



A. Koch

Einführung in Computer Microsystems

4. Einführung in die Logiksynthese

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2007



A. Koch

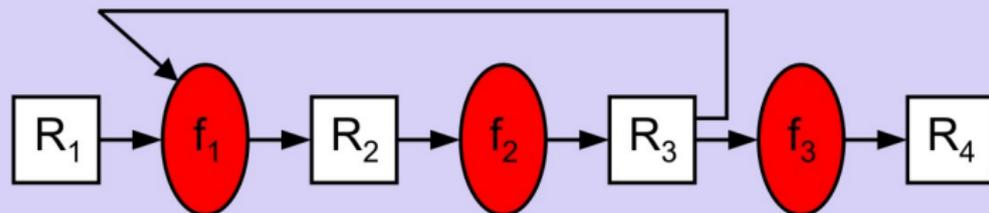
Einführung



- Abbildung von RTL-Modell auf Gattermodell
 - Register-Transfer-Ebene auf Logikebene
- Wichtige Hersteller von **Entwurfswerkzeugen**
 - Für ASICs: Synopsys
 - Für FPGAs: Synplicity
 - Gibt aber auch noch diverse andere Anbieter



Optimiert im wesentlichen Logik **zwischen** getakteten Registern

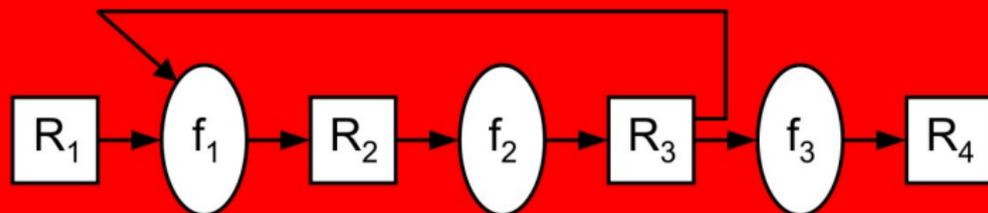


Vergleich: High-Level-Synthese



A. Koch

Beginnt **oberhalb** von RTL und optimiert über Taktgrenzen **hinweg**

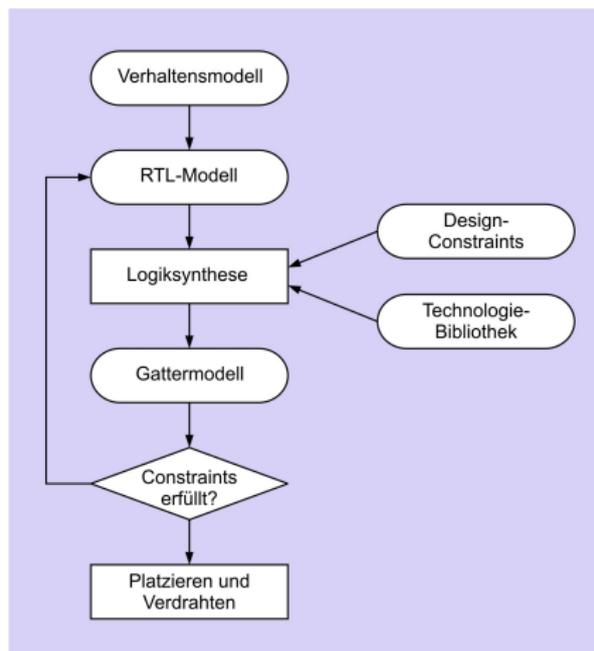


Noch experimentell, nur eingeschränkte praktische Bedeutung

Ablauf der Logiksynthese



A. Koch



- RTL-Modell in Verilog
- Design-Constraints
 - Wie schnell?
 - Wie groß?
 - (Wieviel Energie?)
- Zieltechnologie
 - AND, OR
 - Addierer, Flip-Flops
 - Abbildung auf LUTs
 - Genaue Laufzeiten
 - Genaue Flächenangaben

Vorteile der Logiksynthese



A. Koch

- Kürzere **Entwurfszeit**
- Weniger **fehler**anfällig
- **Anforderungen** an **Zeit** und **Fläche** aufstellbar
- **Portabilität** zwischen verschiedenen Chip-Herstellern
- Leichtere **Exploration** des Entwurfsraumes
 - Wieviel langsamer, wenn 25% kleiner?
- Einheitlicher **Entwurfstil** bei Team-Arbeit
- Leichtere **Wiederverwendung** von (Teil-)Entwürfen

Wichtigste synthetisierbare Verilog-Konstrukte



A. Koch

Signale und Variablen

wire, reg

prozedural

always, begin, end,
if, else,
case,
function, task, =, <=

Struktur

module,
input, inout, output,
parameter,

assign

Eingeschränkt: for

Einige **nicht**synthetisierbare Verilog-Konstrukte



A. Koch

- **initial**: Stattdessen explizites Reset-Verhalten beschreiben
 - **Zeitkontrolle**: # und @ innerhalb von Block
 - Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie
- ➔ Prä- und Post-Synthese-Simulationen können differieren

Synthetisierbare Operatoren



A. Koch

arithmetisch

`*`, `/`, `+`, `-`, `%`

logisch

`!`, `&&`, `||`

bit-weise

`~`, `&`, `|`, `^`, `^~`, `~^`

Reduktion

`&`, `~&`, `|`, `~|`, `^`, `^~`, `~^`

Relation

`>`, `<`, `>=`, `<=`

Gleichheit

`==`, `!=` **aber kein `===`
und `!==` mit `x` und `z`**

Shift

`>>`, `<<`, `<<<`, `>>>`

Konkatenation

`{` `}`

bedingt

`?:`



A. Koch

Synthesergebnisse

Vorgehen



A. Koch

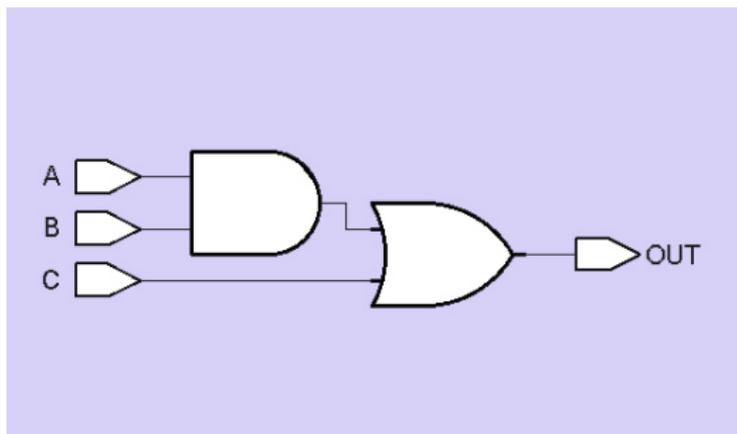
- Zunächst Abbildung auf **allgemeines** Gattermodell
- Noch weitgehend **ohne** Berücksichtigung der Zieltechnologie
- Reine **Zwischendarstellung**

Synthese einer `assign`-Anweisung



```
assign OUT = (A & B) | C;
```

A. Koch



Annahme hier: Alle Signale 1b breit

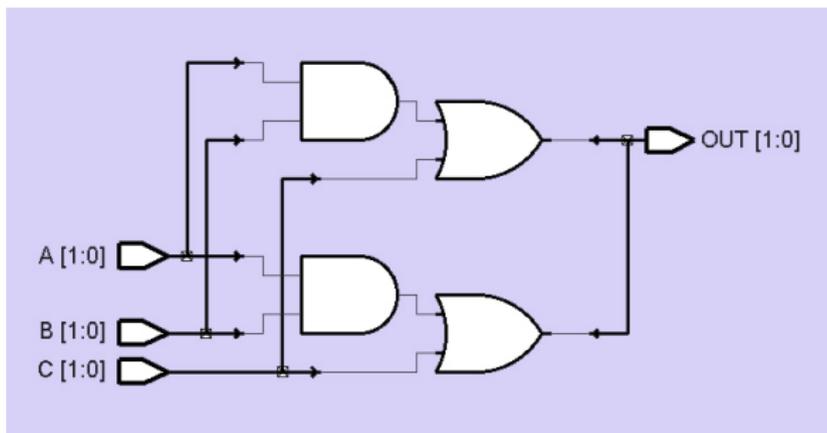
Synthese einer `assign`-Anweisung

Hier: 2b breite Signale



```
assign OUT = (A & B) | C;
```

A. Koch



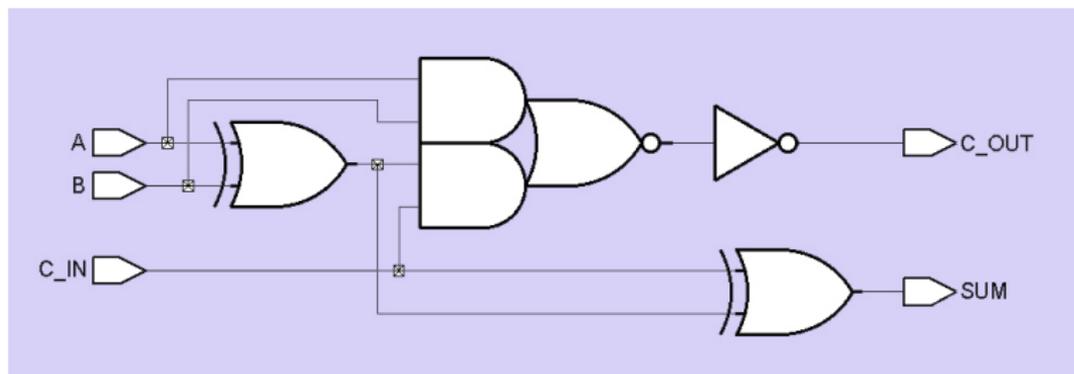
Synthese einer `assign`-Anweisung



```
assign {C_OUT, SUM} = A + B + C_IN
```

A. Koch

1b-Volladdierer



Merkwürdiges Gatter in der Mitte: AND-OR-INVERT (AOI),
sehr effizient in ASIC-Technologie realisierbar

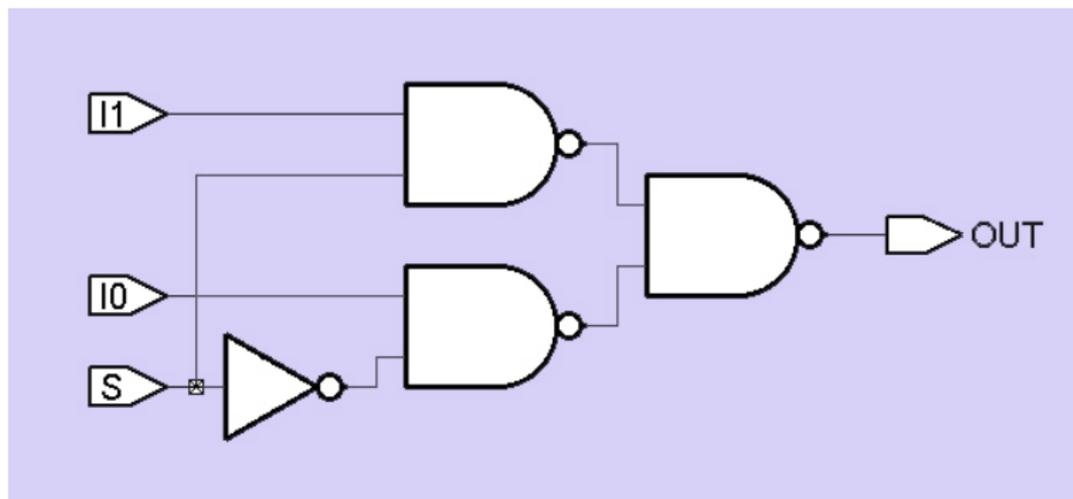
Synthese einer `assign`-Anweisung



```
assign OUT = (S) ? I1 : I0
```

A. Koch

Multiplexer



Synthese von `if/else` und `case`

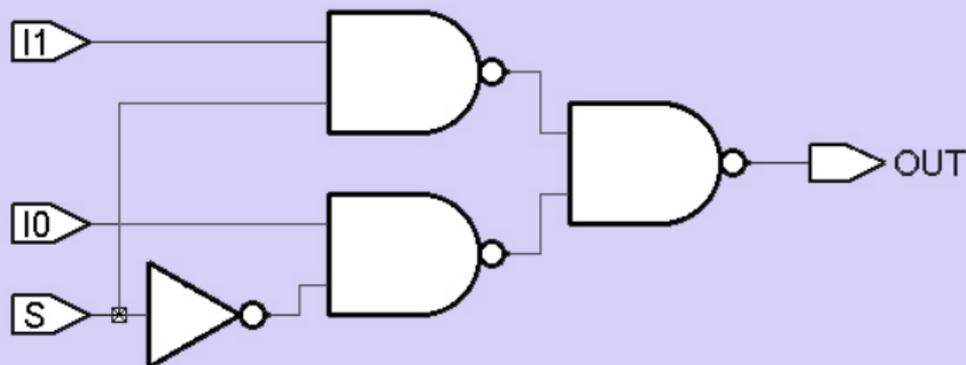


```
if (S) OUT = I1;  
else  OUT = I0;
```

```
case (S)  
  0: OUT = I0;  
  1: OUT = I1;  
endcase
```

A. Koch

Multiplexer



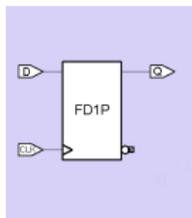
Synthese von Speicherelementen



A. Koch

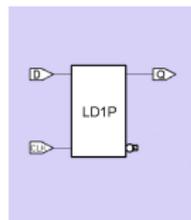
```
always @ (posedge CLK)
  Q <= D;
```

Vorderflankengesteuertes
Flip-Flop



```
always @ (CLK or D)
  if (CLK) Q = D;
```

Pegelgesteuertes Latch

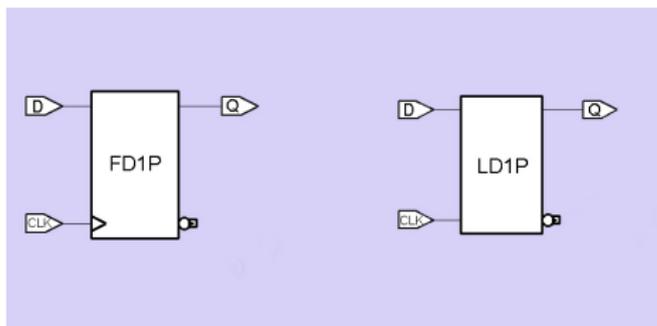


Wichtig: Unterschiede



- **Hardware-Register**
 - (flankengesteuertes) Flip-Flop
 - (pegelgesteuertes) Latch

A. Koch



- **Verilog-Datentyp `reg`**

➔ **Nicht** identisch



A. Koch

Register-Synthese

Grundlage der Modellierung: `always`-Blöcke



A. Koch

- **flankengesteuerte** `always`-Blöcke

```
always @(posedge CLK)
```

...

```
always @(posedge CLK, negedge nRESET)
```

...

- **flankenfreie** `always`-Blöcke

```
always @(CLK, D)
```

...

```
always @ (A, B, C_IN)
```

...

Grundlegende Definitionen

Potenzielle Register und Vollständigkeit



Potenzielle Register

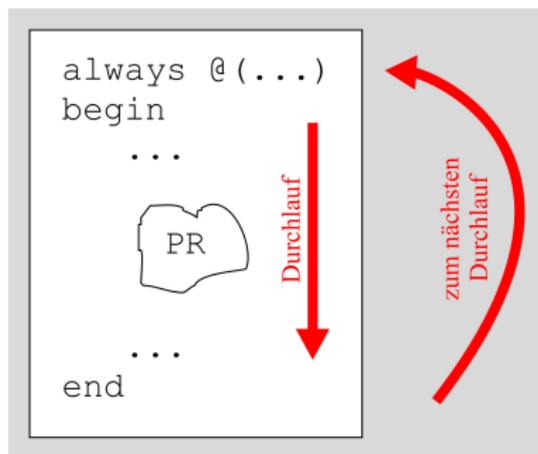
Eine Variable q ist ein potenzielles Register (PR), wenn sie in einem always-Block **geschrieben** wird ($q = \dots$, $q \leq \dots$)..

A. Koch

Vollständigkeit

Ein potenzielles Register ist vollständig, falls es bei **jedem** Durchlauf des always-Blocks **nicht-redundant** geschrieben wird.

Nicht-redundant: Nicht mit sich selber, also nicht $q = q$



Grundlegende Definitionen



Räumliche Lokalität

Ein potenzielles Register ist räumlich lokal, wenn es nur innerhalb **eines** always-Blockes verwendet wird (lesend oder schreibend).

A. Koch

```
always @(...)
```

```
begin
```

```
...
```



```
...
```

```
end
```

PR darf nicht außerhalb
des always-Blockes
referenziert sein



Beispiele: Räumliche Lokalität



A. Koch

```
module r_lokal_1 (  
  input wire A,  
  output reg C);  
  
reg B;  
  
always @ (A, B)  
begin  
  B = ~A;  
  if (B) C = A;  
  else C = ~A;  
end  
endmodule
```

```
module r_lokal_2 (  
  input wire A);  
  
reg B, C;  
  
always @ (A)  
  C = A;  
  
always @ (C)  
  B = ~C;  
  
endmodule
```

In beiden Modulen ist **B** räumlich lokal, nicht aber **C**.

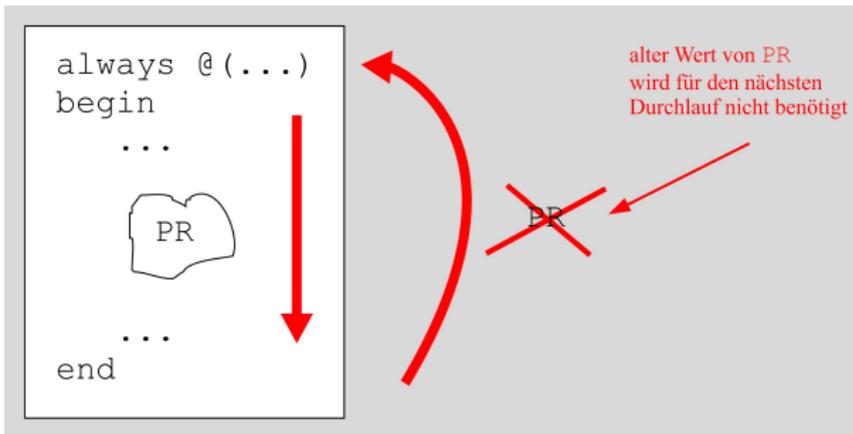
Grundlegende Definitionen



Zeitliche Lokalität

Ein potenzielles Register ist zeitlich lokal (kurz: **lokal**), falls es räumlich lokal ist **und** nie **vor** dem Schreiben gelesen wird.

A. Koch



Der alte Wert braucht also nicht zwischengespeichert zu werden.

Beispiele: Zeitliche Lokalität



A. Koch

```
module z_lokal_1 (  
  input wire A,  
  output reg C);
```

```
reg TMP;
```

```
always @ (A, TMP)  
begin  
  TMP = ~A;  
  C = TMP;  
end  
endmodule
```

TMP ist zeitlich lokal

```
module z_lokal_2 (  
  input wire A, B,  
  output reg C);
```

```
reg TMP;
```

```
always @ (A, B, TMP)  
begin  
  if (B) TMP = ~A;  
  C = TMP;  
end  
endmodule
```

TMP ist **nicht** zeitlich lokal



A. Koch

Flankenfreier Always-Block

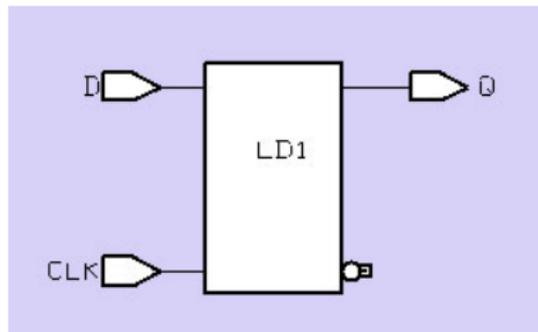
Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



A. Koch

```
always @(CLK, D)  
if (CLK) Q = D;
```



Latch wegen **unvollständigem** Q

Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



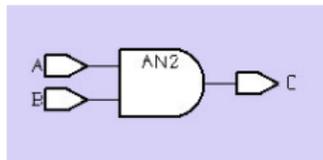
A. Koch

```
always @(CLK, D)
  if (CLK) Q = D;
else    Q = 0;
```

```
always @(A, B)
  if (A) C = B;
else  C = 0;
```

Latch vermieden
da `q` nun
vollständig

Signalnamen wie
CLK irrelevant.
Kombinatorische
Logik!



Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



```
module decoder (  
  input wire [3:0] I;  
  output reg [9:0] DECIMAL);  
always @(I)  
  case (I)  
    4'h0: DECIMAL = 10'b0000000001;  
    4'h1: DECIMAL = 10'b0000000010;  
    4'h2: DECIMAL = 10'b0000000100;  
    4'h3: DECIMAL = 10'b0000001000;  
    4'h4: DECIMAL = 10'b0000010000;  
    4'h5: DECIMAL = 10'b0000100000;  
    4'h6: DECIMAL = 10'b0001000000;  
    4'h7: DECIMAL = 10'b0010000000;  
    4'h8: DECIMAL = 10'b0100000000;  
    4'h9: DECIMAL = 10'b1000000000;  
  endcase  
endmodule
```

A. Koch

- Latch wegen **unvollständigem case**
- Wie vollständig formulieren?
- Durch angeben von **default**



- Ein **unvollständiges** potenzielles Register (PR) erzeugt zunächst ein **Latch**
- Ist das PR jedoch **lokal**, wird das Latch aber anschließend wegoptimiert
- Auf Nummer sicher gehen!
- Damit PR **kein** Latch wird
 - PR **vollständig** beschreiben
 - oder PR nur **lokal** verwenden

Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



```
module z_lokal (  
    input wire A,  
    output reg C);
```

```
reg TMP;
```

```
always @ (A, TMP)
```

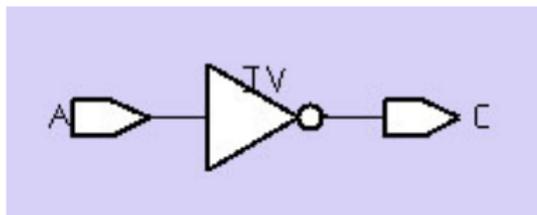
```
begin
```

```
    TMP = ~A;
```

```
    C = TMP;
```

```
end
```

```
endmodule
```



A. Koch

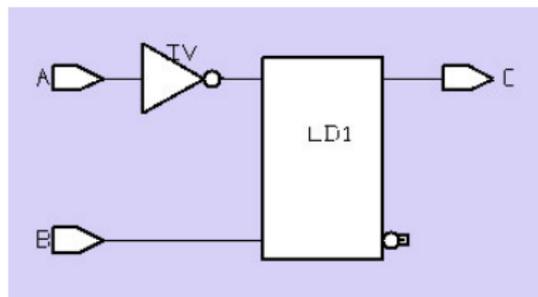
Latch wird vermieden, da **TMP** **vollständig** ist (und zeitlich lokal).

Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



```
module lokal (  
    input wire A, B,  
    output reg C);  
  
reg TMP;  
  
always @ (A, B, TMP)  
begin  
    if (B) TMP = ~A;  
    C = TMP;  
end  
endmodule
```



A. Koch

Latch entsteht, da **TMP unvollständig** ist und **nicht** zeitlich lokal.

Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



A. Koch

```
module lokal(  
  input wire A, B, C,  
  output reg D);
```

```
  reg TMP;
```

```
  always @ (TMP, A, B, C)
```

```
    if (C) begin
```

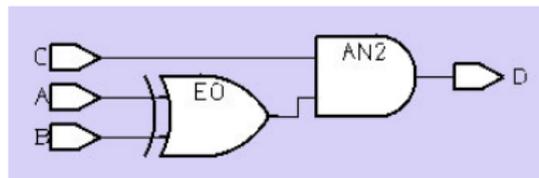
```
      TMP = A + B;
```

```
      D = TMP;
```

```
    end
```

```
    else D = 0;
```

```
endmodule
```



Latch wird vermieden, da **TMP** zwar **unvollständig** ist, aber räumlich und zeitlich **lokal**.

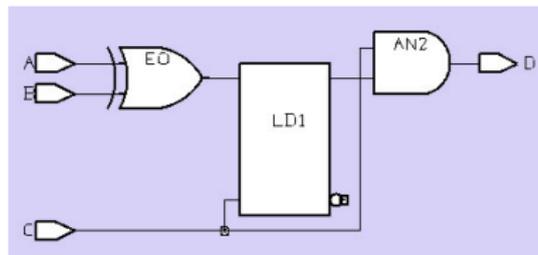
Beispiele: Logik oder Latch?

Flankenfreier **always**-Block



A. Koch

```
module lokal (  
  input wire A, B, C,  
  output reg D);  
  
reg TMP;  
  
always @ (TMP, A, B, C)  
  if (C) begin  
    D = TMP;  
    TMP = A + B;  
  end  
  else D = 0;  
  
endmodule
```



Latch entsteht, da **TMP** **unvollständig** ist und nur räumlich, nicht aber **zeitlich lokal** ist.

Richtlinien

Flankenfreier `always`-Block



A. Koch

- Zur **Vermeidung** von Latches
 - PR **vollständig** beschreiben
 - oder nur **lokal** verwenden
- Verilog-Funktionen ergeben **kein** Latch
 - Sie haben keinen internen Zustand
 - Keine globalen oder static-Variablen wie in z.B. in Java
- In flankenfreien `always`-Blöcken immer **alle** Lesevariablen in Aktivierungsliste
 - `always @(*)`
- Hier immer die **blockende** Zuweisung = verwenden!



A. Koch

Getakteter Always-Block

Flip-Flop

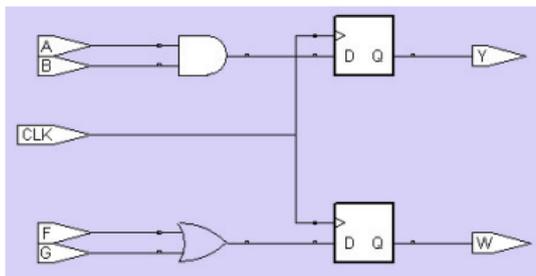
Getakteter **always**-Block



A. Koch

- Mit **posedge** oder **negedge** in der Aktivierungsliste
- Jedes **nicht-lokale** potenzielle Register wird Flip-Flop
- Vollständigkeit ist nun **irrelevant**.

```
always @(posedge CLK)  
begin  
  Y <= A & B;  
  W <= F | G;  
end
```



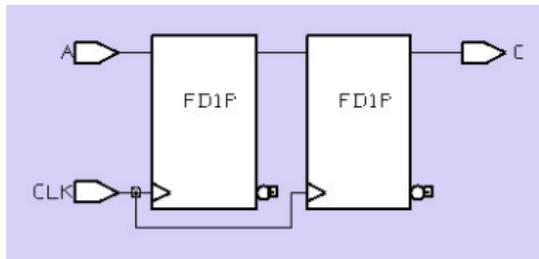
Flip-Flop

Getakteter **always**-Block



A. Koch

```
module lokal (  
  input wire CLK, A,  
  output reg C);  
  
reg B;  
  
always @ (posedge CLK)  
begin  
  B <= A;  
  C <= B;  
end  
  
endmodule
```



Für **B** entsteht ein Flip-Flop,
da B zwar räumlich, nicht
aber **zeitlich lokal** ist.

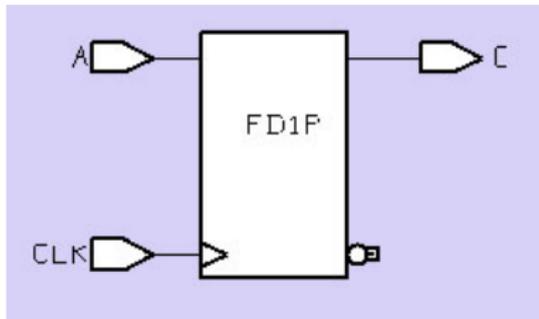
Flip-Flop

Getakteter **always**-Block



A. Koch

```
module lokal (  
  input wire CLK, A,  
  output reg C);  
  
reg B;  
  
always @ (posedge CLK)  
begin  
  B = A;  
  C <= B;  
end  
  
endmodule
```



Für **B** entsteht **kein** Flip-Flop,
da **B** räumlich und zeitlich
lokal ist.

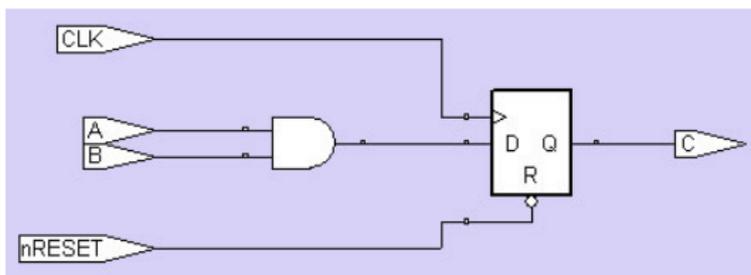
Flip-Flop mit Reset

Getakteter **always**-Block



A. Koch

```
module FF (  
  input wire CLK, nRESET, A, B,  
  output reg C);  
  
  always @( posedge CLK,  
           negedge nRESET)  
    if (!nRESET) C <= 0;  
    else          C <= A & B;  
  
endmodule
```



Flip-Flop mit
asynchronem
Reset



A. Koch

Zuweisungen

Richtlinien für Zuweisungen



A. Koch

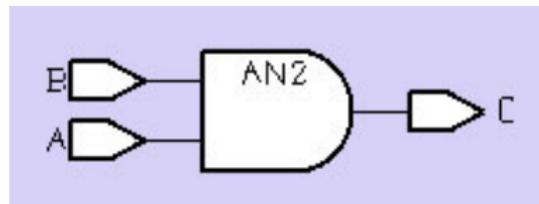
- Für spätere **Flip-Flops**
 - **nichtblockende** Zuweisung
- Für kombinatorische Logik, lokale Hilfsvariablen und Latches
 - **blockende** Zuweisung
- Für die Synthese aber **unerheblich**
 - Es gelten die Regeln von **Vollständigkeit** und **Lokalität**
- Trotzdem Richtlinien befolgen
 - Sonst Prä-Synthese und Post-Synthese Simulation nur **schwer** vergleichbar

So nicht: Nichtblockende Zuweisungen



A. Koch

```
always @ (A, B)
begin
  C <= 0;
  if (B) C <= A;
end
```



Trotz \leq **kombinatorische** Logik, da c vollständig beschrieben und lokal verwendet.

So nicht: Blockende Zuweisungen



A. Koch

```
always @ (posedge CLK)
```

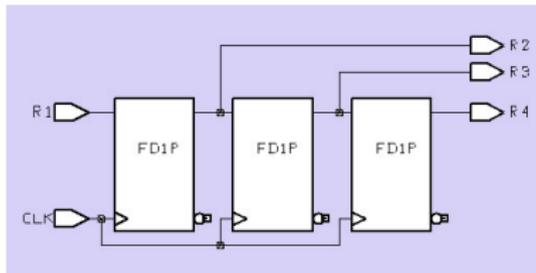
```
R2 = R1;
```

```
always @ (posedge CLK)
```

```
R3 = R2;
```

```
always @ (posedge CLK)
```

```
R4 = R3;
```



- In **Simulation**: **Zufällige** Ausführungsreihenfolge
 - R2=, R3=, R4 oder R4=, R2=, R3= ...
- In **Synthese**: Immer Schieberegister aus **Flip-Flops**
 - R2, R3, R4 nicht nur **lokal** verwendet



A. Koch

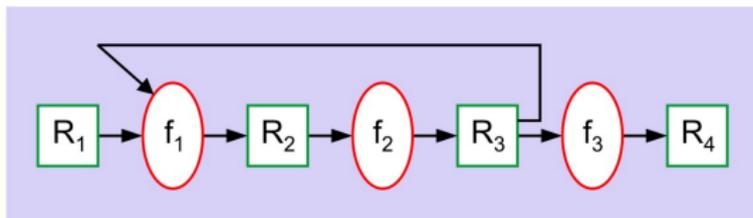
Kombinatorische Logik und Register

Trennung von kombinatorischer Logik und Registern



A. Koch

- Werden in RTL ja **getrennt** betrachtet
- Also auch im Verilog-Modell sauber trennen
- Beispiel
 - Diesmal **keine** Pipeline!
 - Teil eines digitalen Weckers
 - Benutzer soll Anzahl von Wecktönen wählen können (zwischen 1 und 100)



1. Möglichkeit: Getrennte `always`-Blöcke



A. Koch

```
module alarm_count (  
  input wire    CLOCK,           // Takt  
               RESET,           // Reset  
               INCALARM,        // =1 wenn Taste gedrückt  
  output reg [6:0] CURRENTCOUT; // jetziger Zustand (Alarmanzahl bis 100)  
  
  // interne Variable  
  reg [6:0] NEXTCOUNT;         // naechster Zustand  
  
  // kombinatorische Uebergangsfunktion, die bei jeder Aenderung  
  // des jetzigen Zustandes CURRENTCOUNT oder der Eingabe INCALARM  
  // den naechsten Zustand NEXTCOUNT vorlaeufig berechnet  
  
  always @(CURRENTCOUT, INCALARM) begin  
    NEXTCOUNT = CURRENTCOUT;  
    if (INCALARM == 1)    NEXTCOUNT = CURRENTCOUT+1;  
    if (NEXTCOUNT > 100) NEXTCOUNT = 100;  
  end  
  
  // naechsten Zustand mit jedem Takt endgueltig zum jetzigen machen;  
  // synchrones Reset  
  
  always @(posedge CLOCK)  
  if (RESET == 1)    CURRENTCOUT <= 0;  
  else                CURRENTCOUT <= NEXTCOUNT;  
  
endmodule
```

2. Möglichkeit: Logik als Funktion



A. Koch

```
module alarm_count (  
  input wire    CLOCK,           // Takt  
              RESET,            // Reset  
              INCALARM,         // =1 wenn Taste gedrückt  
  output reg [6:0] CURRENTCOUNT; // jetziger Zustand (Alarmanzahl bis 100)  
  
  // kombinatorische Uebergangsfunktion, die bei jeder Aenderung des jetzigen  
  // Zustandes CURRENTCOUNT oder der Eingabe INCALARM  
  // den naechsten Zustand NEXTCOUNT vorlaeufig berechnet  
  
  function [6:0] NEXTCOUNT(  
    input [6:0] CURRENTCOUNT,  
    input      INCALARM);  
  
  begin  
    NEXTCOUNT = CURRENTCOUNT;  
    if (INCALARM == 1)    NEXTCOUNT = CURRENTCOUNT+1;  
    if (NEXTCOUNT > 100) NEXTCOUNT = 100;  
  end  
endfunction  
  
  // naechsten Zustand mit jedem Takt endgueltig zum jetzigen machen;  
  // synchrones Reset  
  
  always @(posedge CLOCK)  
  if (RESET == 1) CURRENTCOUNT <= 0;  
  else            CURRENTCOUNT <= NEXTCOUNT (CURRENTCOUNT, INCALARM);  
  
endmodule
```

3. Möglichkeit: Keine saubere Trennung

Alles in einem **always**-Block



A. Koch

```
module alarm_count (  
  input wire    CLOCK,           // Takt  
              RESET,           // Reset  
              INCALARM,        // =1 wenn Taste gedrückt  
  output reg [6:0] CURRENTCOUT; // jetziger Zustand (Alarmanzahl bis 100)  
  
  // interne Variable  
  reg [6:0] NEXTCOUNT;         // naechster Zustand  
  
  // mit jedem Takt naechsten Zustand kombinatorisch in NEXT berechnen  
  // und dann in PRESENT zum jetzigen machen;  
  
  always @(posedge CLOCK) begin  
    NEXTCOUNT = CURRENTCOUT;  
    if (INCALARM == 1)    NEXTCOUNT = CURRENTCOUT+1;  
    if (NEXTCOUNT > 100) NEXTCOUNT = 100;  
  
    if (RESET == 1)      CURRENTCOUT <= 0;  
    else                  CURRENTCOUT <= NEXTCOUNT;  
  end  
  
endmodule
```

- Kürzer, aber nicht mehr sauber getrennt
- Nachteil: Wird bei Erweiterung leicht unübersichtlich
- In der Industrie verpönt, dort i.d.R. strikte Trennung



A. Koch

Zusammenfassung: Potenzielle Register



- **getakteter always-Block**: `always @(posedge CLK)...`
- **flankenfreier always-Block**: `always @(CLK, D)...`
- **PR** ist **potenzielles Register**, falls **PR** in einem `always`-Block **geschrieben** wird
- **PR** ist **vollständig**, wenn es in **jedem** Durchlauf eines `always`-Blockes geschrieben wird
- **PR** ist **räumlich lokal**, wenn es nur **innerhalb** eines `always`-Blockes auftritt
- Ein räumlich lokales **PR** ist **zeitlich lokal** (kurz: **lokal**), falls es nie **vor** dem Schreiben gelesen wird

Zusammenfassung: Getaktete `always`-Blöcke



A. Koch

- Jedes **nicht-lokale** PR wird ein getaktetes **Flip-Flop**
- An solche Flip-Flops wird mit `<=` zugewiesen
- An kombinatorische Hilfsvariablen mit `=`
- Spätere Flip-Flops werden an nur **einer** Stelle geschrieben
 - Ausgenommen der Reset, der steht extra
- Jedes Flip-Flop bekommt bei Reset einen definierten Wert

Zusammenfassung: Flankenfreie `always`-Blöcke



A. Koch

- Ein vollständiges **oder** lokales PR wird **kombinatorische** Logik
- Ein nicht-lokales **und** unvollständiges PR wird ein **Latch**
- Es wird stets **blockend** mit `=` zugewiesen
- Die Aktivierungsliste enthält alle **Lesevariablen** des `always`-Blockes
- Ein Takt in der Aktivierungsliste wird in der Synthese wie jede Variable auch behandelt



A. Koch

`for`-Anweisung

Synthese von `for`-Anweisungen



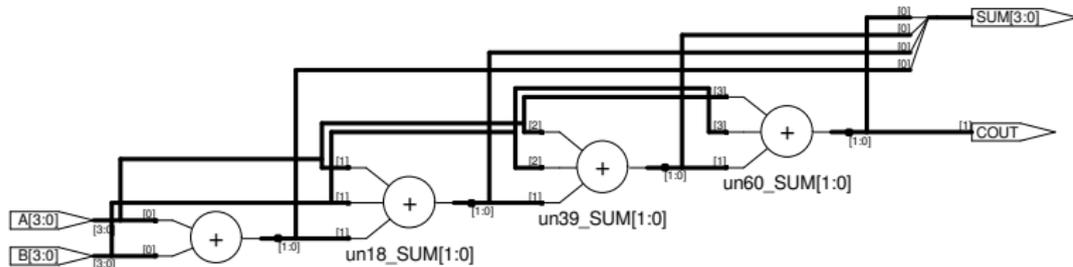
- **Nicht** als sequentielle Schleife
 - Wie in normaler Programmiersprache
- Stattdessen: **Räumlich** "ausgerollt"
 - Parallele Abarbeitung

```
module unrolled_for (input      [3:0] A, B,
                    output reg [3:0] SUM,
                    output reg      COUT);

integer I;
reg C;

always @(*) begin
  C = 0;
  for (I = 0; I < 4; I = I + 1) begin
    {C, SUM[I]} = A[I] + B[I] + C;
  end
  COUT = C;
end
```

A. Koch





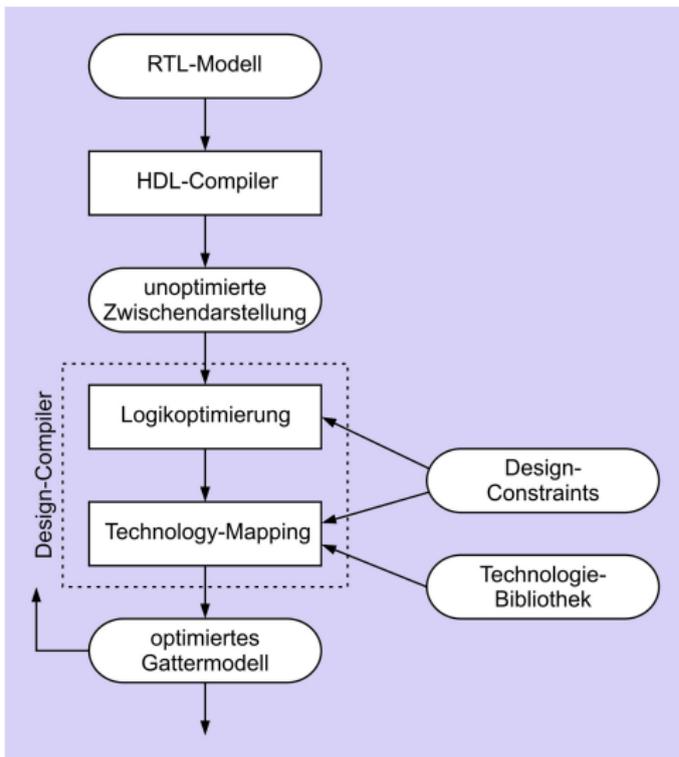
A. Koch

Verfeinerter Ablauf der Synthese

Verfeinerter Entwurfsablauf der Synthese



A. Koch

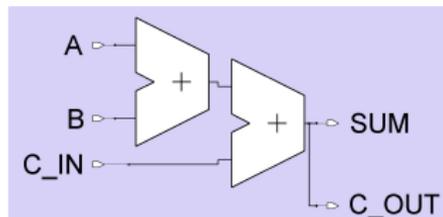


Syntheseablauf: Technologieabbildung

technology mapping

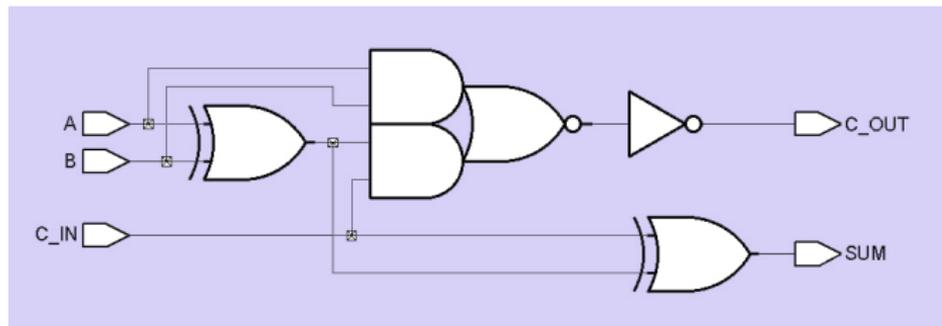


Unoptimierte Zwischendarstellung eines 1b-Addierers



A. Koch

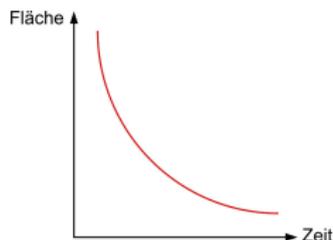
Ergebnis für ASIC-Zielbibliothek (LSI 10K)



Design-Constraints



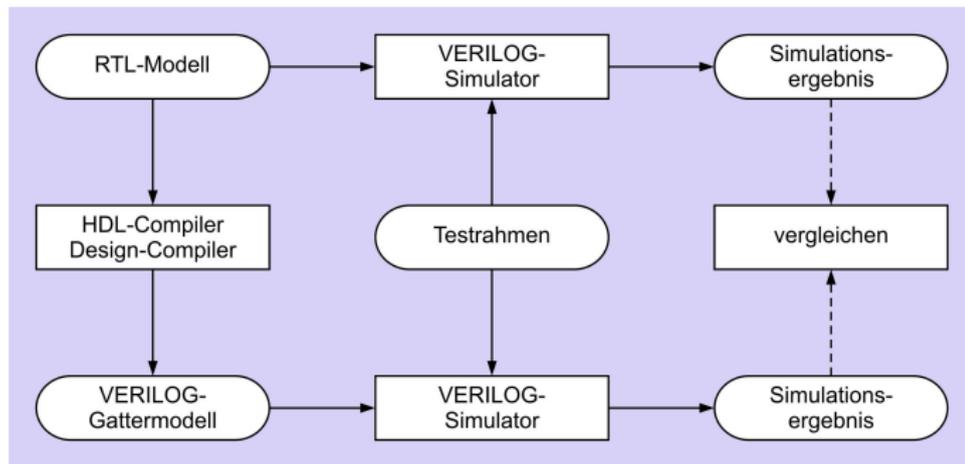
- Zeit
 - Timing-Analyse
 - Geschätzt nach Synthese (ohne Verdrahtungsverzögerung)
 - Exakt nach Platzieren und Verdrahten
 - ➔ Layout-Ebene
- Fläche
 - Geschätzt nach Synthese (ohne Verdrahtungsfläche!)
 - Exakt nach Platzieren und Verdrahten
- Elektrische Leistungsaufnahme
 - Simulation auf Layout-Ebene
 - Bestimmung von Umschaltfrequenzen



A. Koch

Verifikation

Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



A. Koch

- Prä-Synthese ./ Post-Synthese-Simulation
- Gleiche Testdaten
 - Bei Post-Synthese aber genauere Tests möglich
 - Hier nun **genauer**es Zeitverhalten
- **Differenzen** in Ergebnissen durch anderes Zeitverhalten
- Vergleich etwas aufwendiger

Beispiel: 4b-Vergleicher



- Größenvergleich von zwei 4b breiten Eingabewerten **A** und **B**
- Bestimmt Flags für $<$, $>$, und $=$
- Erstes Ziel: Möglichst schnelle Schaltung, Fläche egal

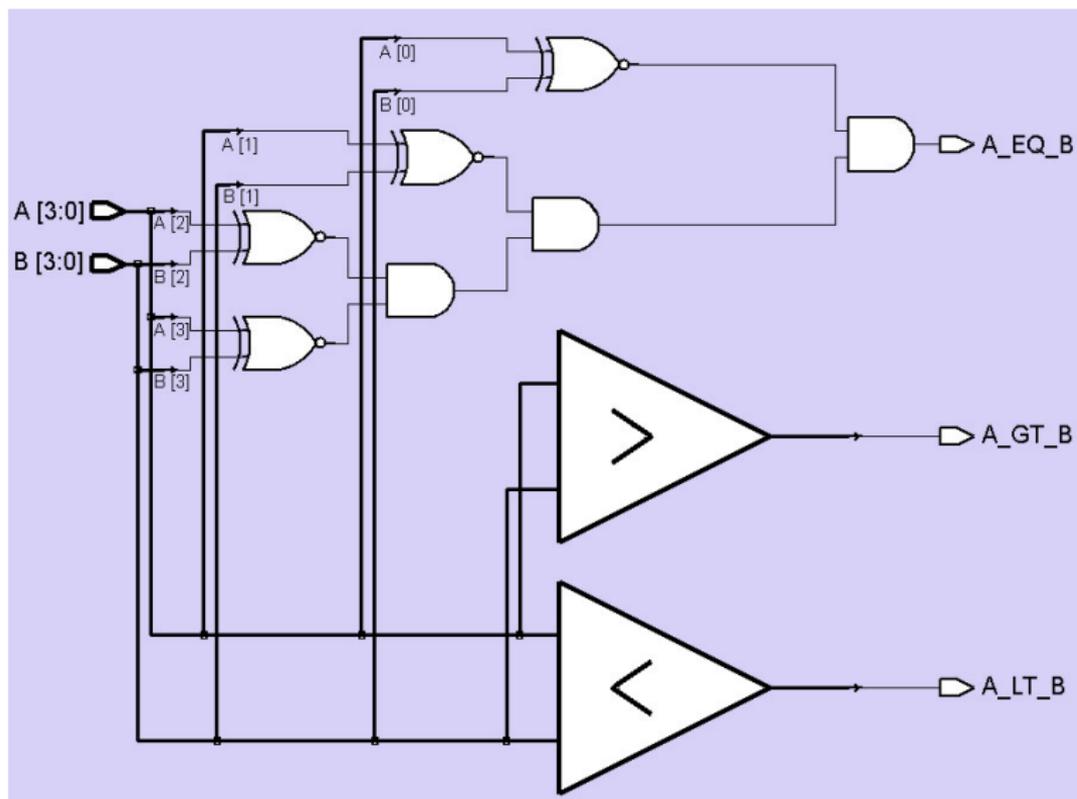
A. Koch

```
// Vergleich
module mag_comp (
  input wire [3:0] A, B,
  output wire    A_GT_B, A_LT_B, A_EQ_B);

assign A_GT_B = (A > B);
// A groesser B
assign A_LT_B = (A < B);
// A kleiner B
assign A_EQ_B = (A == B);
// A gleich B

endmodule
```

Zwischendarstellung des 4b-Vergleichers



A. Koch

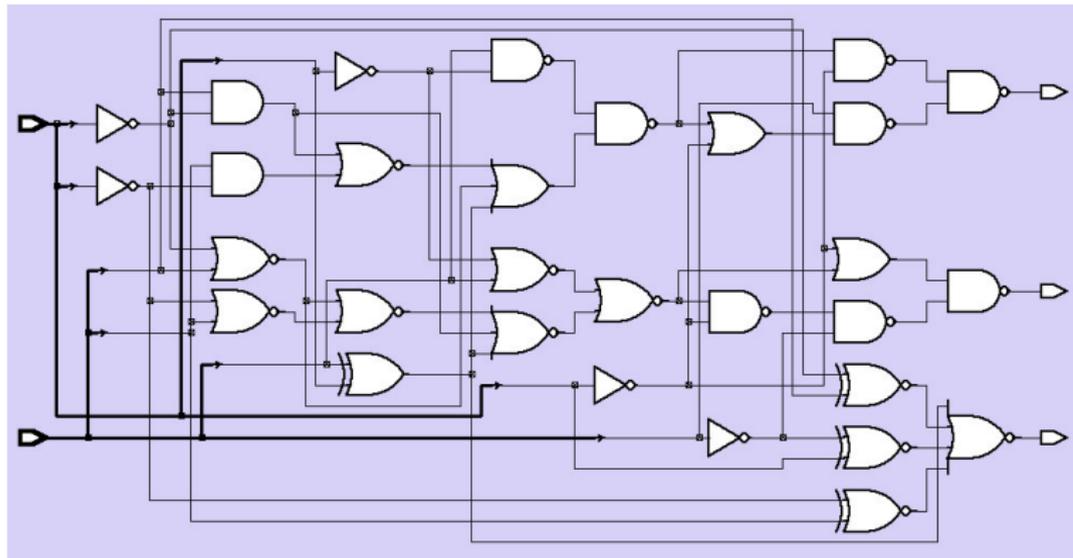
4b-Vergleicher in LSI Logic 10K

ASIC-Technologie

Benutzt um 1993



A. Koch

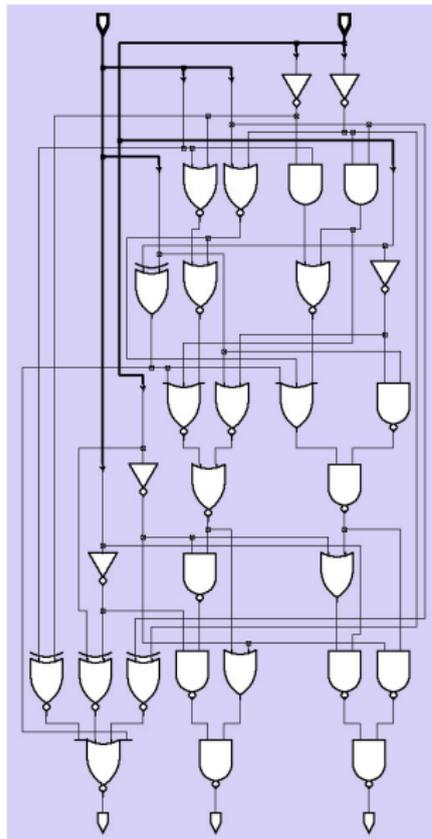


Gatternetzliste des 4b-Vergleichers

Als **strukturelles** Verilog



```
module mag_comp_gate (  
  input wire [3:0] A,  
           B,  
  output wire A_GT_B, A_LT_B, A_EQ_B);  
  
  wire  
  N107, N108, N109, N110, N111, N112, N113, N114, N115,  
  N116, N117, N118, N119, N120, N121, N122, N123, N124,  
  N125, N126, N127, N128, N129, N130, N131, N132, N133;  
  
  nr4 U89 (.A(N107), .B(N108), .C(N109), .D(N110), .Z(A_EQ_B));  
  nd2 U90 (.A(N111), .B(N112), .Z(A_GT_B));  
  nd2 U91 (.A(N113), .B(N114), .Z(A_LT_B));  
  iv U92 (.A(B[0]), .Z(N115));  
  iv U93 (.A(B[1]), .Z(N116));  
  iv U94 (.A(B[2]), .Z(N117));  
  nr2 U95 (.A(N119), .B(N120), .Z(N118));  
  iv U96 (.A(B[3]), .Z(N121));  
  nd2 U97 (.A(N123), .B(N124), .Z(N122));  
  iv U98 (.A(A[3]), .Z(N125));  
  en U99 (.A(N116), .B(A[1]), .Z(N108));  
  en U100 (.A(N125), .B(B[3]), .Z(N109));  
  en U101 (.A(N115), .B(A[0]), .Z(N110));  
  an2 U102 (.A(A[1]), .B(N116), .Z(N126));  
  nr2 U103 (.A(N116), .B(A[1]), .Z(N127));  
  nr2 U104 (.A(N115), .B(A[0]), .Z(N128));  
  nr2 U105 (.A(N127), .B(N128), .Z(N129));  
  nr3 U106 (.A(N129), .B(N126), .C(N107), .Z(N120));  
  nr2 U107 (.A(N117), .B(A[2]), .Z(N119));  
  nd2 U108 (.A(N118), .B(N121), .Z(N130));  
  nd2 U109 (.A(N130), .B(N125), .Z(N114));  
  or2 U110 (.A(N121), .B(N118), .Z(N113));  
  an2 U111 (.A(A[0]), .B(N115), .Z(N131));  
  nr2 U112 (.A(N126), .B(N131), .Z(N132));  
  or3 U113 (.A(N132), .B(N127), .C(N107), .Z(N124));  
  nd2 U114 (.A(A[2]), .B(N117), .Z(N123));  
  or2 U115 (.A(N122), .B(N121), .Z(N133));  
  nd2 U116 (.A(A[3]), .B(N133), .Z(N112));  
  nd2 U117 (.A(N122), .B(N121), .Z(N111));  
  eo U118 (.A(A[2]), .B(B[2]), .Z(N107));  
  
endmodule
```



A. Koch

Schlüssellochblick in Technologie-Bibliothek

an2: AND-Gatter mit zwei Eingängen



A. Koch

```
// 2-Input-AND-Gatter
```

```
// physikalische Zeiteinheit / Simulationsschrittweite
```

```
'timescale 100 ps / 10 ps
```

```
'celldefine
```

```
module an2 (Z, A, B);
```

```
    output Z;
```

```
    input  A, B;
```

```
// Instanz einer VERILOG-Primitive (in KCMS nicht weiter behandelt!)
```

```
and And1 (Z, A, B);
```

```
// Verzögerungszeiten fuer steigende und fallende Flanken
```

```
specify
```

```
    (A *> Z) = (1, 1);
```

```
    (B *> Z) = (1, 1);
```

```
endspecify
```

```
endmodule
```

```
'endcelldefine
```

Test des 4b-Vergleichers: Prä-Synthese



Testrahmen

A. Koch

```
module stimulus;  
  
reg [3:0] A, B;  
wire A_GT_B, A_LT_B, A_EQ_B;  
  
// Instanz des Vergleichers  
mag_comp Mag_comp (A, B, A_GT_B, A_LT_B, A_EQ_B);  
  
initial  
  $monitor ($time,  
    "_A=%d,_B=%d,_A_GT_B=%b,_A_LT_B=%b,_A_EQ_B=%b",  
    A, B, A_GT_B, A_LT_B, A_EQ_B);  
  
// Testmuster  
initial begin  
  A = 10; B = 9;  
  #10 A = 14; B = 15;  
  #10 A = 0; B = 0;  
  #10 A = 8; B = 12;  
  #10 A = 6; B = 14;  
  #10 A = 14; B = 14;  
end  
  
endmodule
```

```
0 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0  
10 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0  
20 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1  
30 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0  
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0  
50 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1
```

Prä-Synthese-Ergebnis

Vergleich: Prä-Synthese mit Post-Synthese

Gleiche Stimuli wie oben



A. Koch

Prä-Synthese

0 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1

Post-Synthese

0 A=10, B= 9, A_GT_B=x, A_LT_B=x, A_EQ_B=x
3 A=10, B= 9, A_GT_B=x, A_LT_B=x, A_EQ_B=0
6 A=10, B= 9, A_GT_B=x, A_LT_B=0, A_EQ_B=0
8 A=10, B= 9, A_GT_B=1, A_LT_B=0, A_EQ_B=0
10 A=14, B=15, A_GT_B=1, A_LT_B=0, A_EQ_B=0
16 A=14, B=15, A_GT_B=1, A_LT_B=1, A_EQ_B=0
18 A=14, B=15, A_GT_B=0, A_LT_B=1, A_EQ_B=0
20 A= 0, B= 0, A_GT_B=0, A_LT_B=1, A_EQ_B=0
23 A= 0, B= 0, A_GT_B=0, A_LT_B=1, A_EQ_B=1
28 A= 0, B= 0, A_GT_B=0, A_LT_B=0, A_EQ_B=1
30 A= 8, B=12, A_GT_B=0, A_LT_B=0, A_EQ_B=1
32 A= 8, B=12, A_GT_B=1, A_LT_B=0, A_EQ_B=0
34 A= 8, B=12, A_GT_B=0, A_LT_B=0, A_EQ_B=0
35 A= 8, B=12, A_GT_B=0, A_LT_B=1, A_EQ_B=0
40 A= 6, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
50 A=14, B=14, A_GT_B=0, A_LT_B=1, A_EQ_B=0
53 A=14, B=14, A_GT_B=0, A_LT_B=0, A_EQ_B=1

Annahme: Jedes Gatter hat
1 Zeiteinheit **Verzögerung**

Diskussion: Prä- ./ Post-Synthese-Simulation

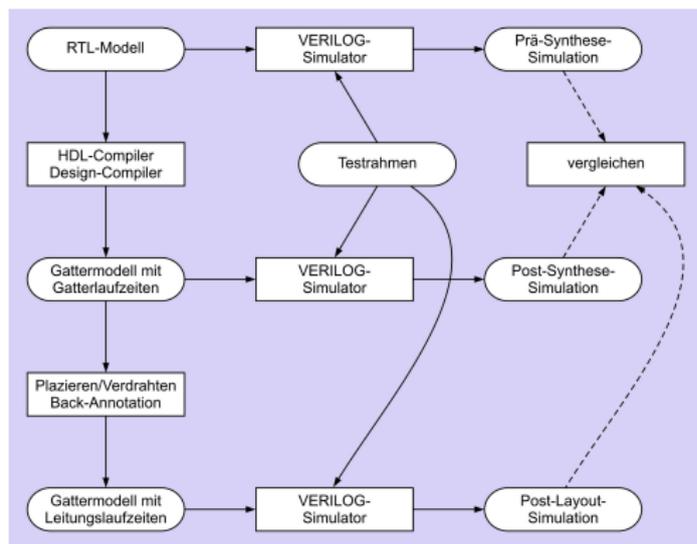


A. Koch

- Unterschiedlich **viele** Ergebnisse
- Verschiedene **Werte**
- Unterschiedliche **Zeiten**
 - Vorher **gar keine** ausser den im Testrahmen
- Manche Ergebnisse schlicht **falsch** (z.B. bei $t=32$)
- Interpretation nötig
“Wenn man lange genug wartet, ist das Ergebnis richtig”!
- Was ist **“lange genug”**?
- Antwort: **Kritischer Pfad** (TGD12)
- Damit passender Takt für RTL wählbar **zwischen**
 - Eingangsregistern
 - Ausgangsregistern

Weitere Verfeinerung der Verifikation

Nun bis auf Layout-Ebene



A. Koch

- Post-Layout-Simulation schliesst ein
 - Gatterverzögerungen
 - Leitungsverzögerung
 - Kann umfassen: Widerstände, Kapazitäten, Induktivitäten



A. Koch

Synthesebeispiel: Zero-Counter

Synthesebeispiel: Zero-Counter



A. Koch

- Spezifikation
 - **Eingabe** ist ein 8b Datenwort
 - **Ausgabe** soll sein die Anzahl der Null-Bits in der Eingabe
- Genauer betrachtet
 - RTL-Modell kann Synthese-Ergebnis **direkt** beeinflussen
 - Wirkung von **Design-Constraints**
- Nicht mehr so relevant
 - **Konkrete** Umsetzung in Gatter-Modell
 - Bei größeren Schaltungen oft schlicht zu **unübersichtlich**

Zero-Counter: Vereinfachte Lösung



```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);
```

```
  integer    I;
```

```
  always @(IN) begin
```

```
    OUT = ^IN[0];
```

```
    for (I=1; I<8; I=I+1)
```

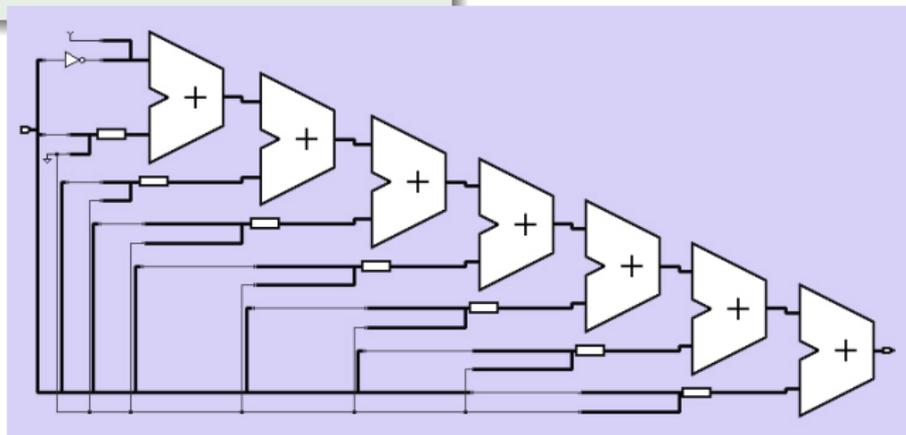
```
      OUT = OUT + ^IN[I];
```

```
  end
```

```
endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer entfallen, Bits werden direkt aufaddiert

A. Koch



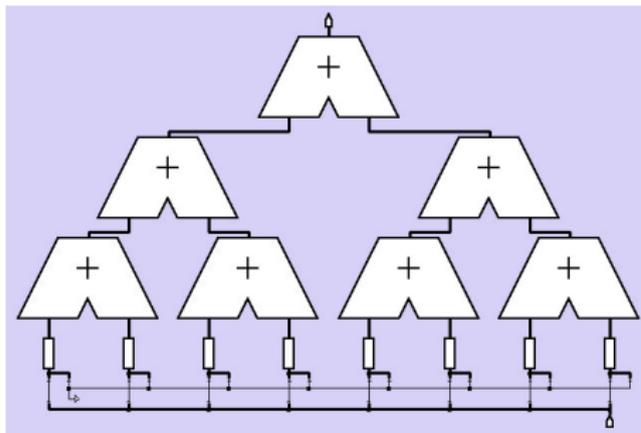
Zero-Counter: Schlaue Lösung



```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);  
  
  always @(IN)  
    OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))  
          + ((~IN[4]+~IN[5]) + (~IN[6]+~IN[7]));  
  
endmodule
```

- Bits werden direkt aufaddiert
- Jetzt aber **hierarchische** Klammerung
- Damit **parallele** Berechnung
- Addierer-**Baum**

A. Koch





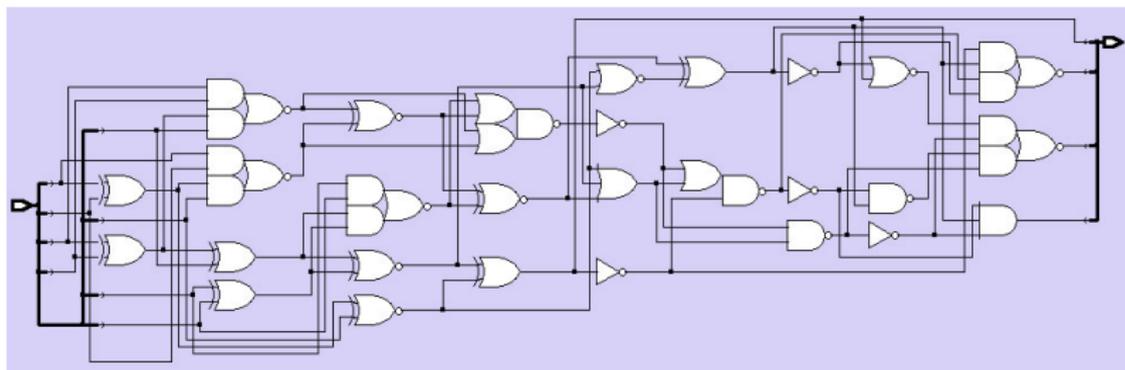
- Festlegen unterschiedlicher **Optimierungsziele**
- Üblich
 - Flächenbedarf
 - Geschwindigkeit (niedrige Verzögerung)
- Noch seltener
 - Energieverbrauch
 - Ausfallsicherheit

Optimierung auf Fläche

8b-Zero-Counter



A. Koch



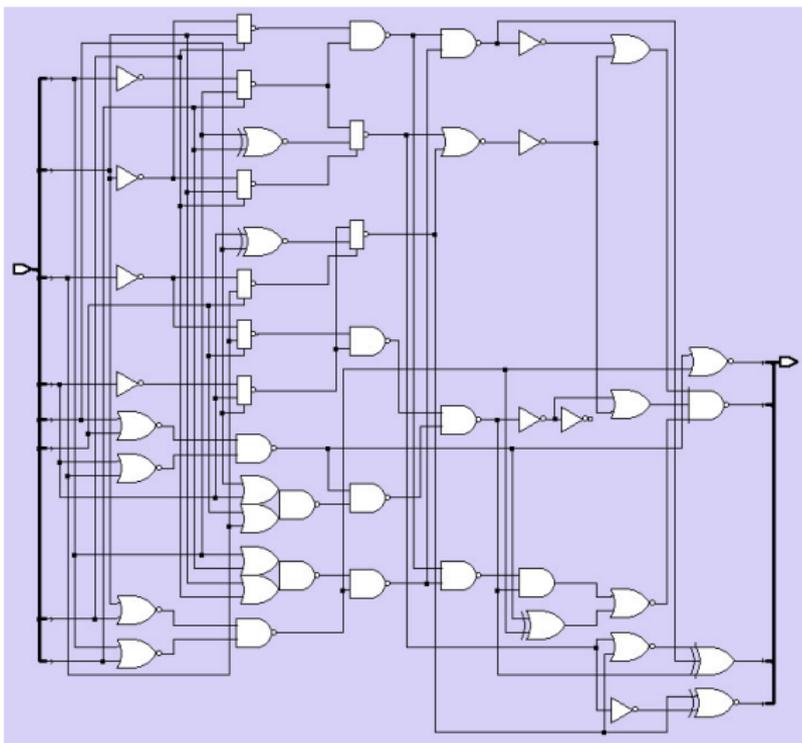
52 Gatter, **17.8ns** Verzögerung

Optimierung auf Geschwindigkeit

8b-Zero-Counter



A. Koch



99 Gatter, **8.2ns** Verzögerung

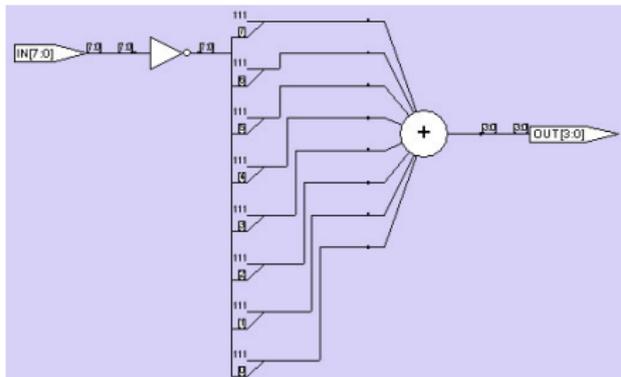
Zero-Counter: Noch bessere Lösung



```
module count(  
  input wire [7:0] IN,  
  output reg [3:0] OUT);  
  
always @(IN)  
  OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))  
        + ((~IN[4]+~IN[5]) + (~IN[6]+~IN[7]));  
  
endmodule
```

- Schlaues Synthesewerkzeug
- Erkennt **Natur** der Berechnung
- Addiert alle Bits **gleichzeitig** mit einem 8b-Addierer

A. Koch





A. Koch

Synthesebeispiel: Schaltwerk

Entwurfsbeispiel: Verkaufsautomat



A. Koch

- Bisher **kombinatorische** Beispiele
 - 4b-Vergleicher
 - 8b-Zero-Counter
- Nun **sequentielle** Logik
 - Mit **Zustandsgedächtnis** und **Takt**
 - Schaltwerk

Spezifikation des Verkaufsautomaten



A. Koch

- Hier: Entwurf der **Steuerung**
- Eingabe: Münzen im Wert von 1 Euro und 50 Cent
- Ausgabe: Ein Eis für **1,50 EUR**
- Eigentlicher Verdienst
 - Überzahlung möglich, aber **kein** Wechselgeld
- Reihenfolge von 50 Cent und 1 Euro beliebig
- Vereinfachung hier: Kurze Pause zwischen zwei Münzeinwürfen

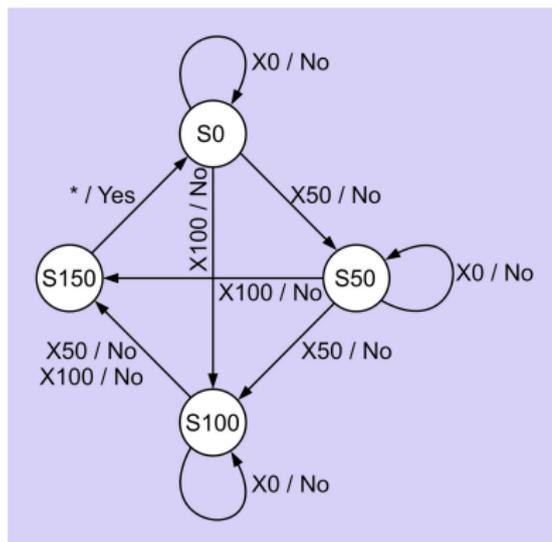
Grobentwurf der Steuerung des Verkaufsautomaten



- Eingangssignal **COIN** aus zwei Bits
 - Kann symbolische Werte **X0**, **X50**, **X100** annehmen
- Ausgabesignal **ICE** steuert Eisausgabe
 - Symbolische Werte **Yes** und **No**
- Zustände **S0**, **S50**, **S100**, **S150**

A. Koch

➔ Zustandsautomat
in **Mealy-Form**



RTL-Modell: Modulkopf

Verwaltungskram, hier passiert noch nichts



```
// Verkaufsautomat
module iglu (
  input wire      CLOCK, // Takt
                   RESET, // Reset
  input wire [1:0] COIN,  // eingeworfene Muenze
  output reg     ICE);   // Warenausgabe

// interne Variable
reg [1:0] PRESENT,      // jetziger Zustand
        NEXT;           // naechster Zustand

// Zustandscodierung
parameter S0  = 2'b00, // bisher nichts eingeworfen
           S50 = 2'b01, // bisher 50 Cent
           S100 = 2'b10, // bisher 100 Cent
           S150 = 2'b11; // bisher 150 Cent oder 200

// Eingabecodierung
parameter X0  = 2'b00, // kein Einwurf
           X50 = 2'b01, // Fuenfziger
           X100 = 2'b10; // Euro

// Ausgabecodierung
parameter Yes = 1'b1, // Warenausgabe
           No  = 1'b0; // keine Ausgabe
```

A. Koch

RTL-Modell: Kombinatorische Logik



```
always @(PRESENT, COIN) // <-- flankenfrei!
  case (PRESENT)

S0: // bisher nichts eingeworfen
  if (COIN == X0) // kein Einwurf
    {NEXT,ICE} = {S0, No};
  else if (COIN == X50) // Fuenfziger
    {NEXT,ICE} = {S50, No};
  else // Euro
    {NEXT,ICE} = {S100,No};

S50: // bisher 50 Cent
  if (COIN == X0) // kein Einwurf
    {NEXT,ICE} = {S50, No};
  else if (COIN == X50) // Fuenfziger
    {NEXT,ICE} = {S100,No};
  else // Euro
    {NEXT,ICE} = {S150,No};

S100: // bisher 100 Cent
  if (COIN == X0) // kein Einwurf
    {NEXT,ICE} = {S100,No};
  else // Fuenfziger oder Euro
    {NEXT,ICE} = {S150,No}; // Ueberzahlung wird dankend
    // ignoriert

S150: // genug bezahlt
  {NEXT,ICE} = {S0, Yes}; // Warenausgabe

endcase
```

Berechne aus dem aktuellen Zustand **PRESENT** und eventuell eingeworfenen Münzen **COIN** **vorläufig** den nächsten Zustand **NEXT** und die Ausgabe **ICE**.

Beachte:
Blockende
Zuweisungen

A. Koch

RTL-Modell: Zustandsspeicher



A. Koch

```
// naechsten Zustand mit jedem Takt endgueltig zum jetzigen machen;  
// synchrones Reset  
//  
always @(posedge CLOCK)           // <-- flankengesteuert!  
  if (RESET == 1'b1)  PRESENT <= S0; // Anfangszustand  
  else                PRESENT <= NEXT;  
  
endmodule // iglu
```

Testrahmen für Verkaufsautomat



```
// Instanz des Verkaufsautomaten
iglu Iglu (CLOCK, RESET, COIN, ICE);
// Takt
always
#20 CLOCK = ~CLOCK;
// Ergebnis drucken
initial begin
  $display (".....Zeit_Reset_Eisausgabe\n");
  $monitor ("%d.....%d.....%d", $time, RESET, ICE);
end
// Stimuli anlegen: Muenzen eingeben
initial begin
  CLOCK = 0; COIN = X0; RESET = 1'b1;
  #50 RESET = 1'b0;
  @(negedge CLOCK);

  // drei Fuenfziger
  #80 $display ("Einwurf_...50_Ct"); COIN = X50; #40 COIN = X0;
  #80 $display ("Einwurf_...50_Ct"); COIN = X50; #40 COIN = X0;
  #80 $display ("Einwurf_...50_Ct"); COIN = X50; #40 COIN = X0;
  // einen Fuenfziger, dann einen Euro
  #160 $display ("Einwurf_...50_Ct"); COIN = X50; #40 COIN = X0;
  #80 $display ("Einwurf_100_Ct"); COIN = X100; #40 COIN = X0;
  // zwei Euro (keine Rueckgabe!)
  #160 $display ("Einwurf_100_Ct"); COIN = X100; #40 COIN = X0;
  #80 $display ("Einwurf_100_Ct"); COIN = X100; #40 COIN = X0;
  // einen Euro, dann einen Fuenfziger
  #160 $display ("Einwurf_100_Ct"); COIN = X100; #40 COIN = X0;
  #80 $display ("Einwurf_...50_Ct"); COIN = X50; #40 COIN = X0;

  #80 $finish;
end
```

A. Koch

Simulationsergebnis



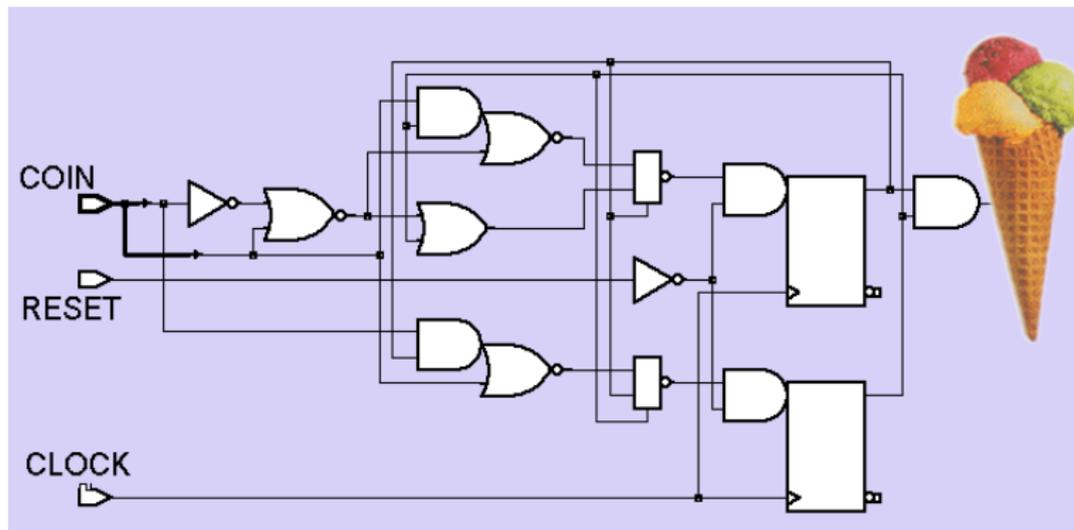
A. Koch

			Zeit	Reset	Eisausgabe
			0	1	x
			20	1	0
			50	0	0
Einwurf	50 Ct				
Einwurf	50 Ct				
Einwurf	50 Ct				
			420	0	1
			460	0	0
Einwurf	50 Ct				
Einwurf	100 Ct				
			740	0	1
			780	0	0
Einwurf	100 Ct				
Einwurf	100 Ct				
			1060	0	1
			1100	0	0
Einwurf	100 Ct				
Einwurf	50 Ct				
			1380	0	1
			1420	0	0

Syntheseergebnis



A. Koch





- RTL-Modellierungsstil **beeinflusst** Gattermodell
 - effizient oder schlecht, im schlimmsten Fall falsch
 - Designer muss eingesetzten Stil genau **beobachten**
- Zielkonflikt
 - **Abstraktes** RTL-Modell
 - **angenehm**, Synthesewerkzeug soll sich um Details kümmern
 - Produziert aber gelegentlich **unerfreuliche** Hardware
 - **Hardware-nahes** RTL-Modell
 - **Mühsam**, ineffizient zu schreiben
 - gute **Kontrolle** über spätere Schaltungsstruktur
 - Unabhängigkeit von Zieltechnologie kann **verloren** gehen
 - Insbesondere bei direkter Instanziierung von **Spezialblöcken**