0] state;
reg    [1:0] state;

parameter s0 = 2'b00;
parameter s1 = 2'b01;
parameter s2 = 2'b10;
parameter s3 = 2'b11;

always @(posedge clk or posedge rst)
if (rst) state <= s0;
else case (state)
      s0: if      (in == 4'h5) state <= s1;
          else if (in == 4'h8) state <= s3;
          else                 state <= s0;
      s1:                      state <= s2;
      s2: if      (in == 4'hF) state <= s0;
          else if (in == 4'hC) state <= s3;
          else                 state <= s2;
      s3: if      (in == 4'h0) state <= s1;
          else                 state <= s3;
 default: begin state <= 2'bxx; $display ("warning: unknown state"); end
endcase
endmodule
```



State Definitions	
SO	00
S1	01
S2	10
S3	11

Etwas anderer Stil + ‚One hot' encoding

```
// modified part of a one hot encoded state machine (from Xilinx website)

module one_hot (CLOCK, RESET, A, B, C, SINGLE, MULTI, CONTIG);
input  CLOCK, RESET, A, B, C;
output SINGLE, MULTI, CONTIG;
reg    SINGLE, MULTI, CONTIG;

parameter [3:0]  S1 = 4'b0001, S2 = 4'b0010, S3 = 4'b0100, ...; // One Hot states

// Declare current state and next state variables
reg [3:0] CurrentState, NextState;

always @ (posedge CLOCK or posedge RESET)                        // this always
begin                                                            // clockes the
  if (RESET) CurrentState = S1; else CurrentState = NextState; // states
end

always @ (CurrentState or A or B or C)                           // this prepares
Begin                                                            // the next state
  case (CurrentState)
    S1: begin
      MULTI = 1'b0; CONTIG = 1'b0; SINGLE = 1'b0;                // set outputs
      if (A && ~B && C)        NextState = S2;                   // jumps
      else if (A && B && ~C)   NextState = S3;
      else                     NextState = S1;
    end;
    S2: begin
      ...
```

Problem :
Hier wird GENAU b'0001 etc. abgefragt! (nicht 4b'xxx1)

Codierung der Zuständе

- **N Bit Binary** (oder anderer maximaler Code)
  - 2^N Zustände: maximale Ausnutzung der Bits, **wenige FFs** nötig
  - Erkennen des States erfordert alle N Signale, **viel Logik** nötig
- **N Bit One Hot**
  - Nur N Zustände, **viele FFs** nötig
  - Erkennen eines Zustands erfordert nur 1 Signal: **Wenig Logik**
- **N Bit Johnson**
  - 2N Zustände
  - Erkennen eines Zustands erfordert 2 Signale: Mittelweg

Extra Logik nötig, um Ausgangs-Muster zu erzeugen

- **‚Encoded‘**
  - Zustände implementieren direkt die benötigten **Ausgangsmuster.** Keine Ausgangslogik notwendig!
  - Mehrdeutigkeiten über zusätzliche Bits oder Nutzung von don‘t care

Beispiel Zustandskodierung: Aufzug

Zustand	Motor Tür		Motor Aufzug		Binary	OneHot	Johnson	Encoded
	ON	UP	ON	UP				
Steht, 1. Stock, Tür geschlossen	0	X	0	X	000	...000001	000	0000
Steht, 1. Stock, Tür öffnet	1	1	0	X	001	...000010	100	1100
Steht, 1. Stock, Tür offen	0	X	0	X	010	...000100	110	0001
Steht, 1. Stock, Tür schließt	1	0	0	X	011	...001000	111	1000
Fahrt 1.→ 2. Stock (aufwärts)	0	X	1	1	100	...010000	011	0011
Fahrt 2.→ 1. Stock (abwärts)	0	X	1	0	101	...100000	001	0010
....								

Implementierung von State Machines

- Zwei wichtige ‚Styles‘:
  1. Update des Zustands als Funktion der Eingänge in **einem** always Block

```
reg    [1:0] state;

always @(posedge clk or posedge rst)
if (rst) state <= s0;
else case (state)
      s0: state <= s1;
...
```

  2. Blöcke: Always Block für Update, kombinatorischer Block für Berechnung

     Bevorzugte Variante, da aufgeräumter

```
always @ (posedge CLOCK or posedge RESET)
begin
  if (RESET) CurrentState = S1; else CurrentState = NextState;
end

always @ (CurrentState or …)
  case (CurrentState)
    S1: NextState = S2;
...
```

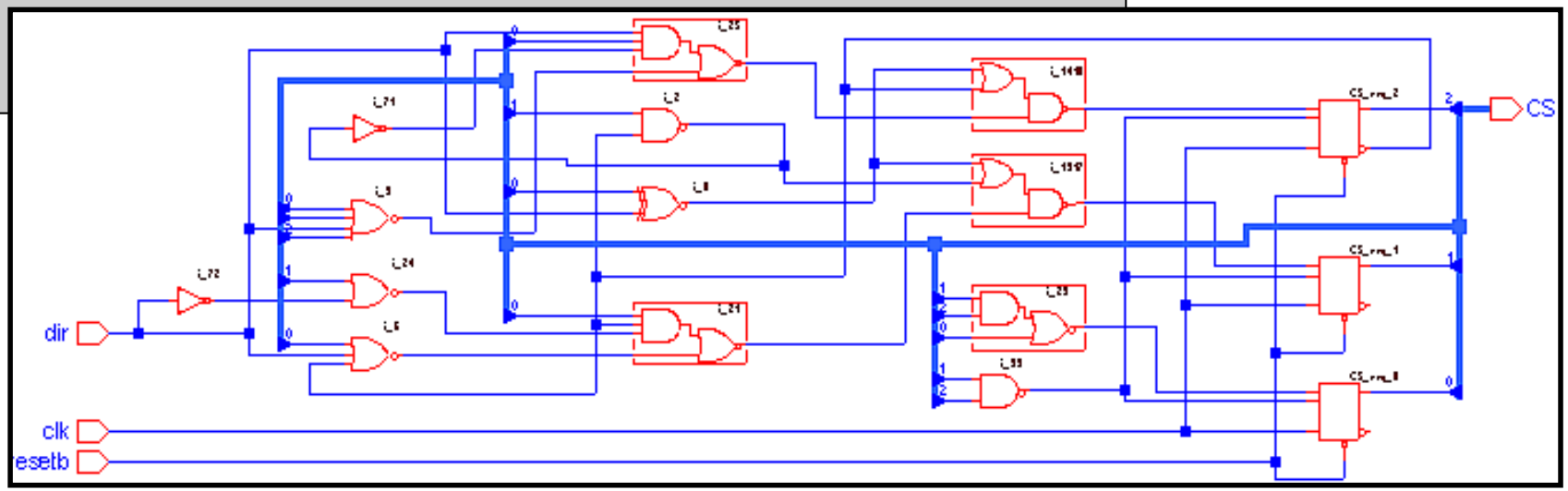
Vergleich verschiedener Implementierungen

- Ein Zustandsautomat (‚Zähler') mit 6 Zuständen, die je nach ‚dir' Signal vorwärts oder rückwärts durchlaufen werden.
- Probiere encoded und one_hot Kodierung
- Probiere synopsys full_case
- Probiere synopsys parallel_case

Versuch 1: encoded, ~~full~~, ~~parallel~~

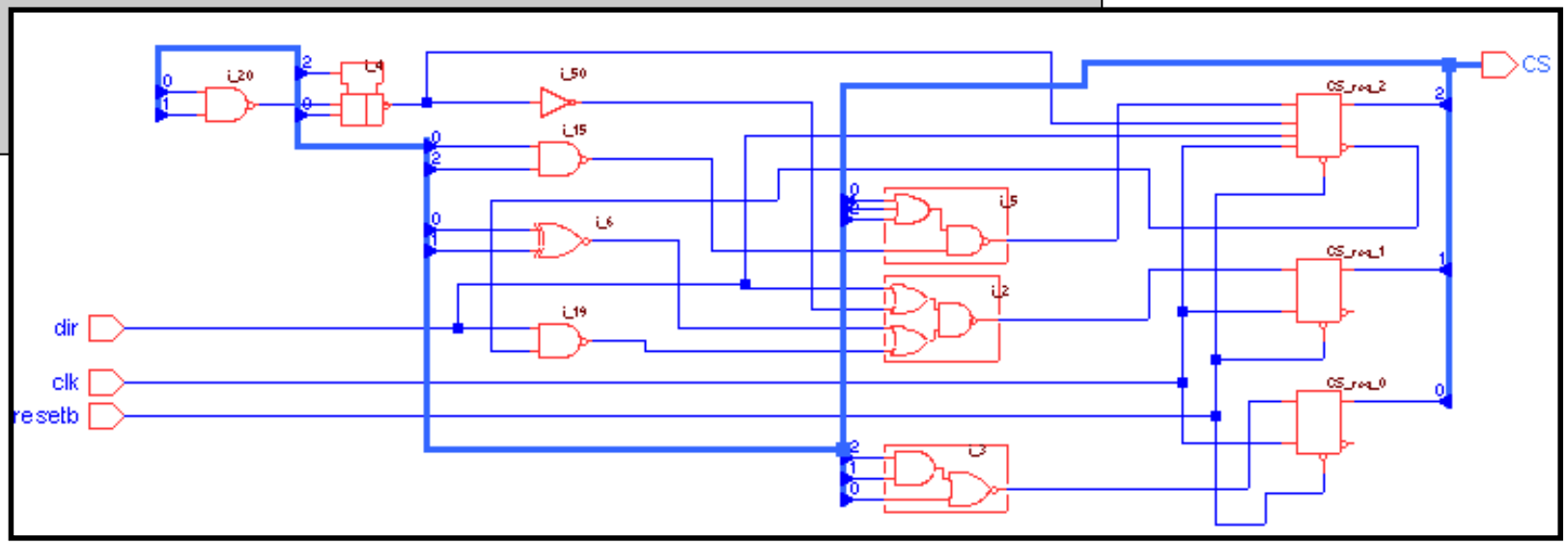
```
module test (input dir, input clk, input resetb, output reg [2:0] CS);

parameter[2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010,
               s3 = 3'b011, s4 = 3'b100, s5 = 3'b101;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
  case (CS)
    s0: CS = dir ? s1 : s5;
    s1: CS = dir ? s2 : s0;
    s2: CS = dir ? s3 : s1;
    s3: CS = dir ? s4 : s2;
    s4: CS = dir ? s5 : s3;
    s5: CS = dir ? s0 : s4;
  endcase
end

endmodule
```

```
Er denkt:

s6: CS = CS;
s7: CS = CS;
```



Versuch 2: encoded, full, ~~parallel~~

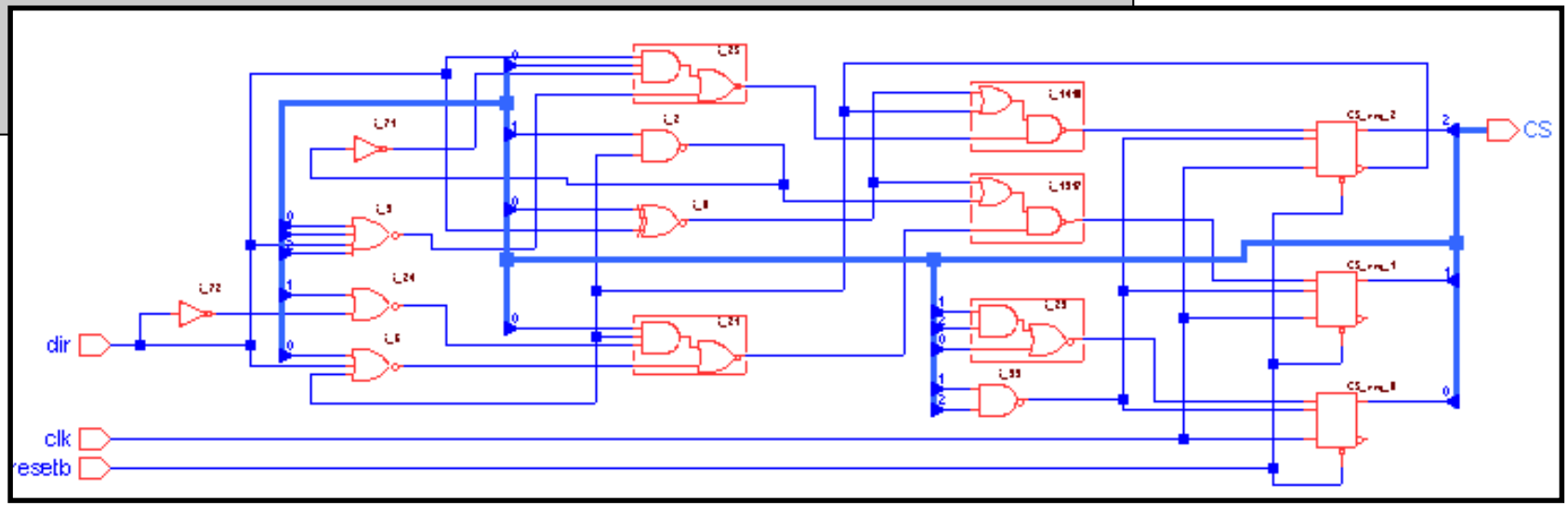
```
module test (input dir, input clk, input resetb, output reg [2:0] CS);

parameter[2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010,
               s3 = 3'b011, s4 = 3'b100, s5 = 3'b101;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
  case (CS) // synopsys full_case
    s0: CS = dir ? s1 : s5;
    s1: CS = dir ? s2 : s0;
    s2: CS = dir ? s3 : s1;
    s3: CS = dir ? s4 : s2;
    s4: CS = dir ? s5 : s3;
    s5: CS = dir ? s0 : s4;
  endcase
end

endmodule
```



Versuch 3: encoded, ~~full~~, parallel

```
module test (input dir, input clk, input resetb, output reg [2:0] CS);

parameter[2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010,
               s3 = 3'b011, s4 = 3'b100, s5 = 3'b101;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
  case (CS) // synopsys parallel_case
    s0: CS = dir ? s1 : s5;
    s1: CS = dir ? s2 : s0;
    s2: CS = dir ? s3 : s1;
    s3: CS = dir ? s4 : s2;
    s4: CS = dir ? s5 : s3;
    s5: CS = dir ? s0 : s4;
  endcase
end

endmodule
```

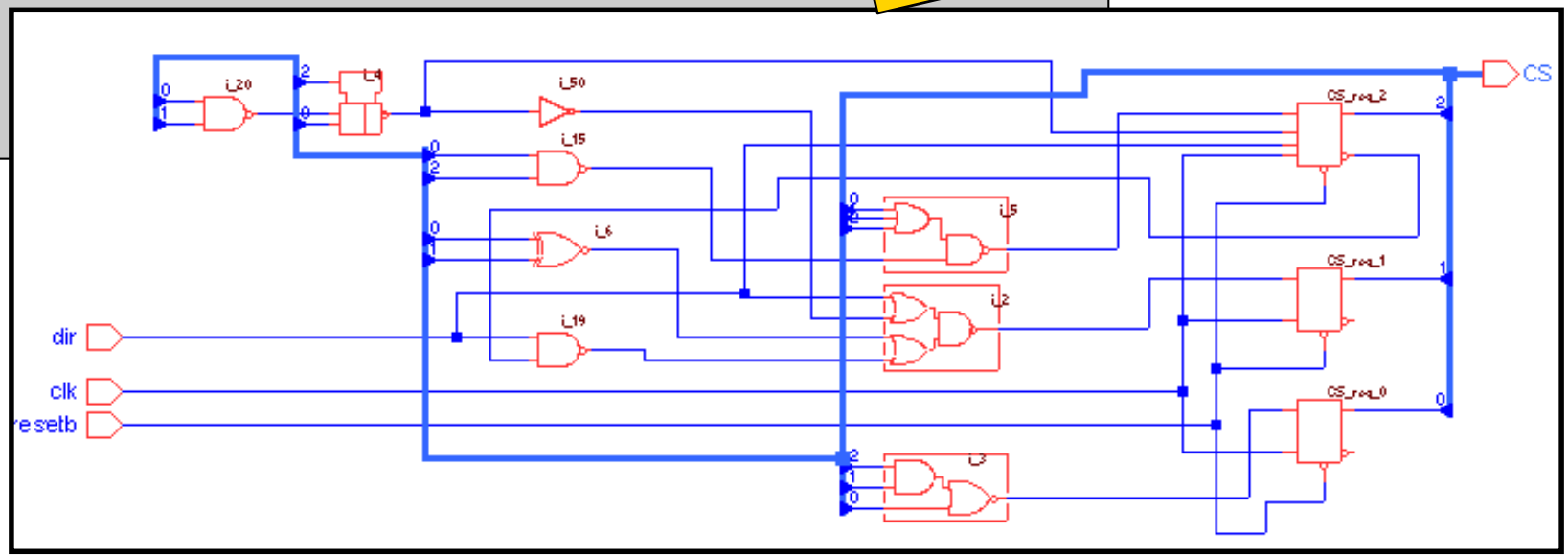
Kein Unterschied zu Versuch 1



Versuch 4: encoded, full, parallel

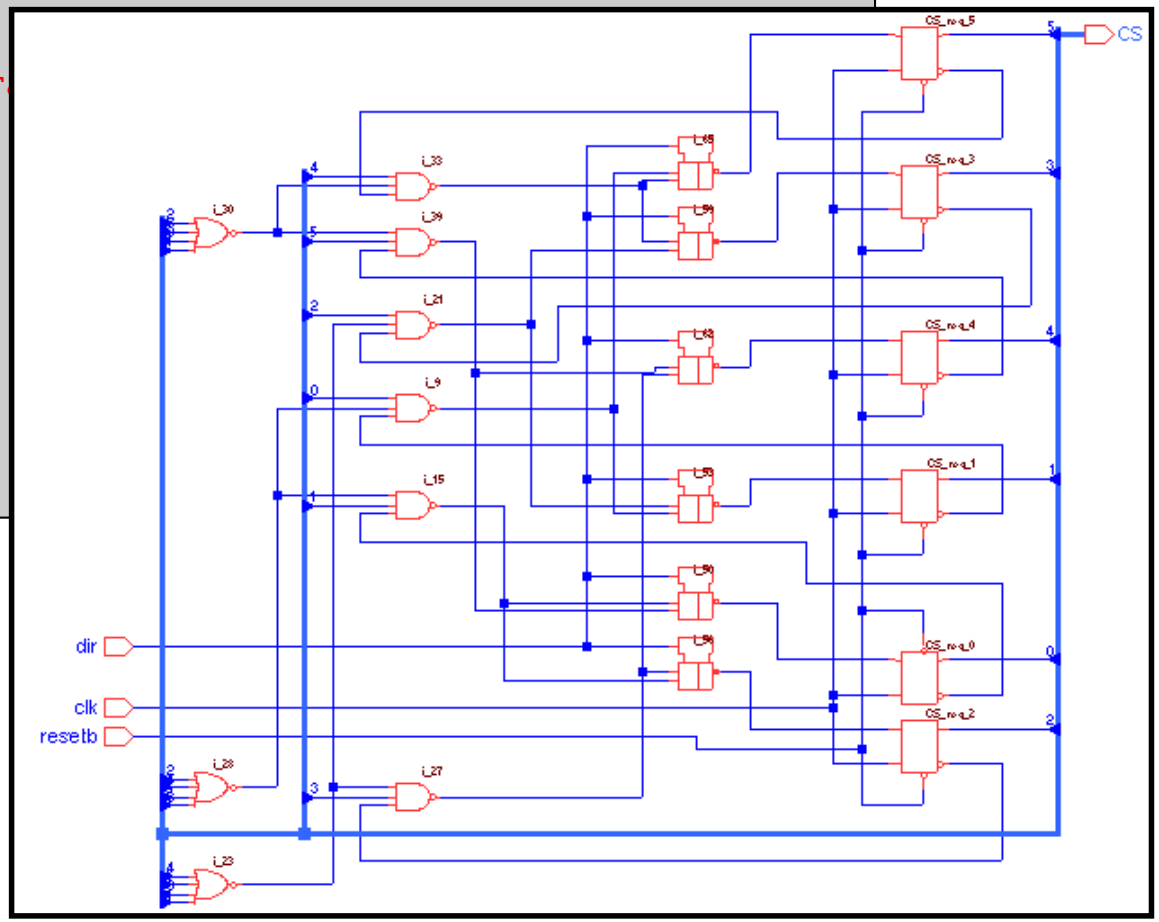
```
module test (input dir, input clk, input resetb, output reg [2:0] CS);

parameter[2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010,
               s3 = 3'b011, s4 = 3'b100, s5 = 3'b101;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
  case (CS) // synopsys full_case parallel_case
    s0: CS = dir ? s1 : s5;
    s1: CS = dir ? s2 : s0;
    s2: CS = dir ? s3 : s1;
    s3: CS = dir ? s4 : s2;
    s4: CS = dir ? s5 : s3;
    s5: CS = dir ? s0 : s4;
  endcase
end

endmodule
```

Kein Unterschied zu Versuch 2



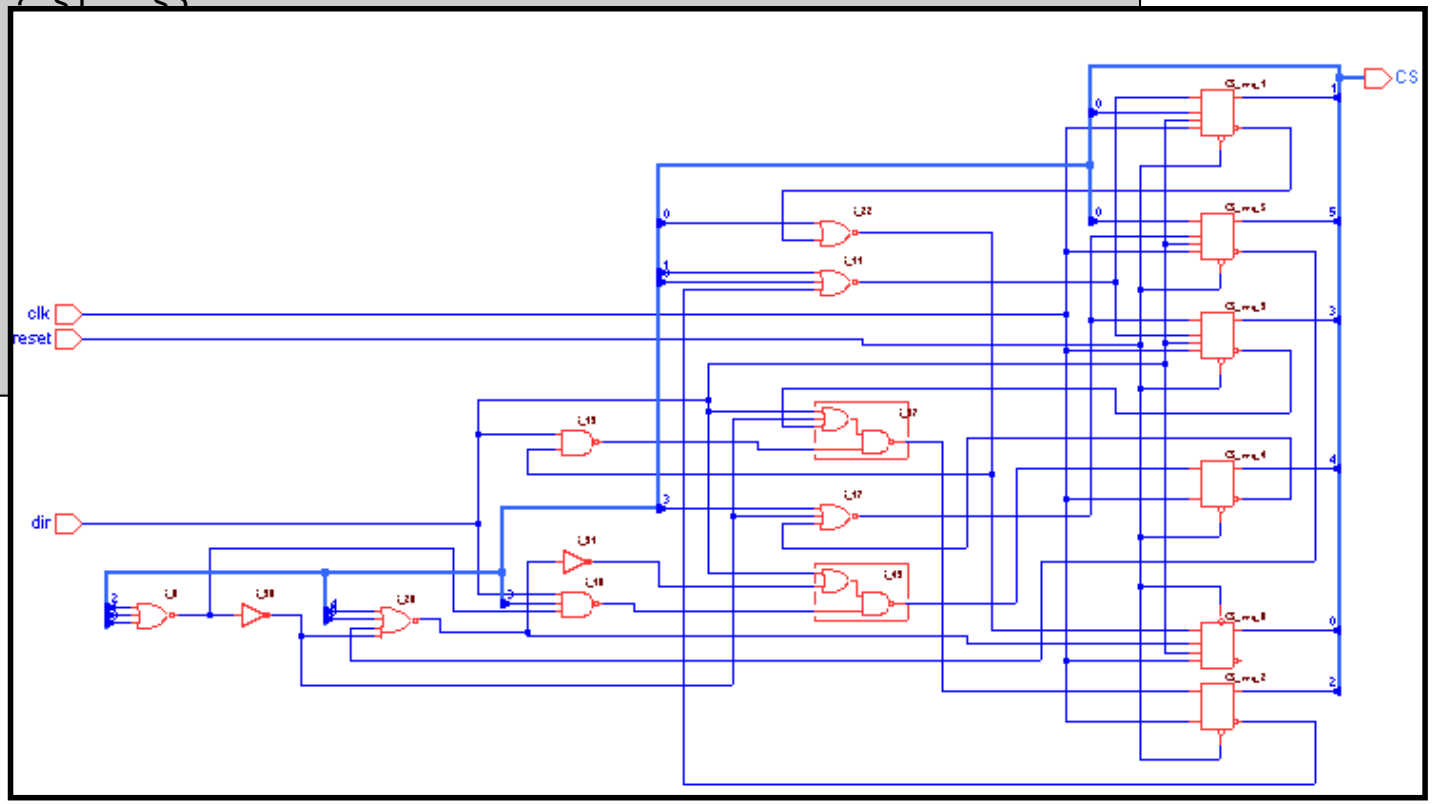
Versuch 5: one_hot, full, parallel, exact comparison

```
module test_noxstates (input dir, input clk, input resetb, output reg [5:0] CS);

parameter[5:0] s0 = 6'b000001, s1 = 6'b000010, s2 = 6'b000100;
parameter[5:0] s3 = 6'b001000, s4 = 6'b010000, s5 = 6'b100000;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
  case (CS) // synopsys full_case par
    s0: CS = dir ? s1 : s5;
    s1: CS = dir ? s2 : s0;
    s2: CS = dir ? s3 : s1;
    s3: CS = dir ? s4 : s2;
    s4: CS = dir ? s5 : s3;
    s5: CS = dir ? s0 : s4;
  endcase
end

endmodule
```



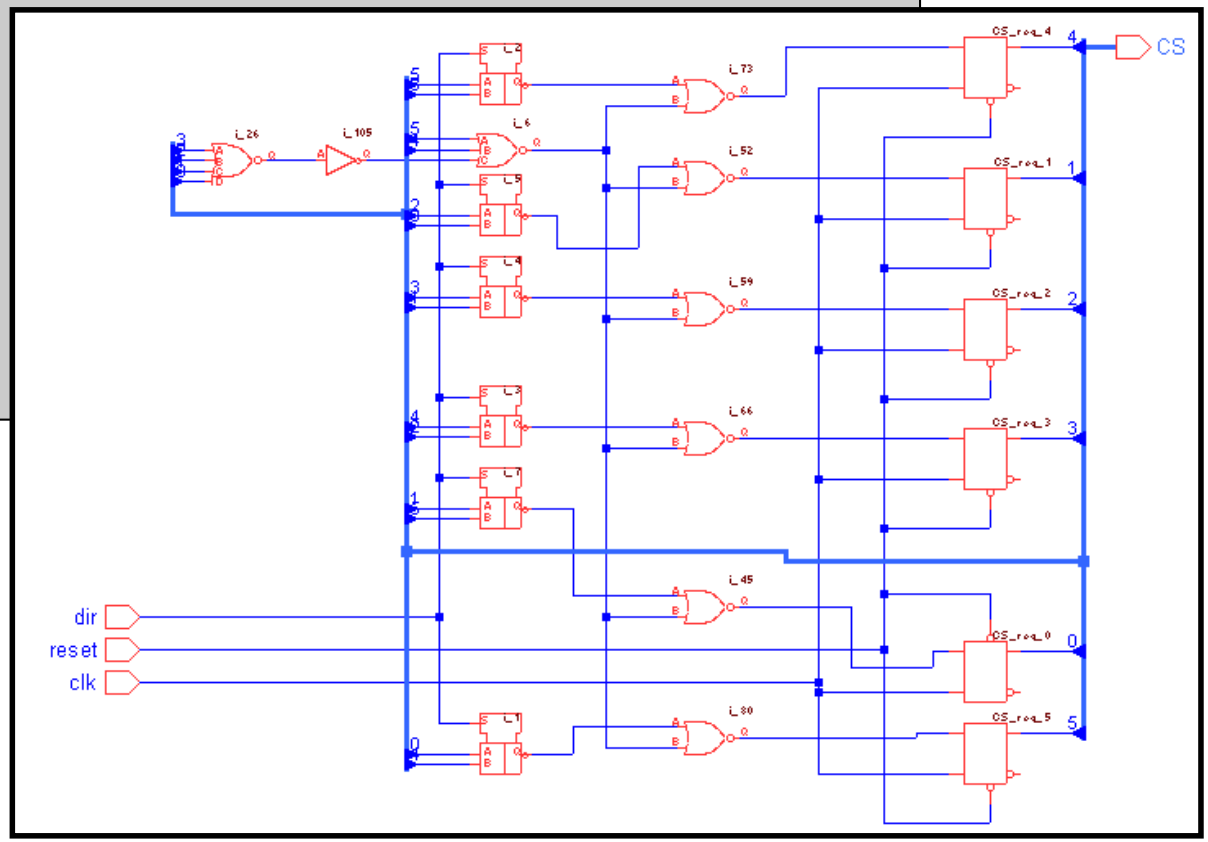
Versuch 6: one_hot, full, ~~parallel~~, casex

```
module test_noxstates (input dir, input clk, input resetb, output reg [5:0] CS);

parameter[5:0] s0 = 6'b000001, s1 = 6'b000010, s2 = 6'b000100;
parameter[5:0] s3 = 6'b001000, s4 = 6'b010000, s5 = 6'b100000;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
   casex (CS) // synopsys full_case
    6'bxxxxx1: CS = dir ? s1 : s5;
    6'bxxxx1x: CS = dir
    6'bxxx1xx: CS = dir
    6'bxx1xxx: CS = dir
    6'bx1xxxx: CS = dir
    6'b1xxxxx: CS = dir
  endcase
end

endmodule
```



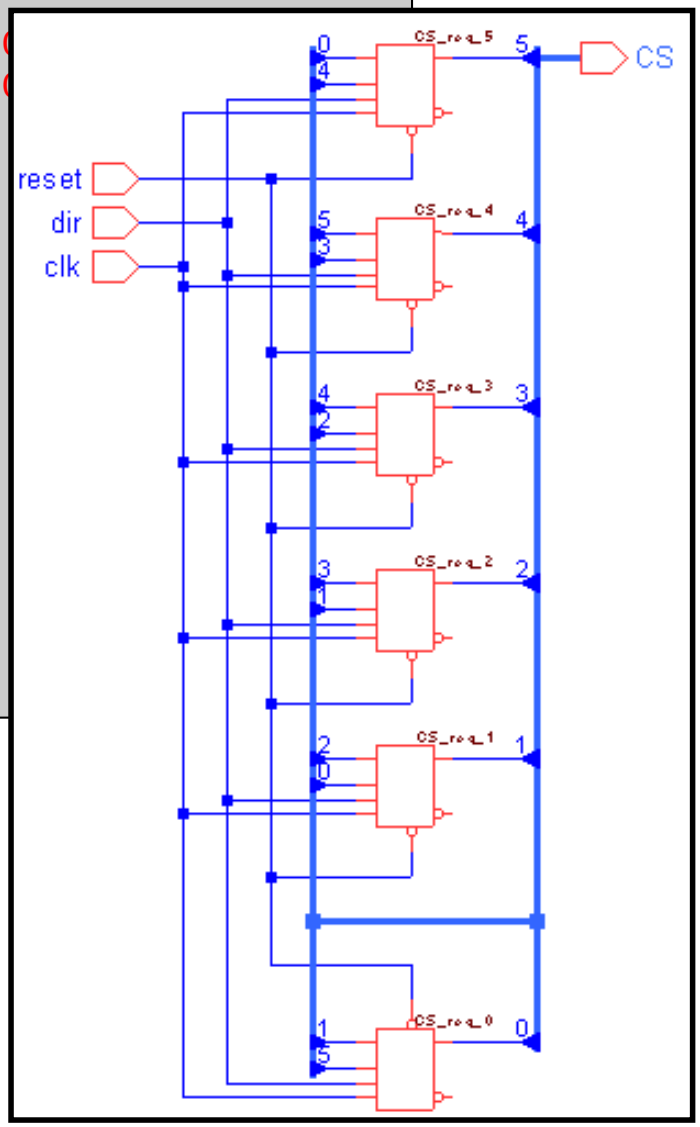
Versuch 7: one_hot, ~~full~~, parallel, casex

```
module test_noxstates (input dir, input clk, input resetb, output reg [5:0] CS);

parameter[5:0] s0 = 6'b000001, s1 = 6'b000010, s2 = 6'b000100;
parameter[5:0] s3 = 6'b001000, s4 = 6'b010000, s5 = 6'b100000;

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
   casex (CS) // synopsys parallel_case
    6'bxxxxx1: CS = dir ? s1 : s5;
    6'bxxxx1x: CS = dir ? s2 : s0;
    6'bxxx1xx: CS = dir ? s3 : s1;
    6'bxx1xxx: CS = dir ? s4 : s2;
    6'bx1xxxx: CS = dir ? s5 : s3;
    6'b1xxxxx: CS = dir ? s0 : s4;
  endcase
end

endmodule
```



Versuch 8: one_hot, full, parallel, casex

```
module test_noxstates (input dir, input clk, input resetb, output reg [5:0] CS);

parameter[5:0] s0 = 6'b000001, s1 = 6'b000010, s2 = 6'b000100
parameter[5:0] s3 = 6'b001000, s4 = 6'b010000, s5 = 6'b100000

always @(posedge clk or negedge resetb)
if (~resetb) CS = s0; else
begin
   casex (CS) // synopsys full_case parallel_case
    6'bxxxxx1: CS = dir ? s1 : s5;
    6'bxxxx1x: CS = dir ? s2 : s0;
    6'bxxx1xx: CS = dir ? s3 : s1;
    6'bxx1xxx: CS = dir ? s4 : s2;
    6'bx1xxxx: CS = dir ? s5 : s3;
    6'b1xxxxx: CS = dir ? s0 : s4;
  endcase
end

endmodule
```



'Professionelle' Zustandsmaschine (Encoded)

```
reg [ 8:0 ] current_state;
reg [ 8:0 ] next_state;

// Ausgangssignale sind direkt im Zustand kodiert:
//                            v--- Ausgangssignal 1
//                            |v-- Ausgangssignal 2
//                            ||v- Ausgangssignal 3
//                            |||
localparam S_RESET        = 9'b100010100;
localparam S_IDLE         = 9'b001010000;
localparam S_START        = 9'b001010010;

// Zuweisungen zu den Ausgangsleitungen
assign Out1 = current_state[ 2 ];

// Eingangssignale zu Vektor zusammenfassen
wire [ 4:0 ] inputs = { in1, in2, in3, in4, in5 };

// Detail: Zähler für eine Verzögerung
reg [ 2:0 ] counter;
```

Getakteter Teil

```
always @( posedge CLK or posedge RESET )
begin
   if( RESET )
   begin
      current_state <= S_RESET;      // Anfangszustand der Statemachine
      counter       <= 'd0;          // Zähler auf Null setzen
   end
   else
   begin
      current_state <= next_state;   // Neuen Zustand zuweisen
      if( current_state == S_COUNT )
        counter <= counter + 1;      // Zähle, wenn in Zustand S_COUNT
   end
end
```


Ermittlung des nächsten Zustands

```
always @(*) begin
   casex( { input, current_state } )
//          v in1
//          |v in2
      { 5'bxxxxx, S_RESET }          : next_state = S_IDLE;

      { 5'bxxx0x, S_IDLE }           : next_state = S_IDLE;
      { 5'bxxx1x, S_IDLE }           : next_state = S_START;

      { 5'bxxxxx, S_START }          : next_state = S_WAIT_RAMP;


      { 5'bxxx1x, S_DONE        }    : next_state = S_DONE;
      { 5'bxxx0x, S_DONE        }    : next_state = S_RESET;

//    ...
//    ...

      default                        : next_state = S_RESET;
   endcase
end
```

Systemfunktionen und -Prozeduren

- Es gibt eine Reihe von Systemfunktionen und –Prozeduren, die ausschließlich der **Simulation** dienen.
- Zweck: Debugging, Logging, Datei Ein- und Ausgabe, Simulation anhalten, etc.
- Sie beginnen immer mit einem vorangestellten „**$**"-Zeichen.
- Sie sind (selbstverständlich) nicht synthetisierbar.
- Häufigste Verwendung: Testbenches.

	Beispiel
Text in der Konsole anzeigen	**$display**(„Inhalt Datenbus: %b.", databus);
Signal/Variable überwachen	**$monitor**(„Inhalt Datenbus: %b.", databus);
Gegenwärtige Simulationszeit	**$time**
Simulation beenden	**$stop** **$finish**
Zufallszahl erzeugen	**$random**(seed);
In eine Datei schreiben	**$fdisplay**(…) **$fwrite**(…) **$fmonitor**(…)
Aus einer Datei lesen	**$readmemx**(„testvectors.dat", mem, 0, 128);

Compiler- und Synthesedirektiven

- Es gibt zwei Arten von Direktiven:
  1. Standard Verilog-Compilerdirektiven. Beginnen immer mit einem Tüddlchen (`).
  2. Toolspezifische Synthesedirektiven. Sind syntaktisch gesehen mit einem Kommentar gleich, da sie immer mit einem Doppelslash (//) beginnen.
- Synthesedirektiven sind toolspezifisch. Ist einem Synthesetool die Direktive unbekannt, wird sie wie ein Kommentar behandelt.
- Synthesedirektiven von Synopsys werden fast immer unterstützt (//synopsys ...).

	Beispiel
Quelldatei einfügen	**`include**("C:\Projects\myverilog.v")
Zeiteinheit und Genauigkeit festlegen	**`timescale** 10 ns / 1 ns
Makro festlegen	**`define** NOP 8'h1f
Fallunterscheidung	**`ifdef `ifndef `else `elsif `endif**
Synthese ausschalten	//synopsys synthesis_off //synopsys translate_off
Synthese einschalten	//synopsys synthesis_on //synopsys translate_on
Diverse Synthesedirektiven	//synopsys infer_mux //synopsys state_vector //...

Zusammenfassung (1/2)

```
module M (P1, P2, P3, P4);
  input P1, P2;
  output [7:0] P3;
  inout P4;

  reg [7:0] R1, M1[1:1024];
  wire W1, W2, W3, W4;
  parameter DATAWIDTH = 8;

  initial
  begin : BlockName
    // Statements
  end

  always @(posedge clk)
  begin
    // Statements
  end

  // Continuous assignments...
  assign W1 = Expression;
  wire [3:0] #3 W2 = Expression;
```

```
  // Module instances...
  COMP U1 (W3, W4);
  COMP U2 (.P1(W3), .P2(W4));

  task T1;
    input A1;
    inout A2;
    output A3;
    begin
      // Statements
    end
  endtask

  function [7:0] F1;
    input A1;
    begin
      // Statements
      F1 = Expression;
    end
  endfunction

endmodule
```

Zusammenfassung (2/2)

```
#delay
wait (Expression)
@(A or B or C)
@(posedge Clk)

Reg = Expression;
Reg <= Expression;
VectorReg[Bit] = Expression;
VectorReg[MSB:LSB] = Expression;
Memory[Address] = Expression;
assign Reg = Expression

MyTaskCall(...);

if (Condition)
   ...
else if (Condition)
   ...
else
   ...
```

```
case (Selection)
   Choice1 :
   ...
   Choice2, Choice3 :
   ...
   default : ...
endcase

for (I=0; I<MAX; I=I+1)
   ...
repeat (8)
   ...

while (Condition)
   ...

forever
   ...
```

Fehlerquellen

- Assign HighByte = data[7:4];

- HighByte ist hier **nicht** automatisch 4 Bit breit. Hier wird nur **eine** Leitung erzeugt, die anderen Bits von data gehen verloren!