Einführung in Computer Microsystems

4. Einführung in die Logiksynthese

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2007

Einführung

Logiksynthese

- 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

Logiksynthese

- 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

Logiksynthese

- 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

Logiksynthese

- 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

Logiksynthese

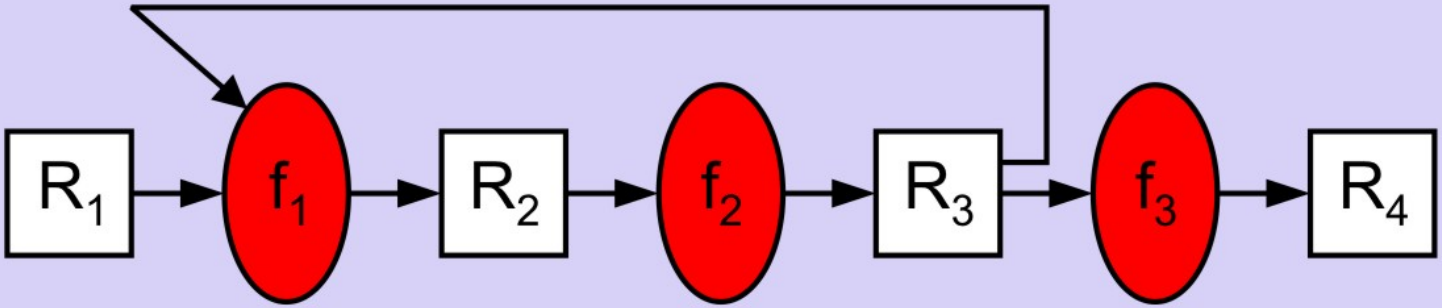
- Abbildung von RTL-Modell auf Gattermodell
  - Register-Transfer-Ebene auf Logikebene
- Wichtige Hersteller von Entwurfswerkzeugen
  - Für ASICs: Synopsys
  - Für FPGAs: Synplicity
  - Gibt aber auch noch diverse andere Anbieter

Logiksynthese

- 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

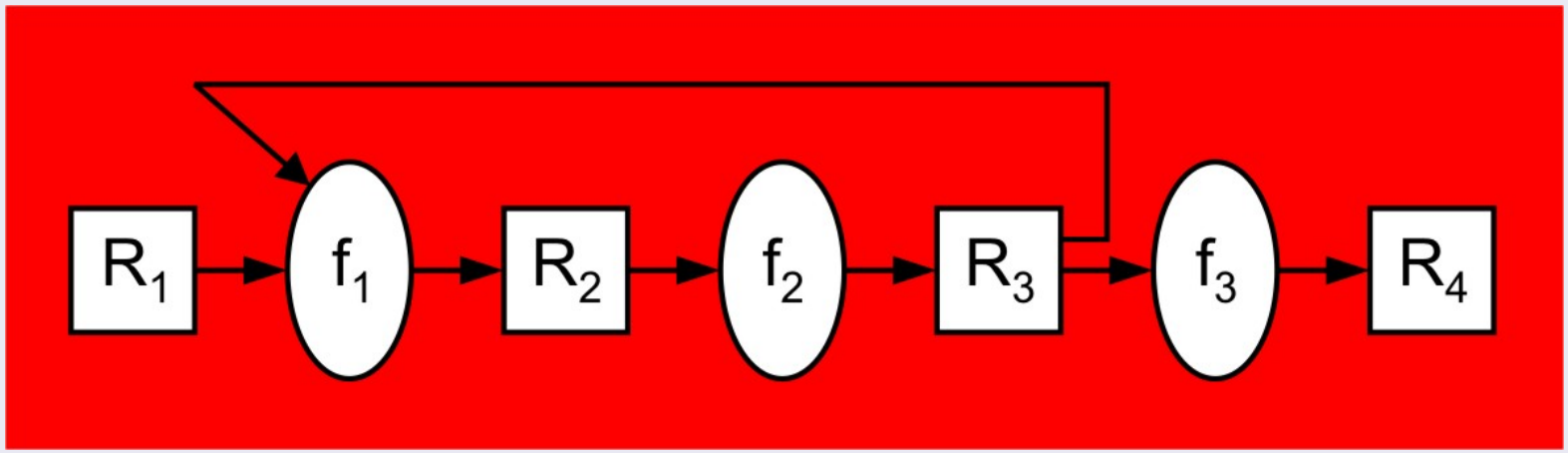
Logiksynthese

Optimiert im wesentlichen Logik zwischen getakteten Registern



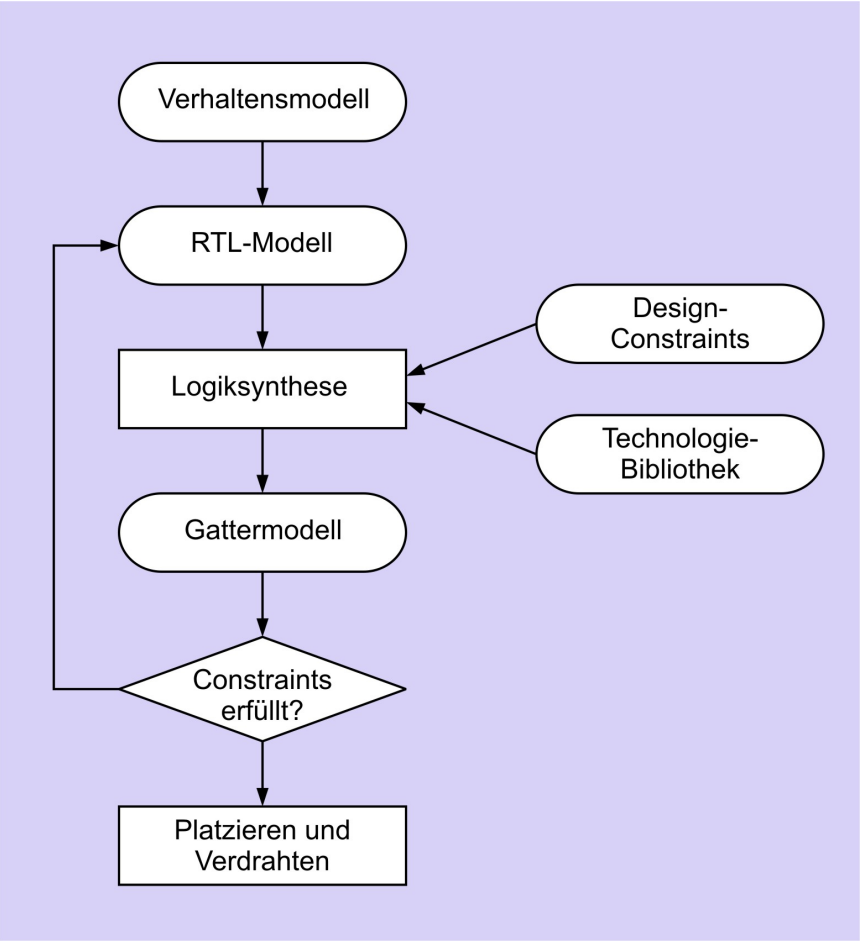
Vergleich: High-Level-Synthese

Beginnt oberhalb von RTL und optimiert über Taktgrenzen hinweg



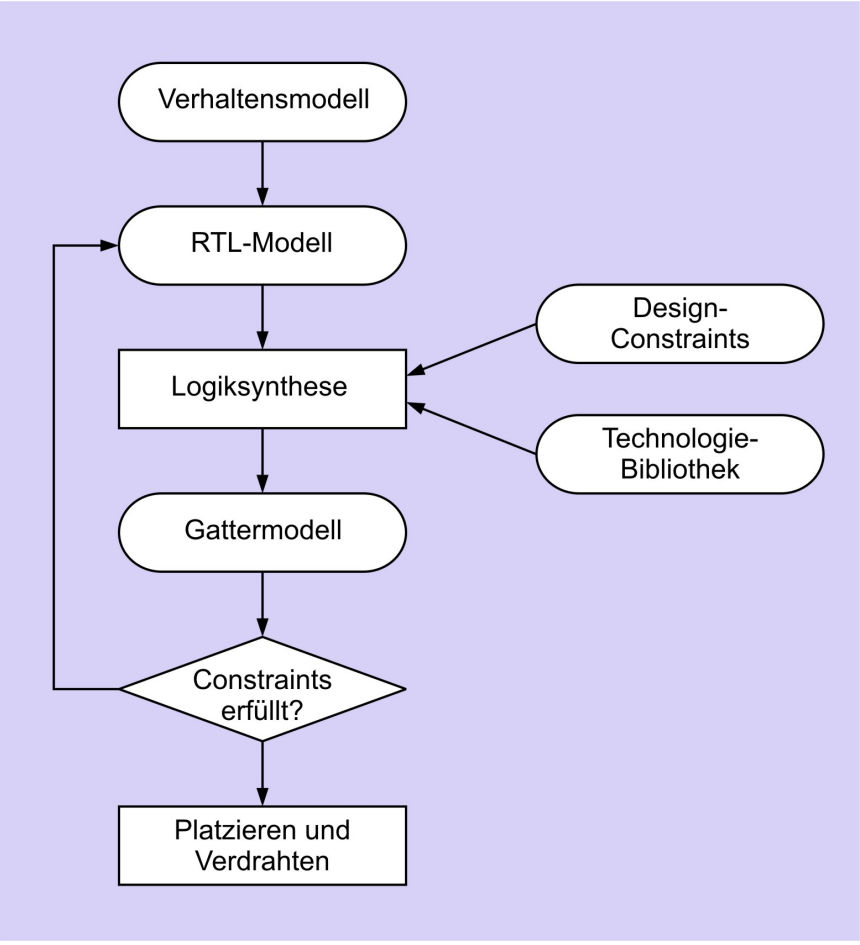
Noch experimentell, nur eingeschränkte praktische Bedeutung

Ablauf der Logiksynthese



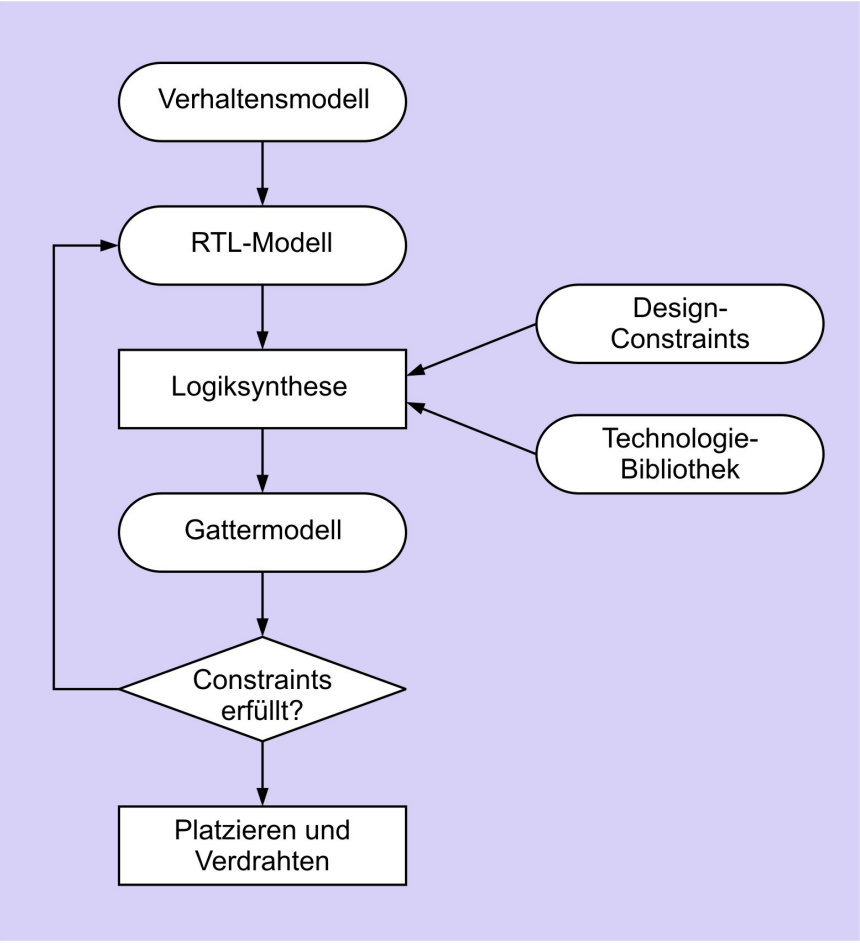
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



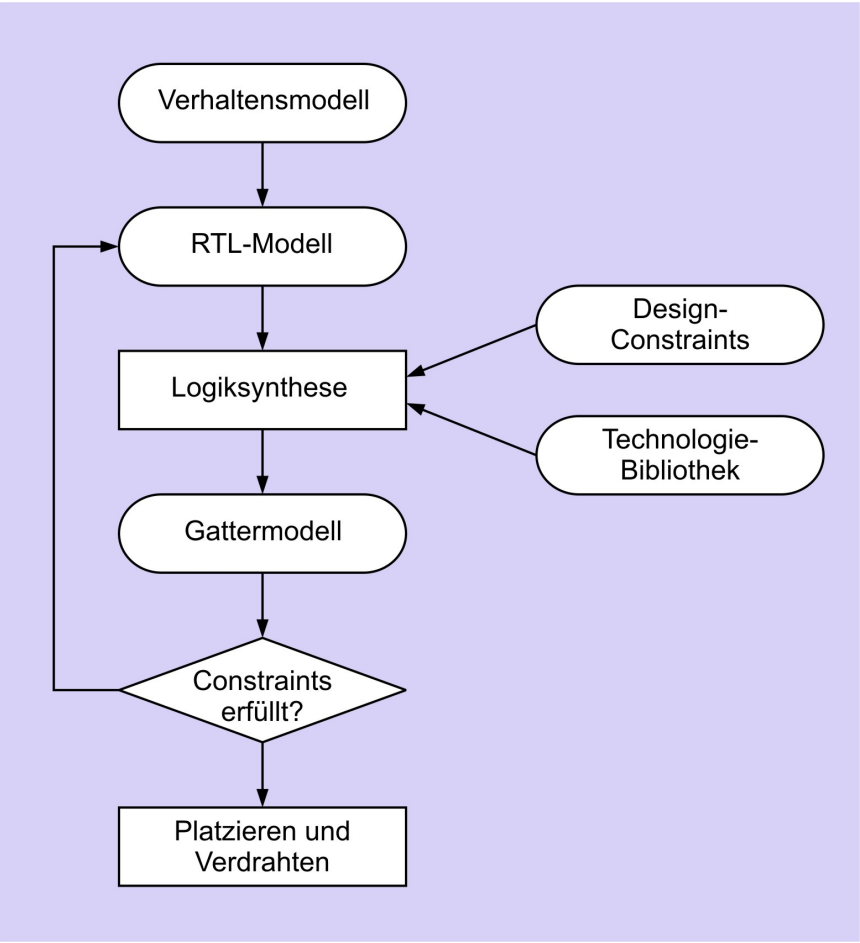
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



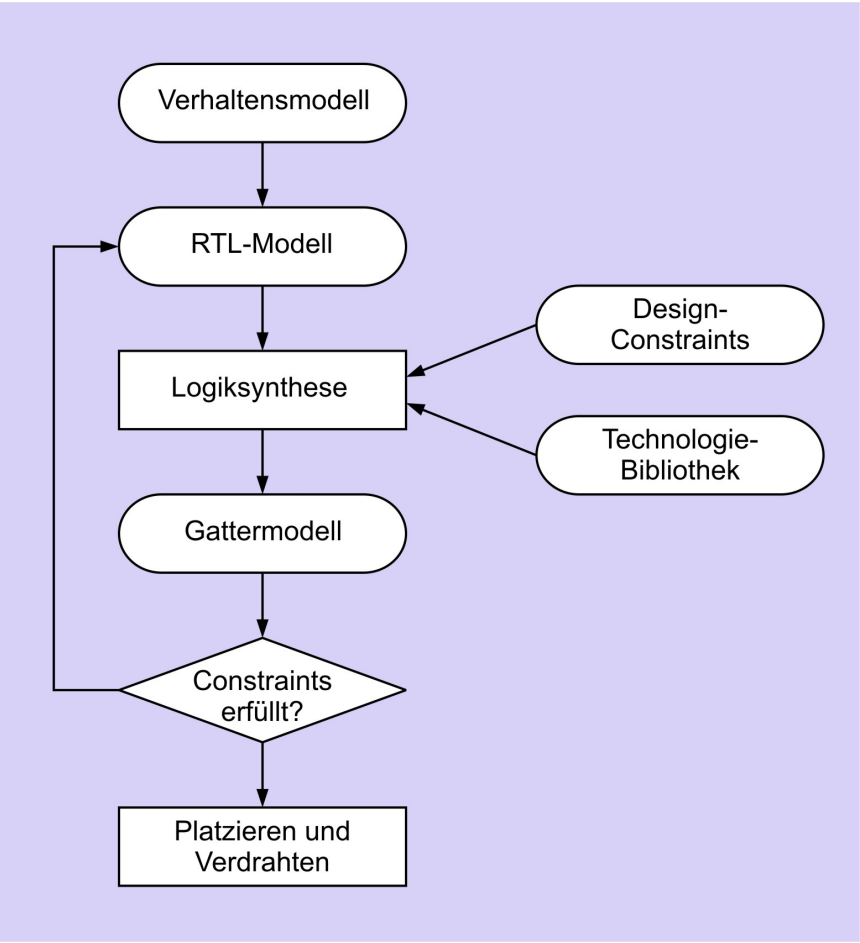
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



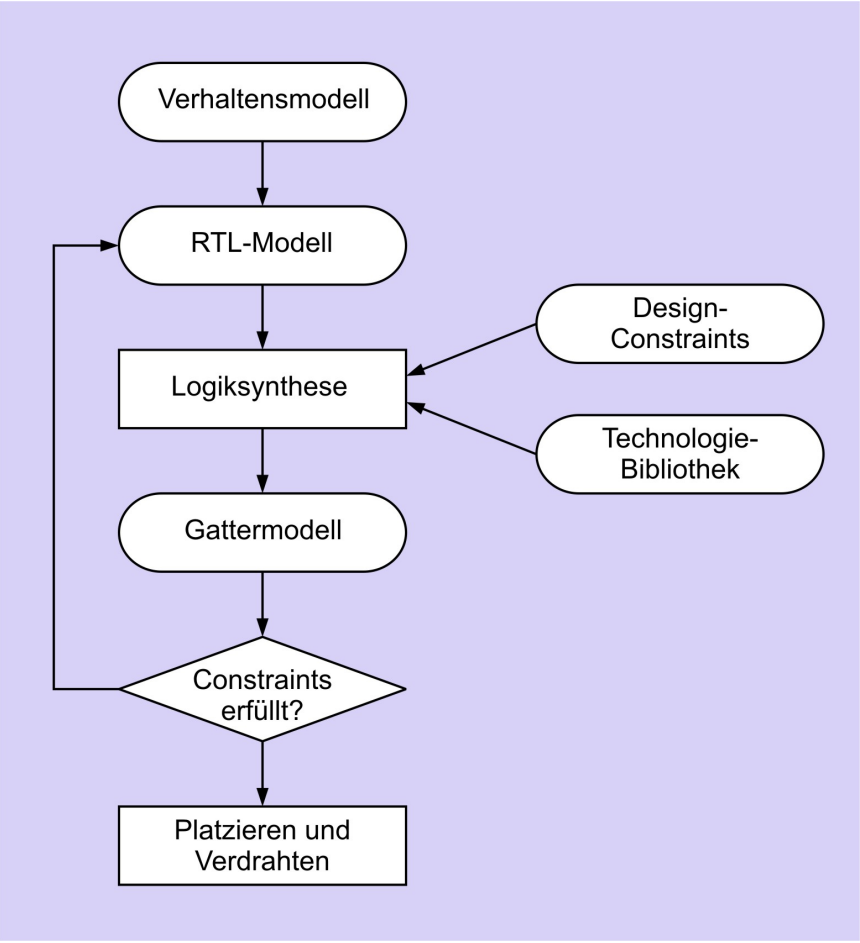
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



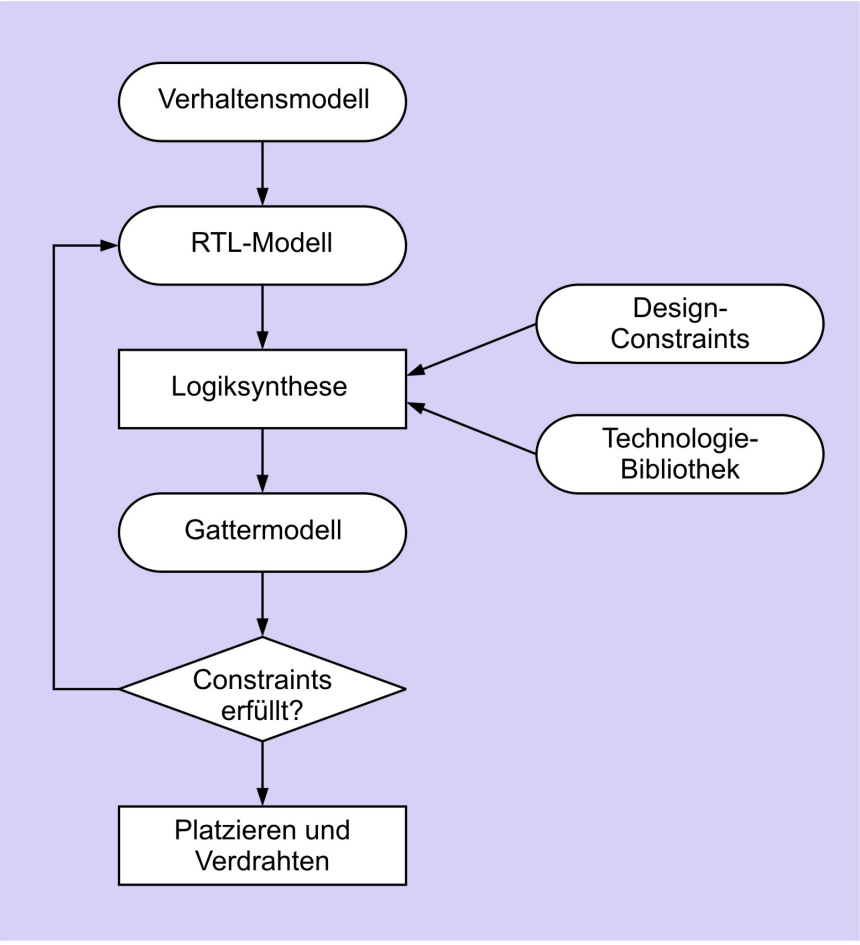
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



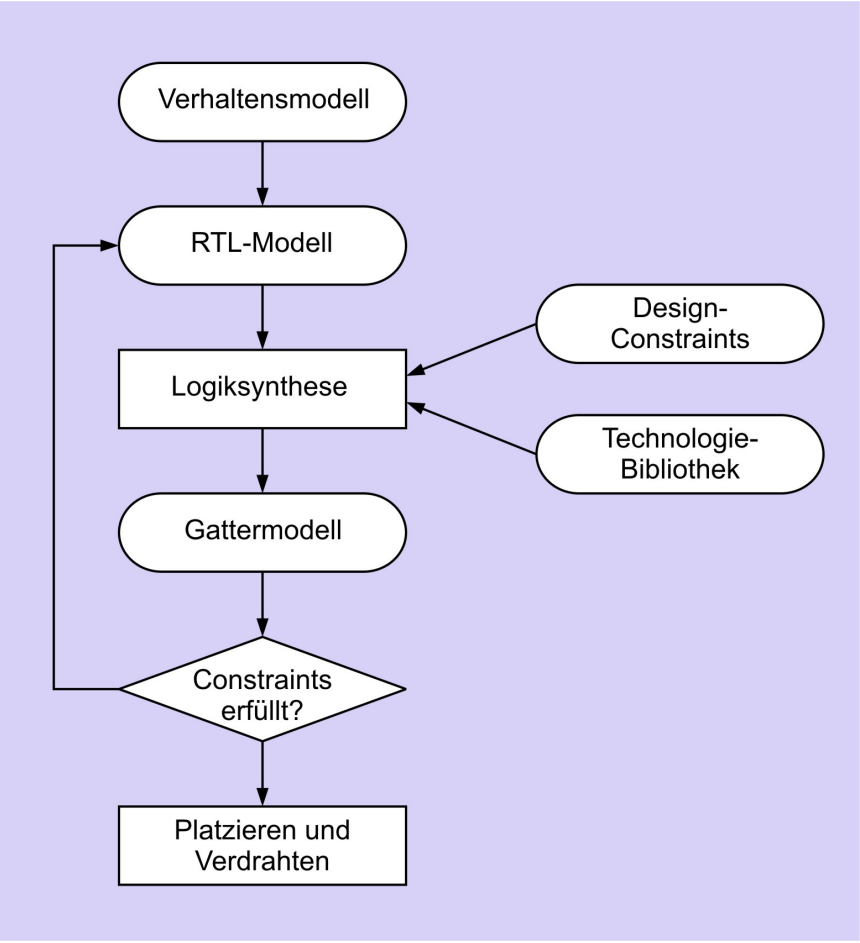
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



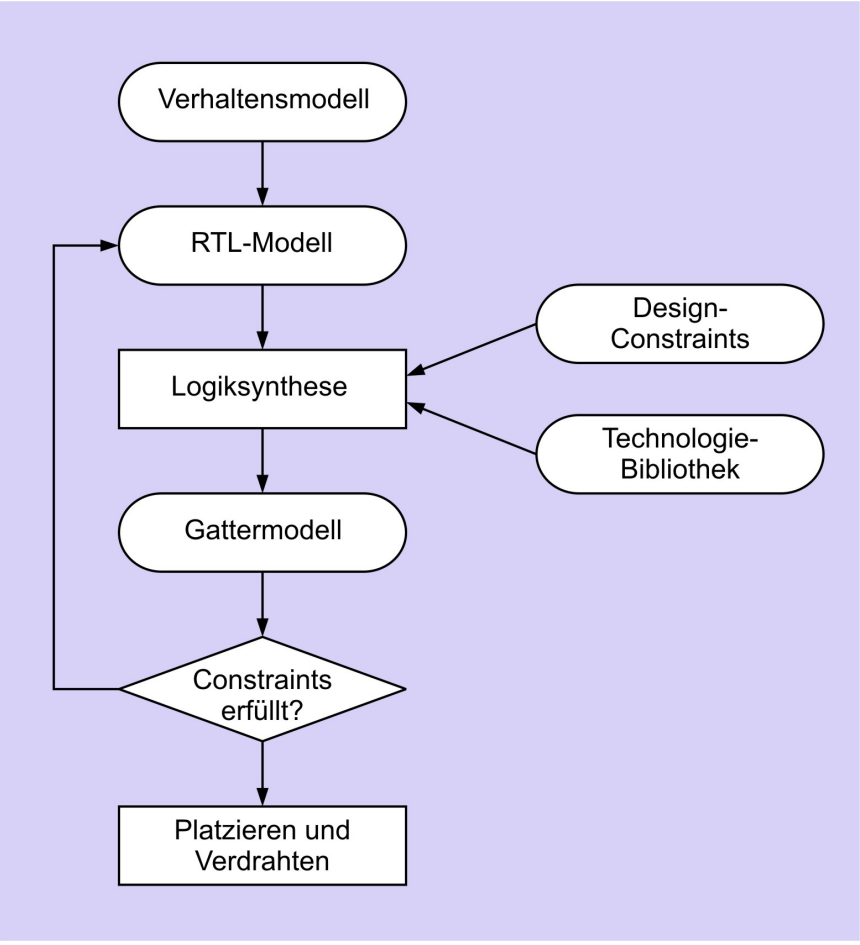
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



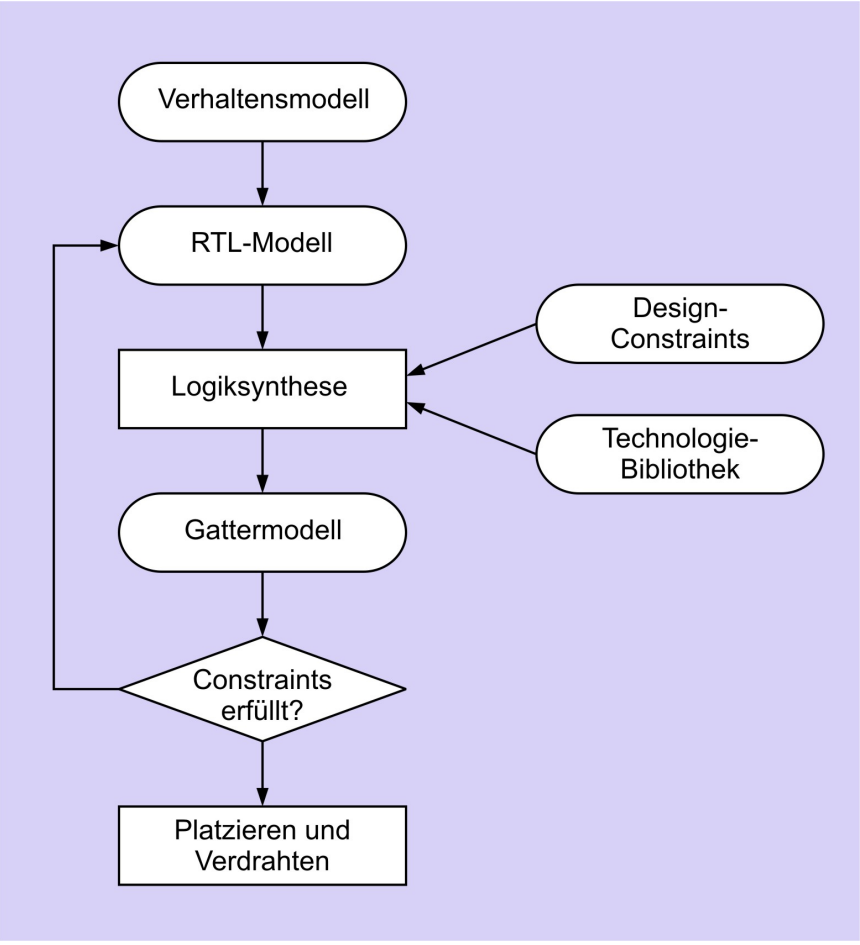
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



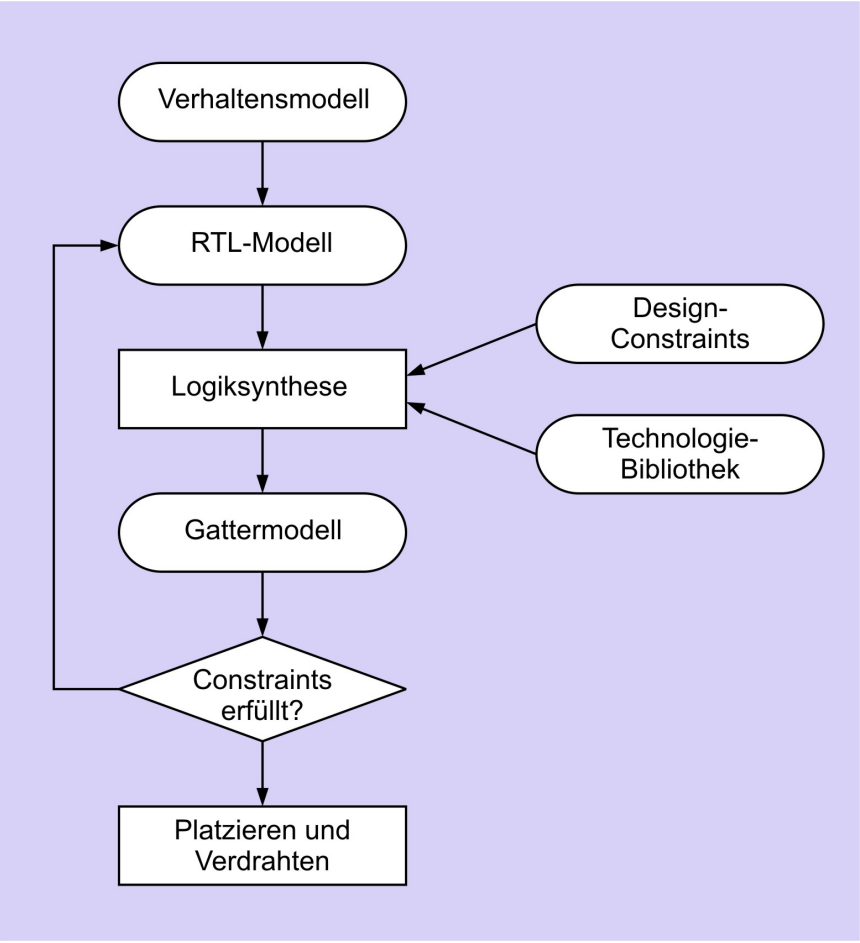
- RTL-Modell in Verilog
- Design-Constraints
  - Wie schnell?
  - Wie groß?
  - (Wieviel Energie?)
- Zieltechnologie
  - AND, OR
  - Addierer, Flip-Flops
  - Abbildung auf LUTs
  - Genaue Laufzeiten
  - Genaue Flächenangaben

Ablauf der Logiksynthese



- 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

Ablauf der Logiksynthese



- 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

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

Vorteile der Logiksynthese

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

Vorteile der Logiksynthese

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

Vorteile der Logiksynthese

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

Vorteile der Logiksynthese

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

Vorteile der Logiksynthese

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

Vorteile der Logiksynthese

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

Vorteile der Logiksynthese

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

Wichtigste synthetisierbare Verilog-Konstrukte

Signale und Variablen	`wire, reg`

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

Struktur	`module,`
	`input, inout, output,`
	`parameter,`
	
	`assign`

Eingeschränkt: `for`

Einige nichtsynthetisierbare Verilog-Konstrukte

- initial: Stattdessen explizites Reset-Verhalten beschreiben
- Zeitkontrolle: `#` und `@` innerhalb von Block
  - Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie

➥Prä- und Post-Synthese-Simulationen können differieren

Einige nichtsynthetisierbare Verilog-Konstrukte

- initial: Stattdessen explizites Reset-Verhalten beschreiben
- Zeitkontrolle: `#` und `@` innerhalb von Block
  - Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie

➥Prä- und Post-Synthese-Simulationen können differieren

Einige nichtsynthetisierbare Verilog-Konstrukte

- initial: Stattdessen explizites Reset-Verhalten beschreiben
- Zeitkontrolle: `#` und `@` innerhalb von Block
  - Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie

➥Prä- und Post-Synthese-Simulationen können differieren

Einige nichtsynthetisierbare Verilog-Konstrukte

- initial: Stattdessen explizites Reset-Verhalten beschreiben
- Zeitkontrolle: `#` und `@` innerhalb von Block
  - Alle Zeitverzögerungen aus Beschreibung der Zieltechnologie

➥Prä- und Post-Synthese-Simulationen können differieren

Einige nichtsynthetisierbare Verilog-Konstrukte

- 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

arithmetisch	`*, /, +, -, %`
logisch	`!, &&, ||`
bit-weise	`~, &, |, ^, ^~, ~^`
Reduktion	`&, ~&, |, ~|, ^, ^~, ~^`
Relation	`>, <, >=, <=`
Gleichheit	`==, !=` **aber kein `===` und `!==` mit `x` und `z`**
Shift	`>>, <<, <<<, >>>`
Konkatenation	`{ }`
bedingt	`?:`

Syntheseergebnisse

Vorgehen

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

Vorgehen

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

Vorgehen

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

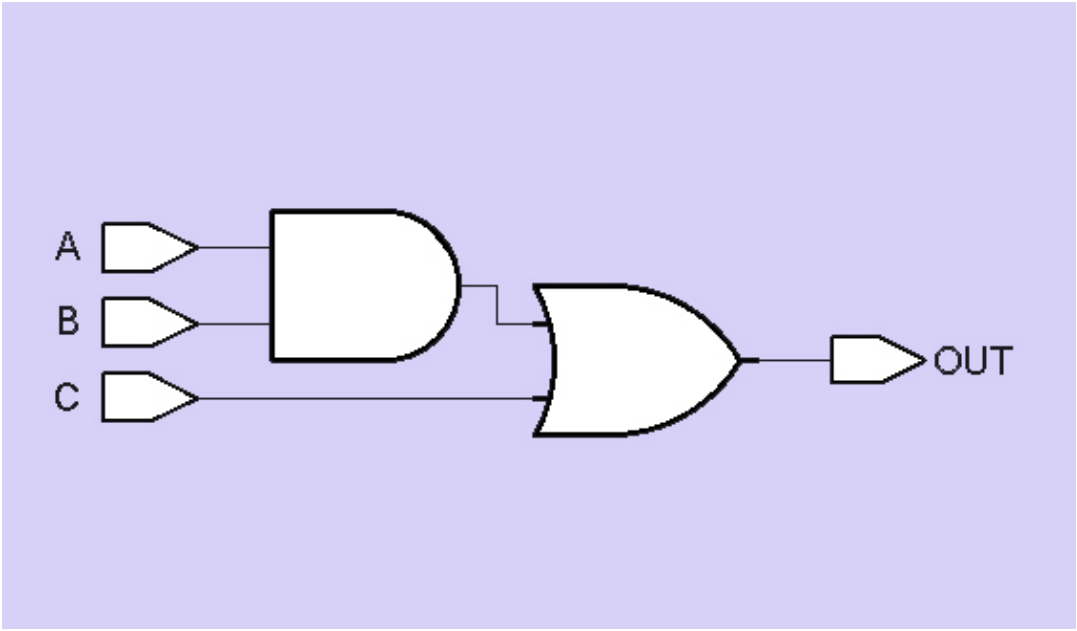
Synthese einer `assign`-Anweisung

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

Annahme hier: Alle Signale 1b breit

Synthese einer `assign`-Anweisung

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



Annahme hier: Alle Signale 1b breit

Synthese einer `assign`-Anweisung

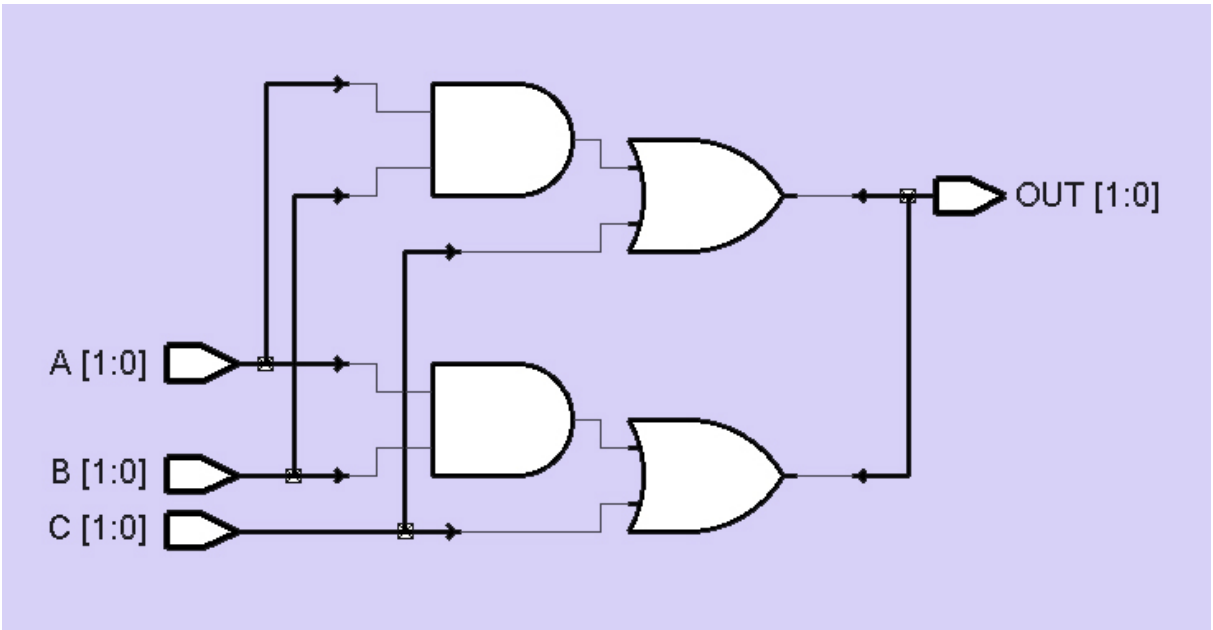
Hier: 2b breite Signale

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

Synthese einer `assign`-Anweisung

Hier: 2b breite Signale


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



Synthese einer `assign`-Anweisung

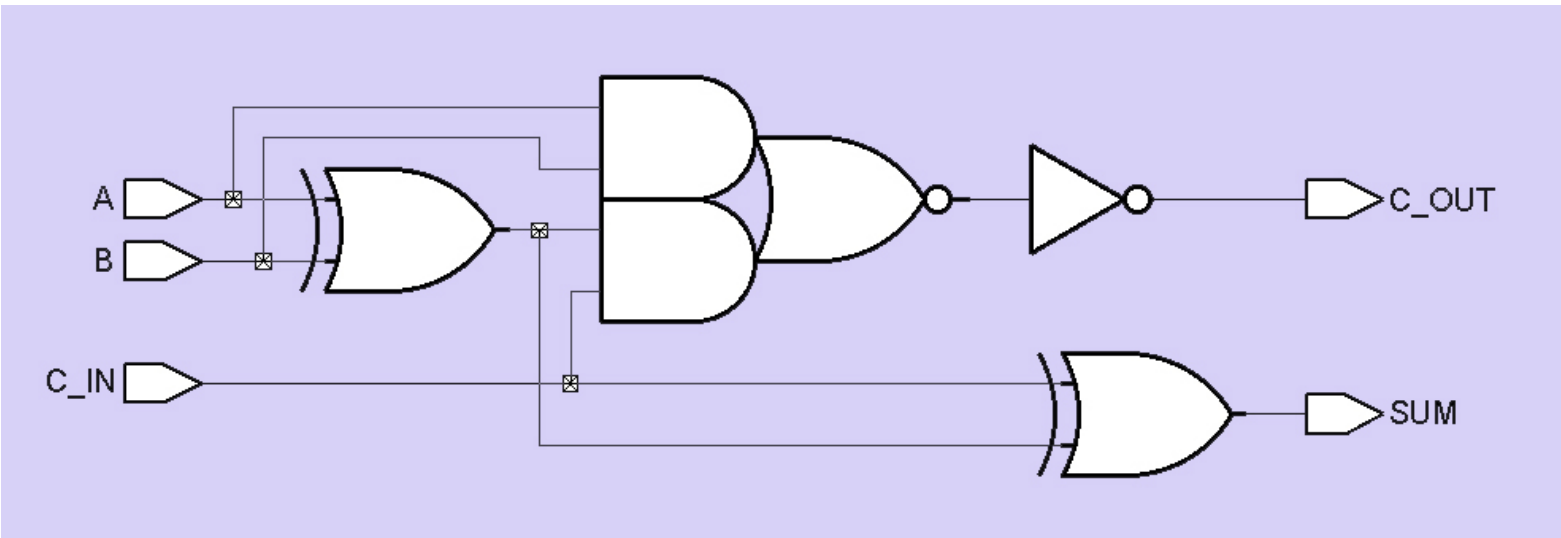
assign {C_OUT, SUM} = A + B + C_IN

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

Synthese einer `assign`-Anweisung

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

1b-Volladdierer



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

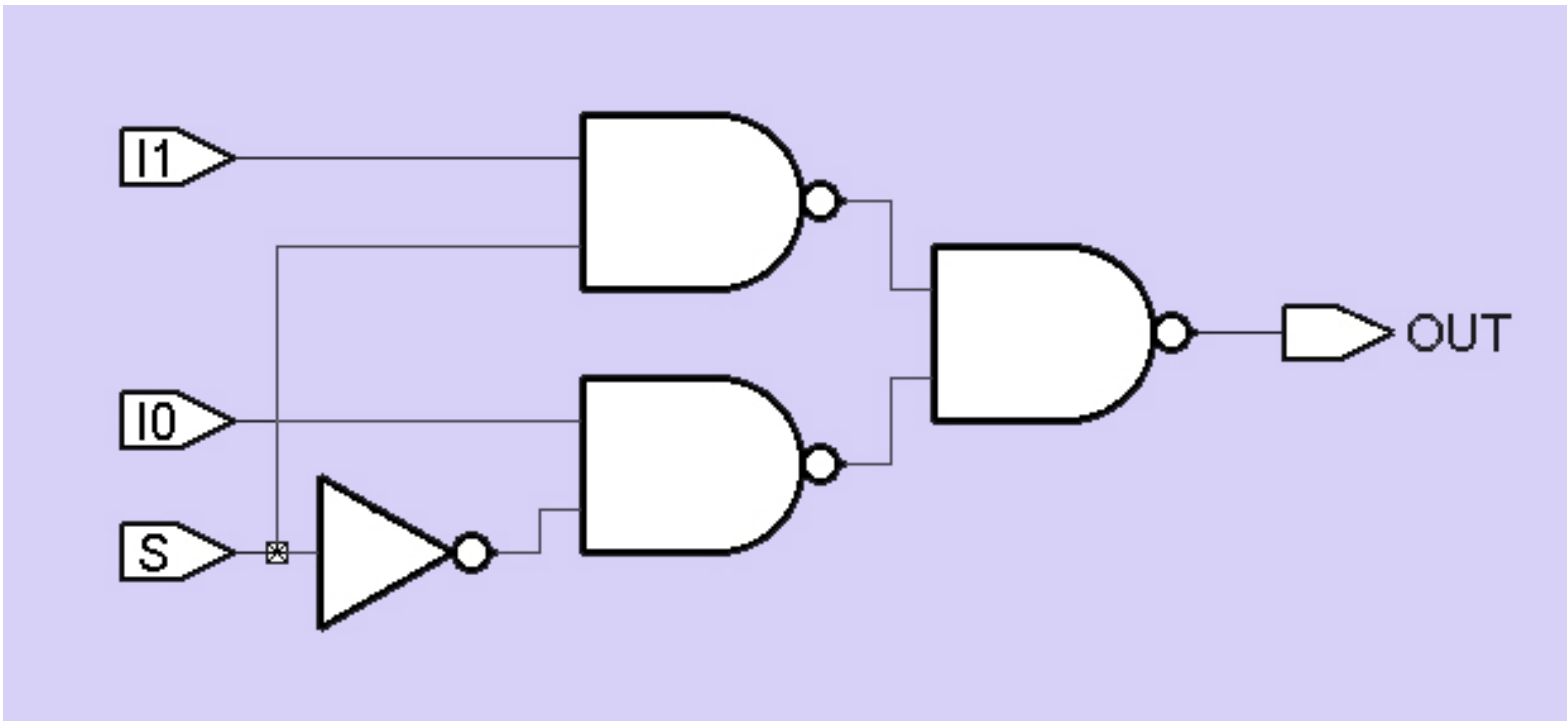
Synthese einer `assign`-Anweisung

assign OUT = (S) ? I1 : I0

Synthese einer `assign`-Anweisung

assign OUT = (S) ? I1 : I0

Multiplexer



Synthese von `if/else` und `case`

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

Synthese von `if/else` und `case`

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

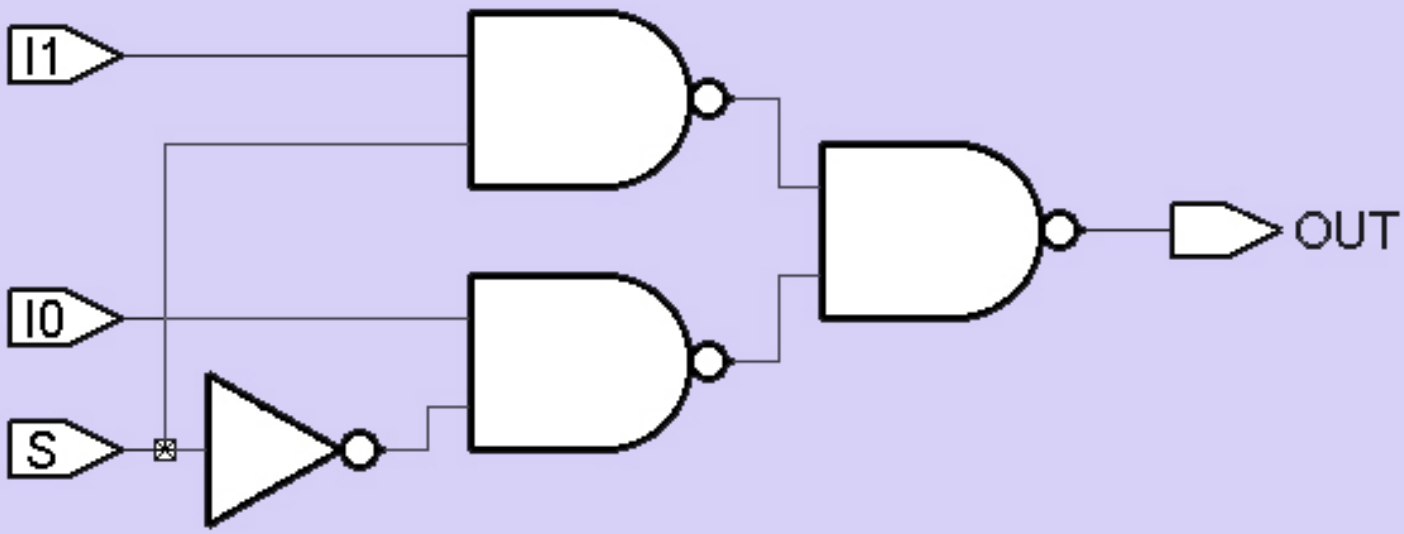
```
case (S)
   0: OUT = I0;
   1: OUT = I1;
endcase
```

Synthese von `if/else` und `case`

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

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

Multiplexer



Synthese von Speicherelementen

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

Vorderflankengesteuertes Flip-Flop

Pegelgesteuertes Latch

Synthese von Speicherelementen

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

Vorderflankengesteuertes Flip-Flop

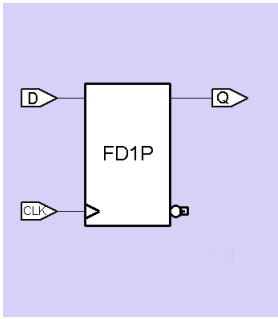
Pegelgesteuertes Latch

Synthese von Speicherelementen

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

Vorderflankengesteuertes Flip-Flop

Pegelgesteuertes Latch

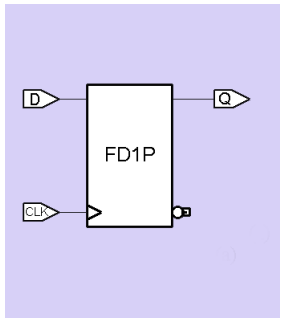


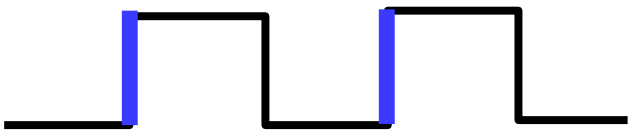
Synthese von Speicherelementen

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

Vorderflankengesteuertes Flip-Flop

Pegelgesteuertes Latch





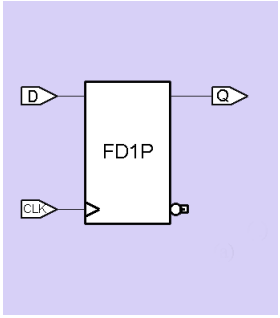
Synthese von Speicherelementen

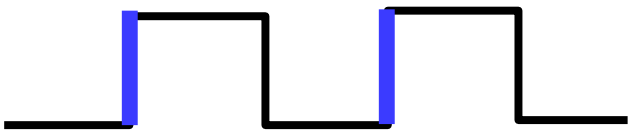
```
always @ (posedge CLK)
  Q <= D;
```

Vorderflankengesteuertes Flip-Flop

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

Pegelgesteuertes Latch

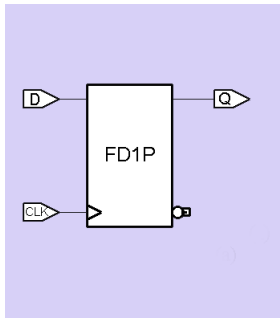


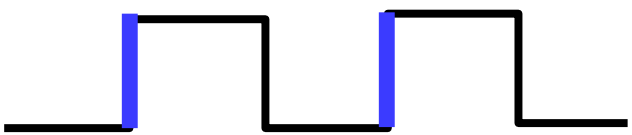


Synthese von Speicherelementen

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

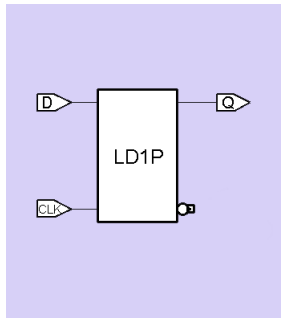
Vorderflankengesteuertes Flip-Flop





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

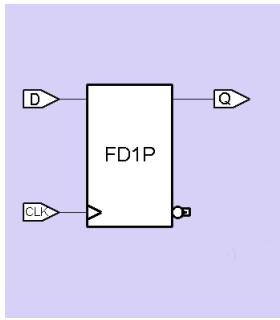
Pegelgesteuertes Latch

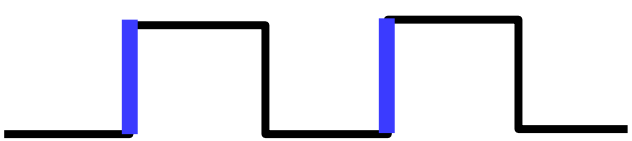


Synthese von Speicherelementen

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

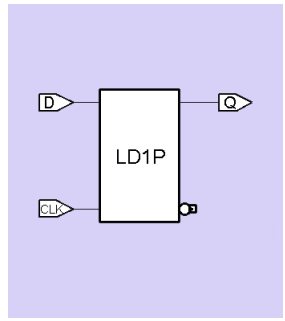
Vorderflankengesteuertes Flip-Flop

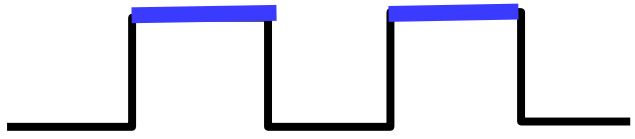




```
always @ (CLK or D)
  if  (CLK) Q = D;
```

Pegelgesteuertes Latch





Wichtig: Unterschiede

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

- Verilog-Datentyp `reg`

➥Nicht identisch

Wichtig: Unterschiede

- Hardware-Register
  - (flankengesteuertes) Flip-Flop
  - (pegelgesteuertes) Latch
- Verilog-Datentyp `reg`

➥Nicht identisch

Wichtig: Unterschiede

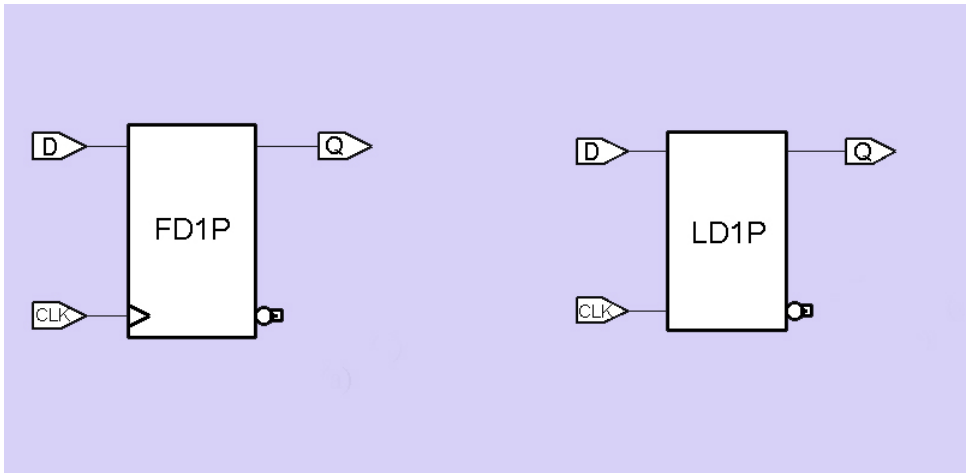
- Hardware-Register
  - (flankengesteuertes) Flip-Flop
  - (pegelgesteuertes) Latch

- Verilog-Datentyp `reg`

➥Nicht identisch

Wichtig: Unterschiede

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

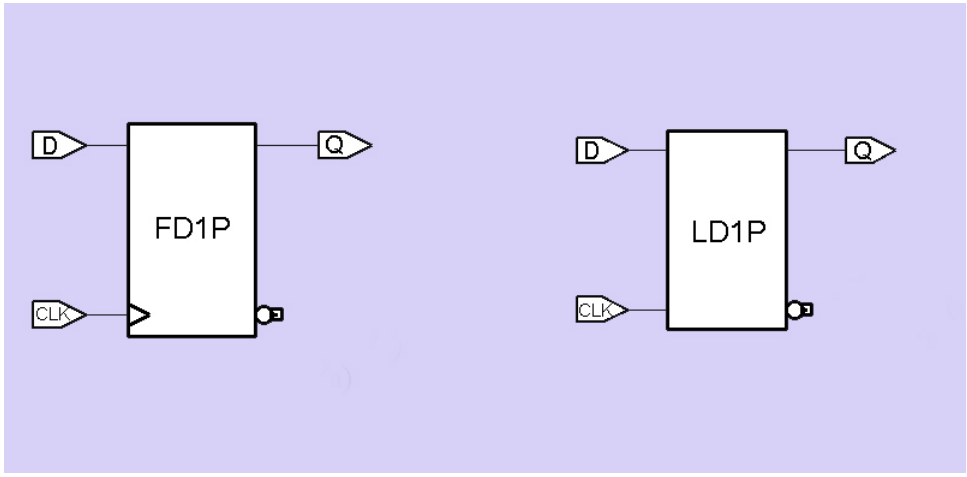


- Verilog-Datentyp `reg`

➥Nicht identisch

Wichtig: Unterschiede

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

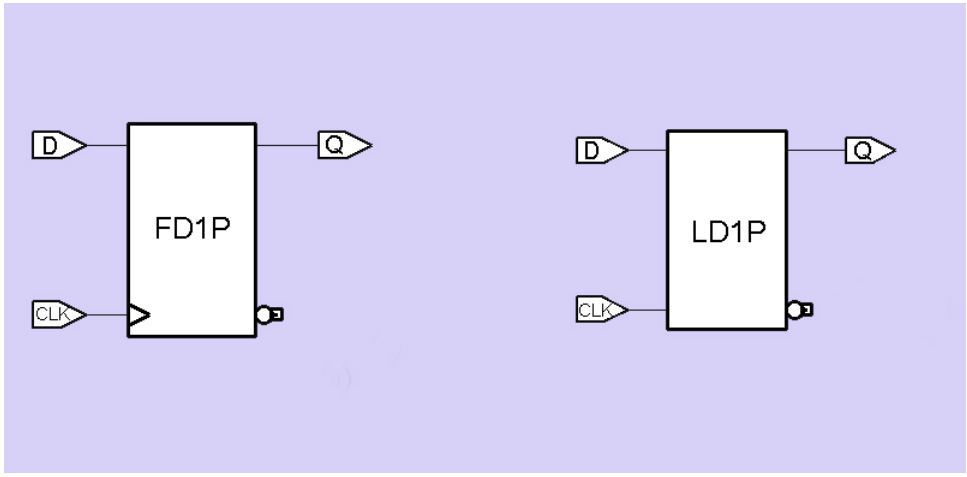


- Verilog-Datentyp **`reg`**

➥Nicht identisch

Wichtig: Unterschiede

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

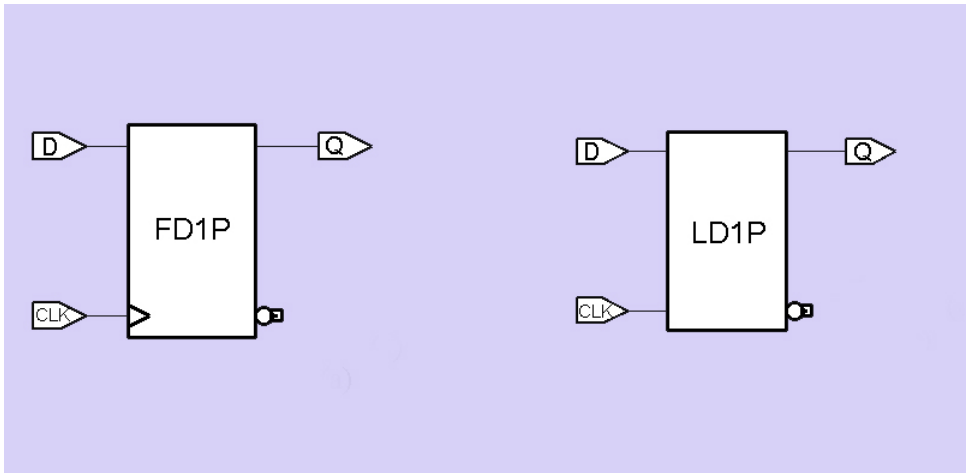


- Verilog-Datentyp **`reg`**

➥Nicht identisch

Wichtig: Unterschiede

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



- Verilog-Datentyp **`reg`**

➥Nicht identisch

Register-Synthese

Grundlage der Modellierung: `always`-Blöcke

- flankengesteuerte always-Blöcke
- flankenfreie always-Blöcke

Grundlage der Modellierung: `always`-Blöcke

- flankengesteuerte always-Blöcke

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

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

- flankenfreie always-Blöcke

Grundlage der Modellierung: `always`-Blöcke

- flankengesteuerte always-Blöcke

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

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

- flankenfreie always-Blöcke

Grundlage der Modellierung: `always`-Blöcke

- flankengesteuerte always-Blöcke

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

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

- flankenfreie always-Blöcke

Grundlage der Modellierung: `always`-Blöcke

- flankengesteuerte always-Blöcke

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

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

- flankenfreie always-Blöcke

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

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

Grundlage der Modellierung: `always`-Blöcke

- flankengesteuerte always-Blöcke

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

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

- flankenfreie always-Blöcke

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

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

Potenzielle Register

Eine Variable Q ist ein potenzielles Register (PR), wenn sie in einem always-Block geschrieben wird (Q = ... , Q <= ...)..

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
Potenzielle Register und Vollständigkeit

Potenzielle Register

Eine Variable q ist ein potenzielles Register (PR), wenn sie in einem always-Block geschrieben wird (q = ... , q <= ...)..

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

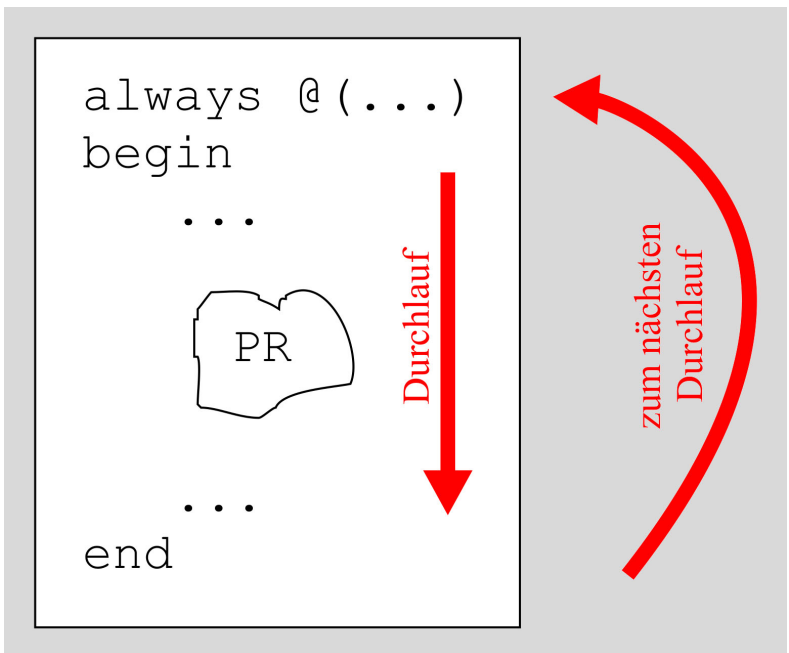
Potenzielle Register

Eine Variable `q` ist ein potenzielles Register (PR), wenn sie in einem always-Block geschrieben wird (`q = ...` , `q <= ...`)..

Vollständigkeit

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

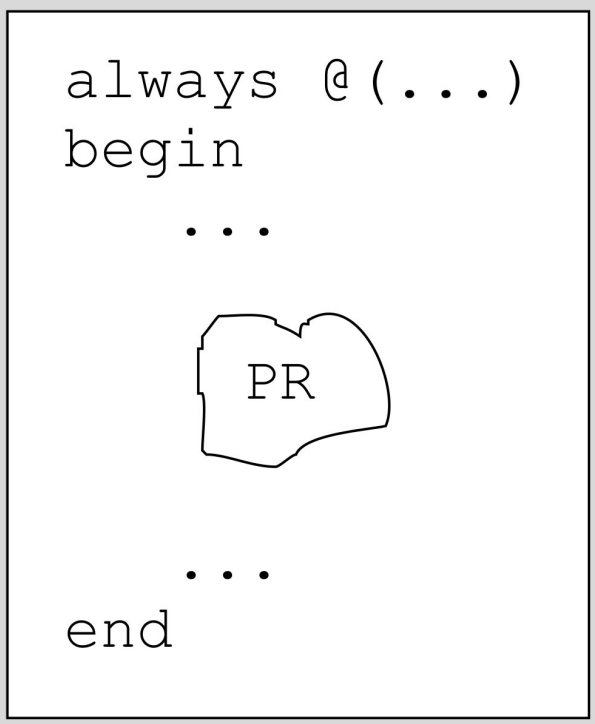
Nicht-redundant: Nicht mit sich selber, also nicht `q = q`

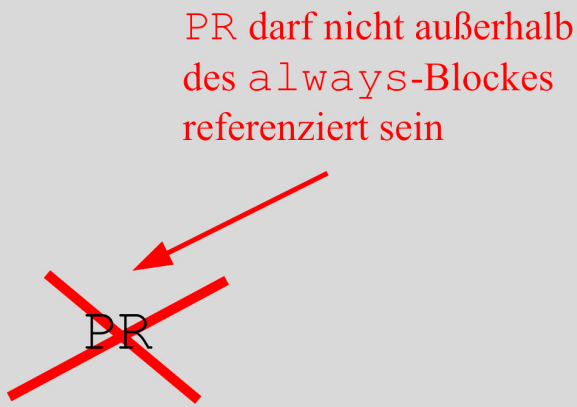


Grundlegende Definitionen

Räumliche Lokalität

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





Beispiele: Räumliche Lokalität

```
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
```

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

Beispiele: Räumliche Lokalität

```
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.

Beispiele: Räumliche Lokalität

```
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.

Beispiele: Räumliche Lokalität

```
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
```

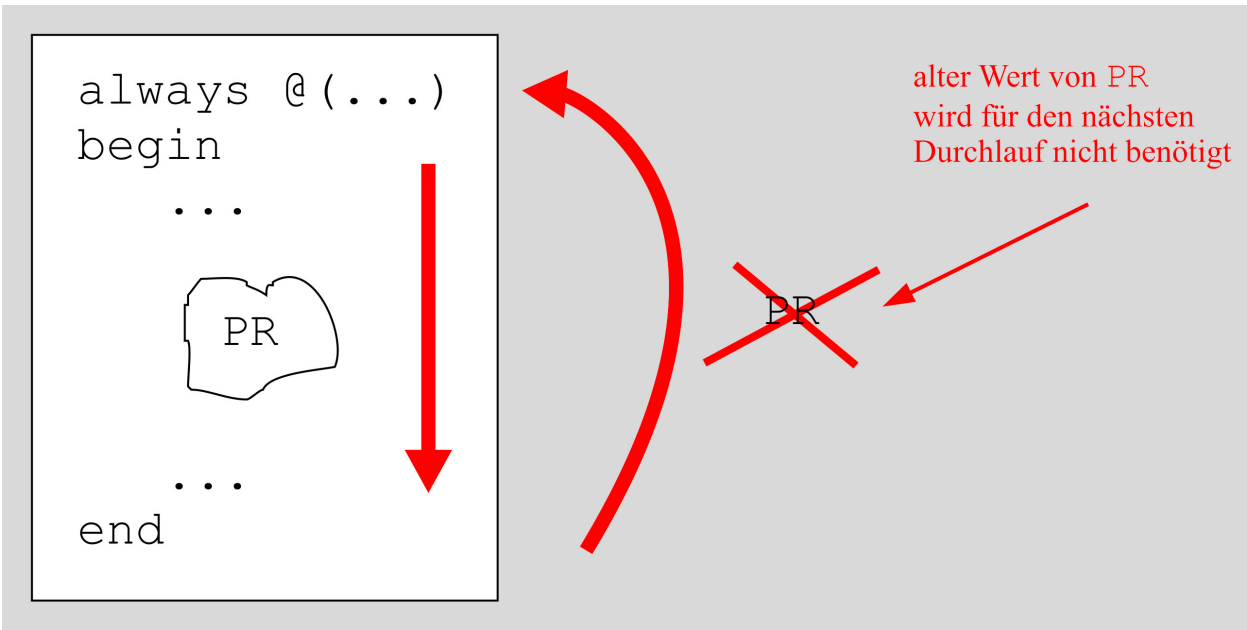
```
module r_lokal_2 (
  input wire A);

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.



Der alte Wert braucht also nicht zwischengespeichert zu werden.

Beispiele: Zeitliche Lokalität

```
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

Beispiele: Zeitliche Lokalität

```
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

Beispiele: Zeitliche Lokalität

```
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

Flankenfreier Always-Block

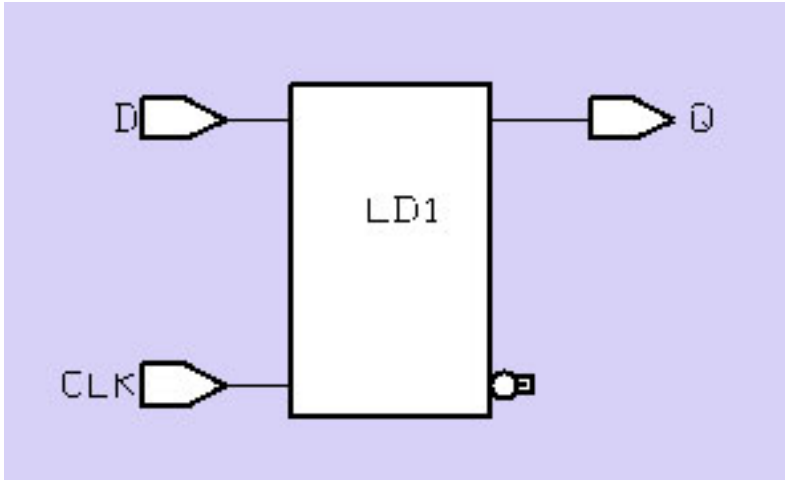
Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch wegen unvollständigem Q

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

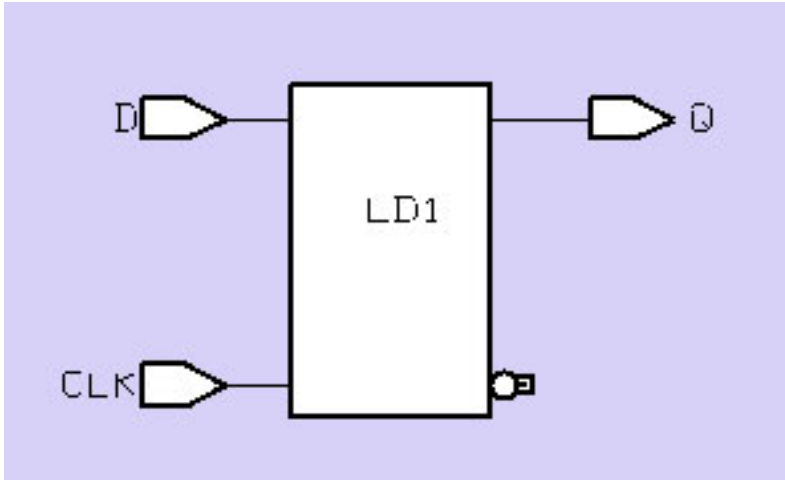
```
always @(CLK, D)
   if  (CLK) Q = D;
```



Latch wegen unvollständigem Q

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

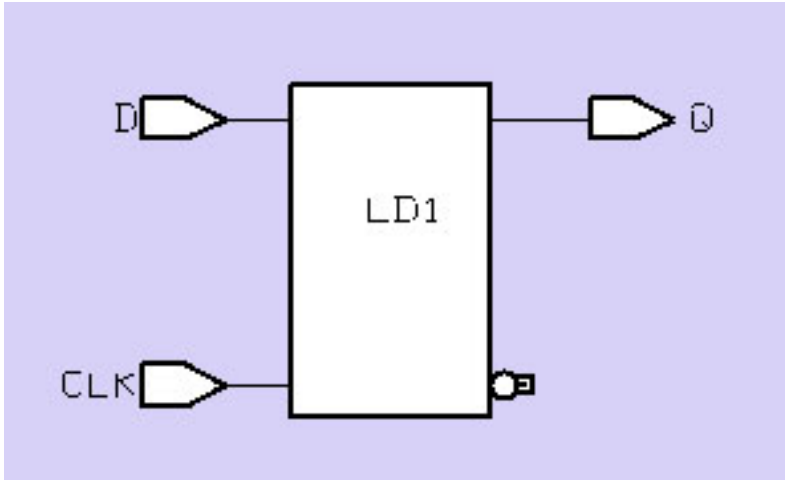
```
always @(CLK, D)
   if  (CLK) Q = D;
```



Latch wegen unvollständigem Q

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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



Latch wegen unvollständigem `Q`

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch vermieden
da Q nun
vollständig

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch vermieden
da Q nun
vollständig

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch vermieden
da `Q` nun
vollständig

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

Signalnamen wie
`CLK` irrelevant.
Kombinatorische
Logik!

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch vermieden
da `Q` nun
vollständig

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

Signalnamen wie
`CLK` irrelevant.
Kombinatorische
Logik!

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch vermieden da `Q` nun vollständig

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

Signalnamen wie `CLK` irrelevant. Kombinatorische Logik!

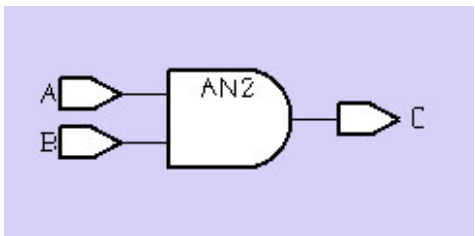
Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

Latch vermieden da `Q` nun vollständig

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

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
```

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

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
```

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

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
```

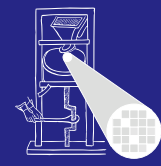
- Latch wegen unvollständigem `case`
- Wie vollständig formulieren?
- Durch angeben von `default`

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
```

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

Diskussion



- 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

Diskussion

- 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

Diskussion

- 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

Diskussion

- 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

Diskussion

- 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

Diskussion

- 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

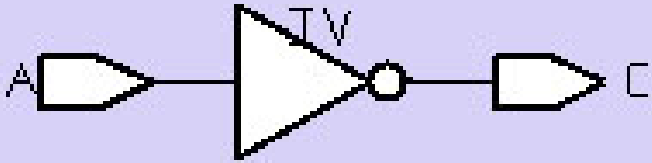
Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

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

reg TMP;

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

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

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

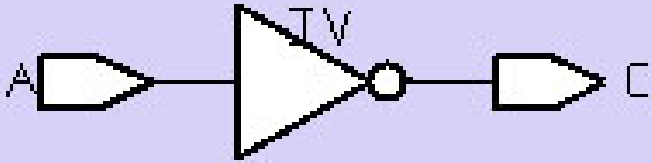
```
module z_lokal (
   input  wire A,
   output reg  C);

reg TMP;

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



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

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

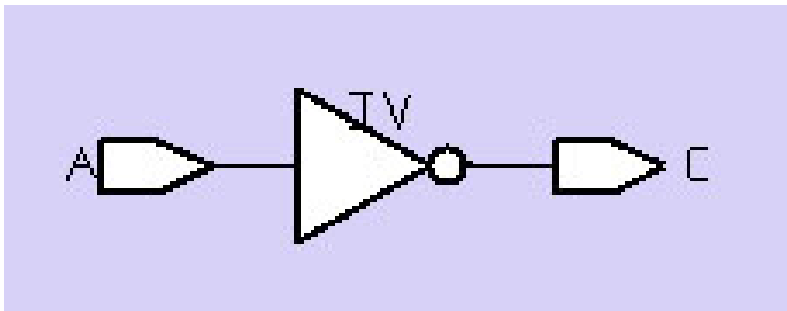
```
module z_lokal (
   input  wire A,
   output reg  C);

reg TMP;

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



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

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
```



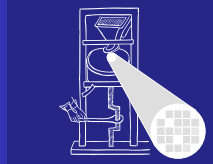
Latch wird vermieden, da `TMP` vollständig ist (und zeitlich lokal).

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

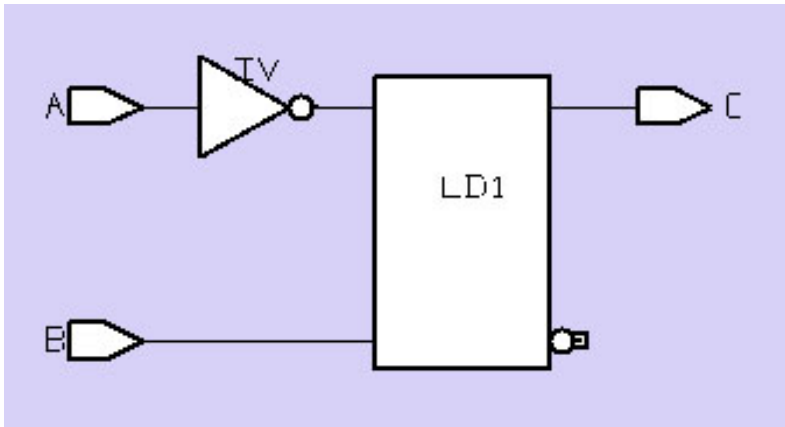
```
module lokal (
   input  wire A, B,
   output reg C);

reg TMP;

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

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

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

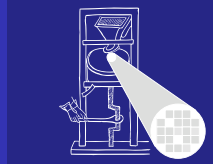


```
module lokal (
   input  wire A, B,
   output reg C);

reg TMP;

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

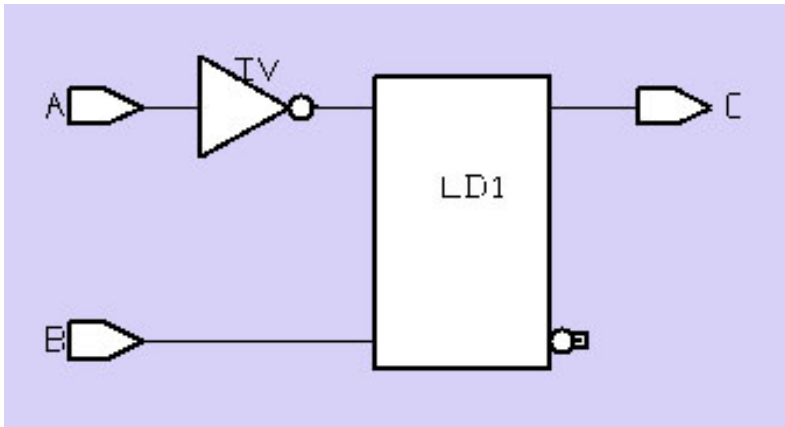


Latch entsteht, da `TMP` unvollständig ist und nicht zeitlich lokal.

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block



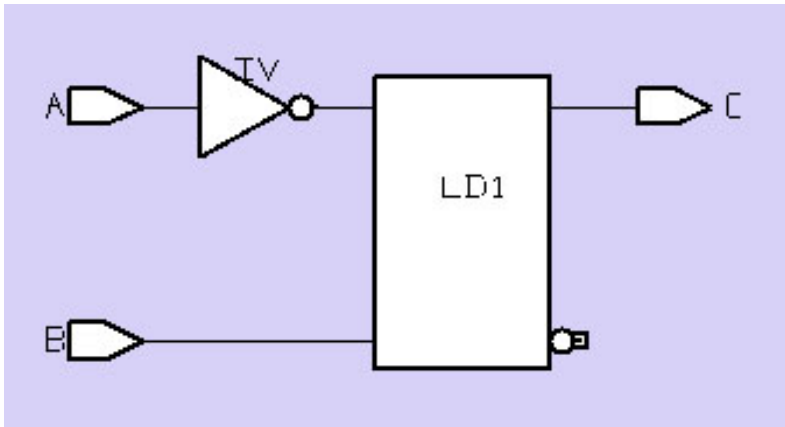
```
module lokal (
   input  wire A, B,
   output reg C);

reg TMP;

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



Latch entsteht, da `TMP` unvollständig ist und nicht 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
```



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

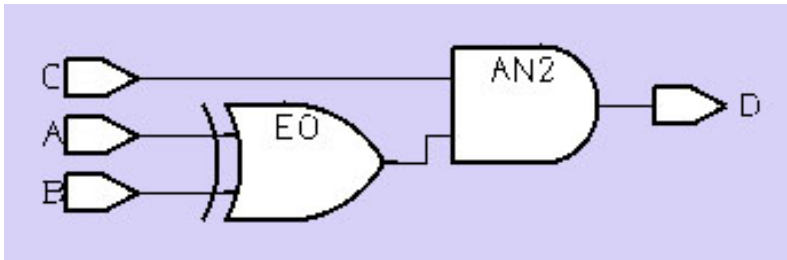
Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

```
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

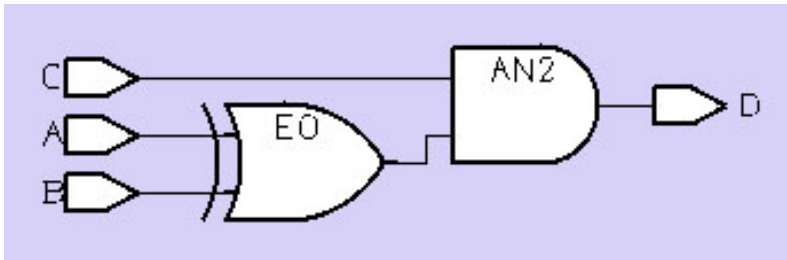
```
module lokal(
   input  wire A, B, C,
   output reg  D);

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

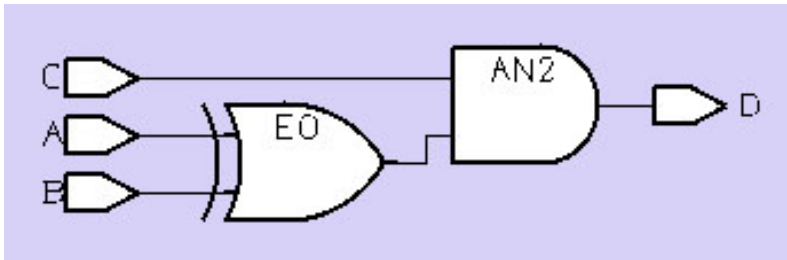
```
module lokal(
   input  wire A, B, C,
   output reg  D);

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

```
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.

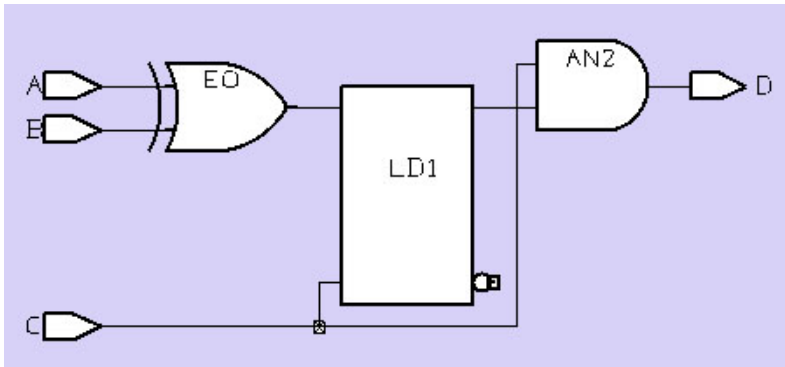
Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

```
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.

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

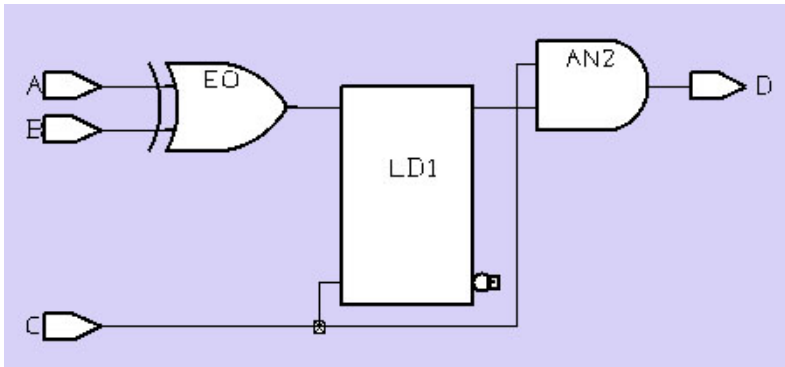
```
module lokal (
   input  wire A, B, C,
   output reg  D);

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.

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

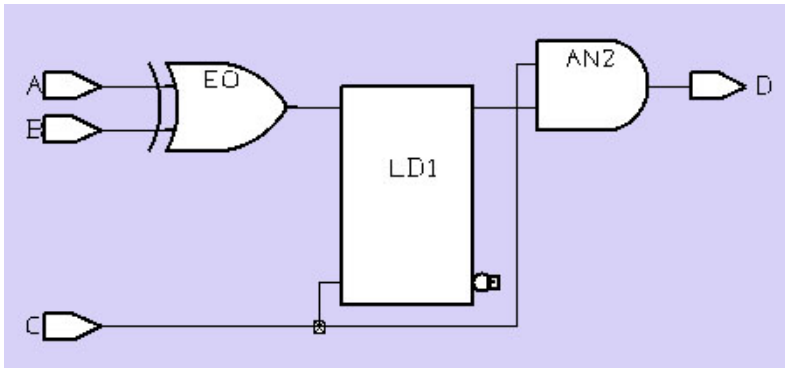
```
module lokal (
   input  wire A, B, C,
   output reg  D);

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.

Beispiele: Logik oder Latch?
Flankenfreier `always`-Block

```
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

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Richtlinien
Flankenfreier `always`-Block

- 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!

Getakteter Always-Block

Flip-Flop
Getakteter `always`-Block

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

Flip-Flop
Getakteter `always`-Block

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

Flip-Flop
Getakteter `always`-Block

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

Flip-Flop
Getakteter `always`-Block

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

Flip-Flop
Getakteter `always`-Block

- 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

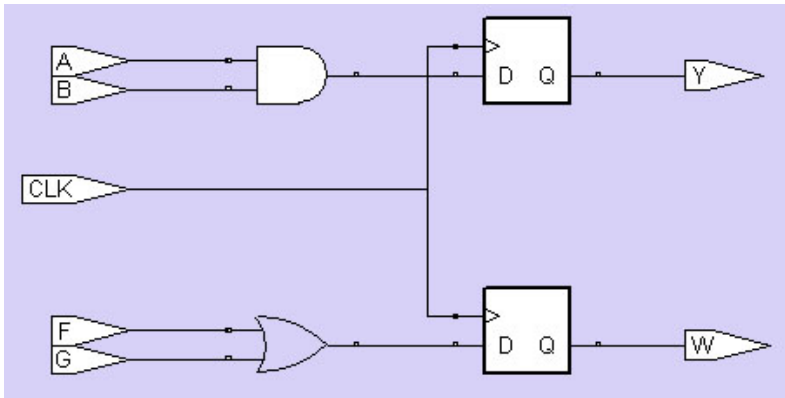
- Mit `posedge` oder `negedge` in der Aktivierungsliste
- Jedes nicht-lokale potenzielle Register wird Flip-Flop
- Vollständigkeit ist nun irrelevant.

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

Flip-Flop
Getakteter `always`-Block

- 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
```



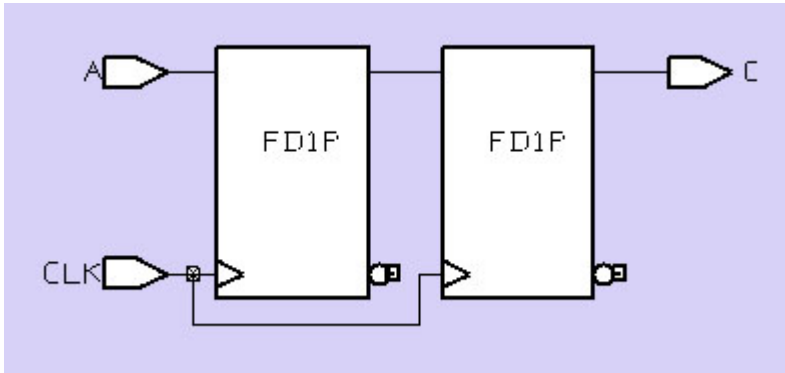
Flip-Flop
Getakteter `always`-Block

```
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

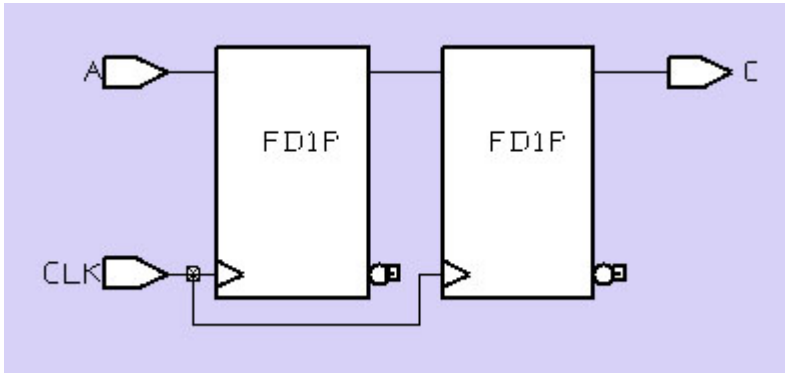
```
module lokal (
   input  wire CLK, A,
   output reg  C);

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

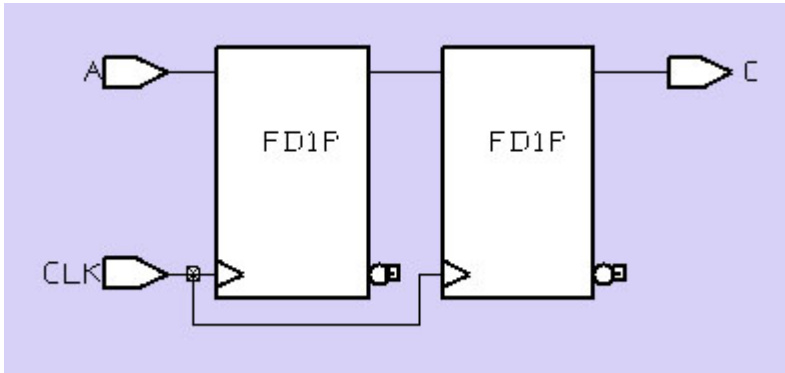
```
module lokal (
   input  wire CLK, A,
   output reg  C);

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

```
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.

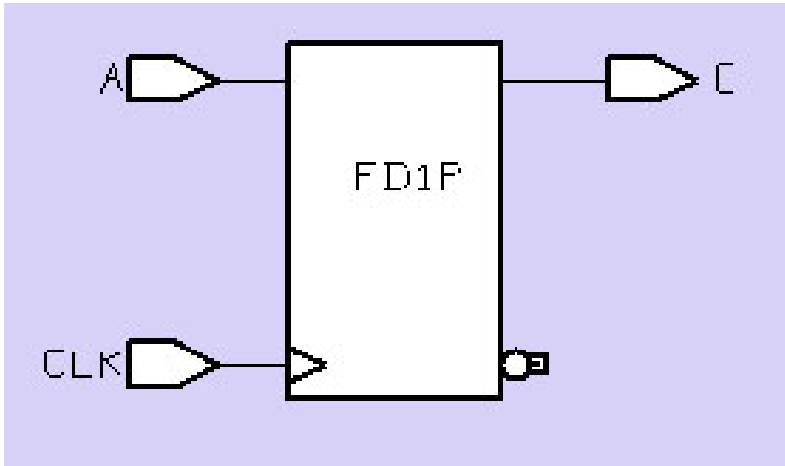
Flip-Flop
Getakteter `always`-Block

```
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
Getakteter `always`-Block

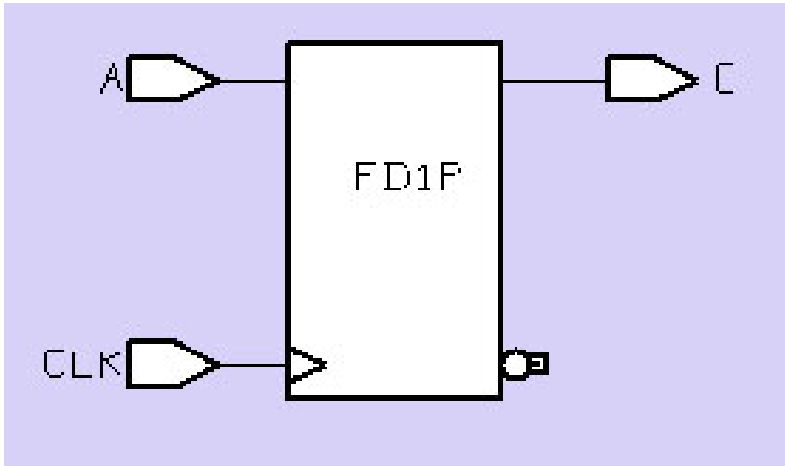
```
module lokal (
   input  wire CLK, A,
   output reg  C);

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
Getakteter `always`-Block

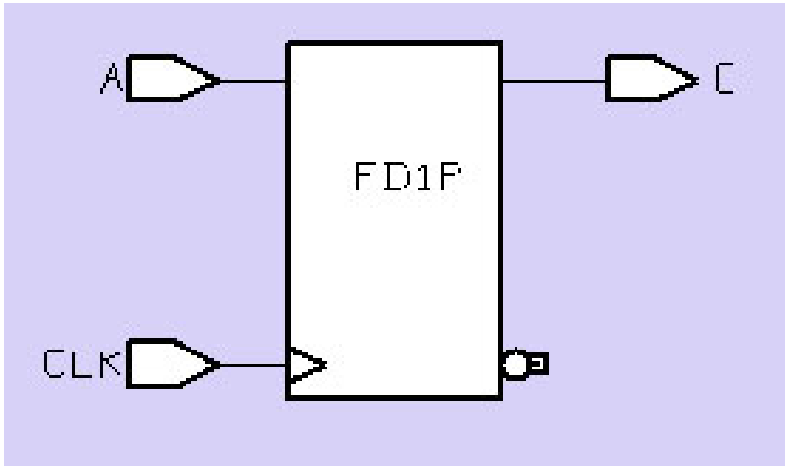
```
module lokal (
   input  wire CLK, A,
   output reg  C);

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
Getakteter `always`-Block

```
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.

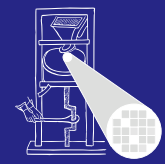
Flip-Flop mit Reset
Getakteter `always`-Block

```
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
```

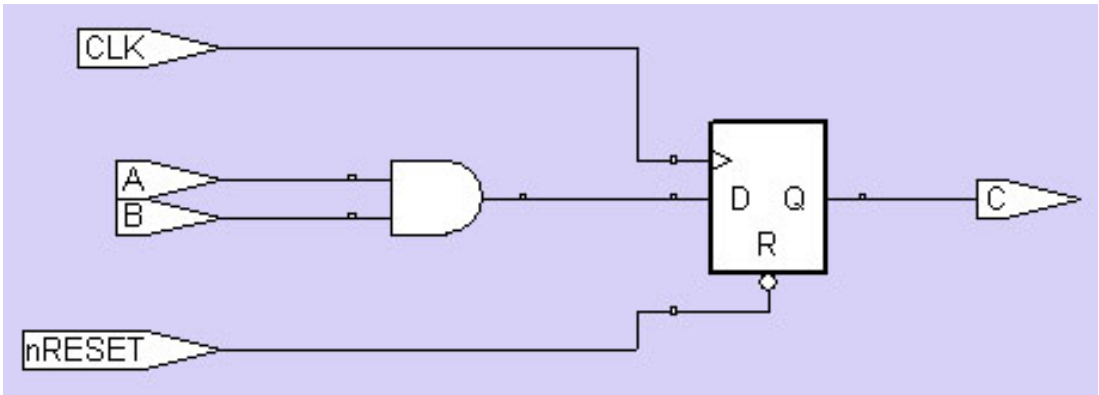
Flip-Flop mit asynchronem Reset

Flip-Flop mit Reset

Getakteter `always`-Block



```
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

Zuweisungen

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

Richtlinien für Zuweisungen

- 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

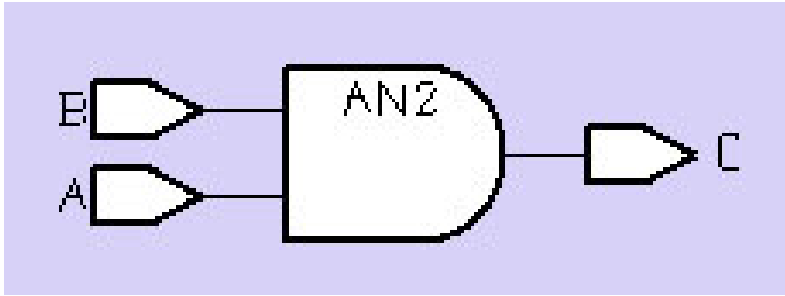
So nicht: Nichtblockende Zuweisungen

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

Trotz <= kombinatorische Logik, da c vollständig beschrieben und lokal verwendet.

So nicht: Nichtblockende Zuweisungen

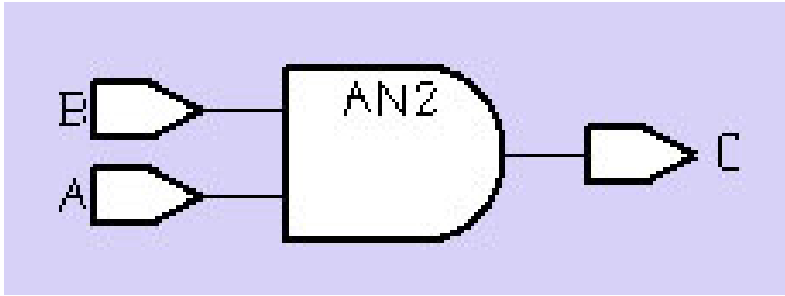
```
always @ (A, B)
begin
   C <= 0;
   if  (B) C <= A;
end
```



Trotz <= kombinatorische Logik, da c vollständig beschrieben und lokal verwendet.

So nicht: Nichtblockende Zuweisungen

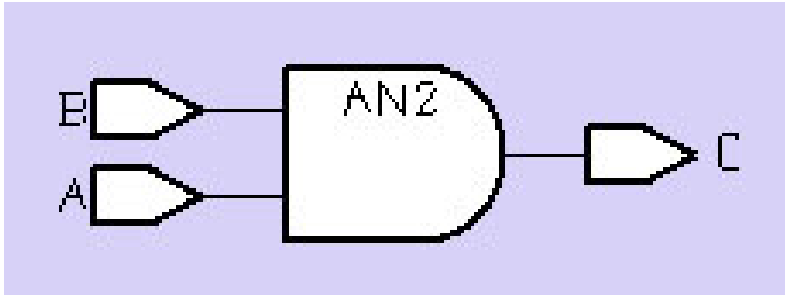
```
always @ (A, B)
begin
   C <= 0;
   if  (B) C <= A;
end
```



Trotz <= kombinatorische Logik, da c vollständig beschrieben und lokal verwendet.

So nicht: Nichtblockende Zuweisungen

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



Trotz <= kombinatorische Logik, da c vollständig beschrieben und lokal verwendet.

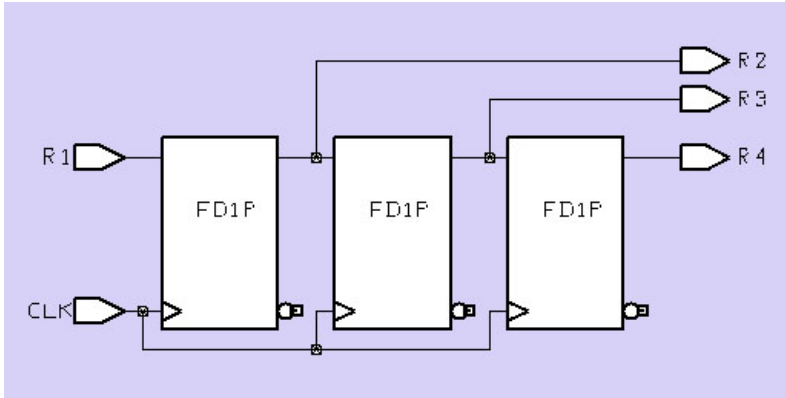
So nicht: Blockende Zuweisungen

```
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 (0,1) verwendet

So nicht: Blockende Zuweisungen

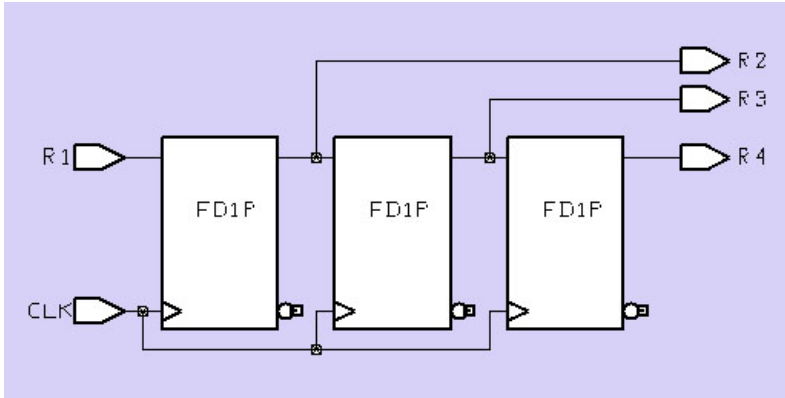
```
always @ (posedge CLK)
  R2 = R1;

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 (0,1) verwendet

So nicht: Blockende Zuweisungen

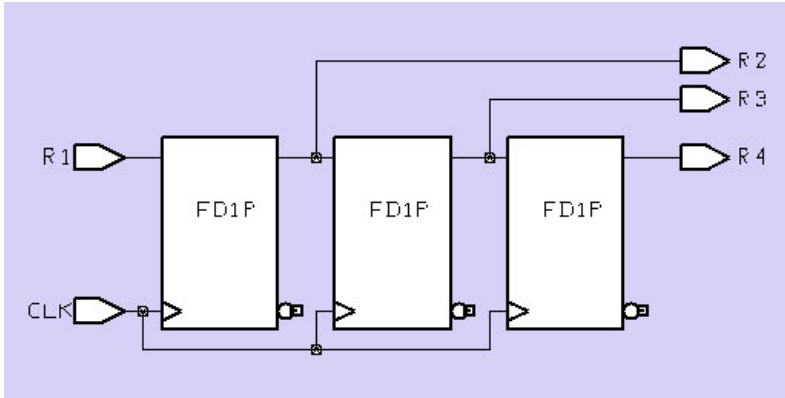
```
always @ (posedge CLK)
  R2 = R1;

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 (0,1) verwendet

So nicht: Blockende Zuweisungen

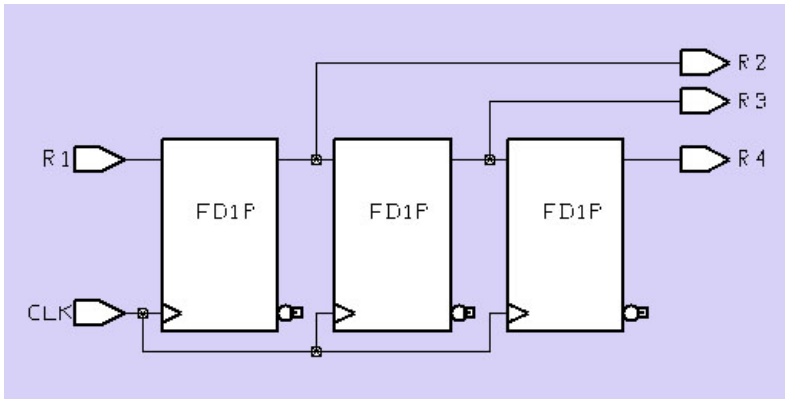
```
always @ (posedge CLK)
  R2 = R1;

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

So nicht: Blockende Zuweisungen

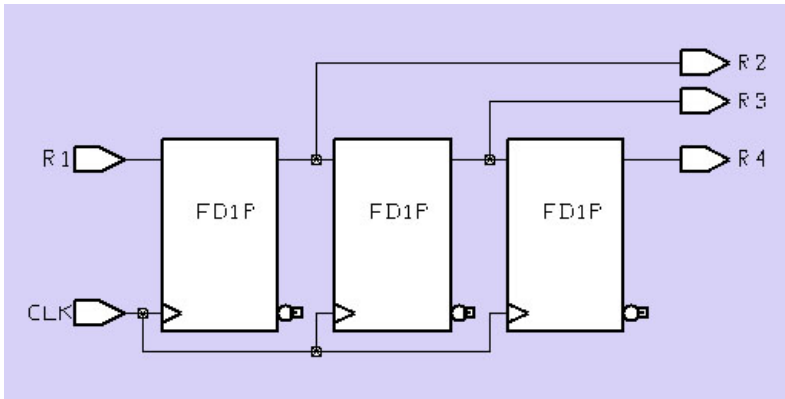
```
always @ (posedge CLK)
  R2 = R1;

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

So nicht: Blockende Zuweisungen

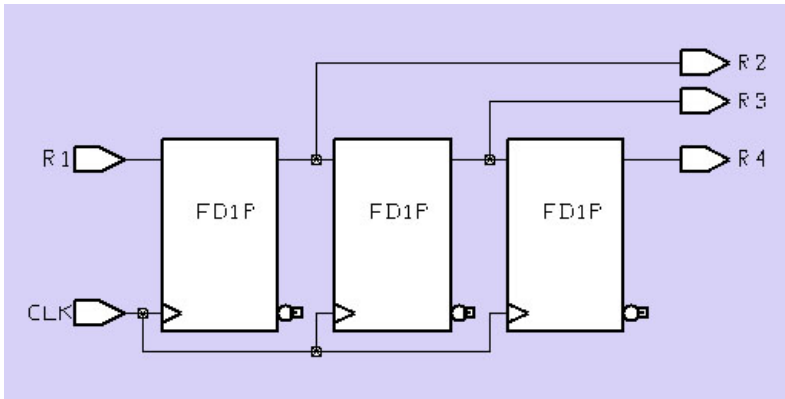
```
always @ (posedge CLK)
  R2 = R1;

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

So nicht: Blockende Zuweisungen

```
always @ (posedge CLK)
  R2 = R1;

always @ (posedge CLK)
  R3 = R2;

always @ (posedge CLK)
  R4 = R3;
```

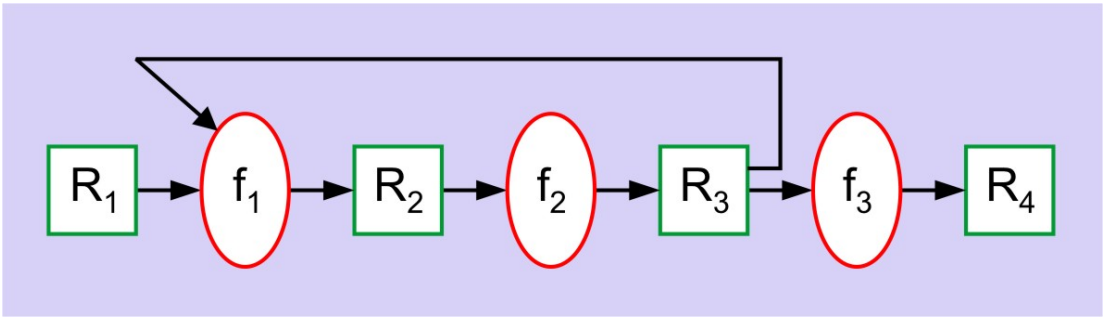


- In Simulation: Zufällige Ausführungsreihenfolge
  - R2=, R3=, R4 oder R4=, R2=, R3= . . .
- In Synthese: Immer Schieberegister aus Flip-Flops
  - R2, R3, R4 nicht nur lokal verwendet

Kombinatorische Logik und Register

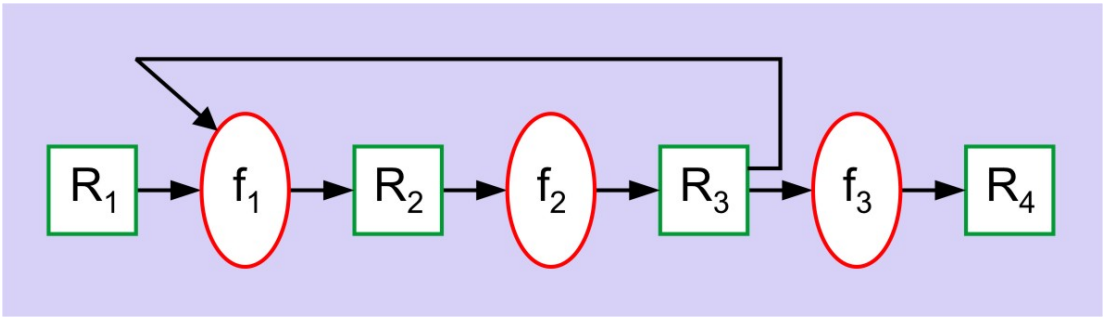
Trennung von kombinatorischer Logik und Registern

- 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)



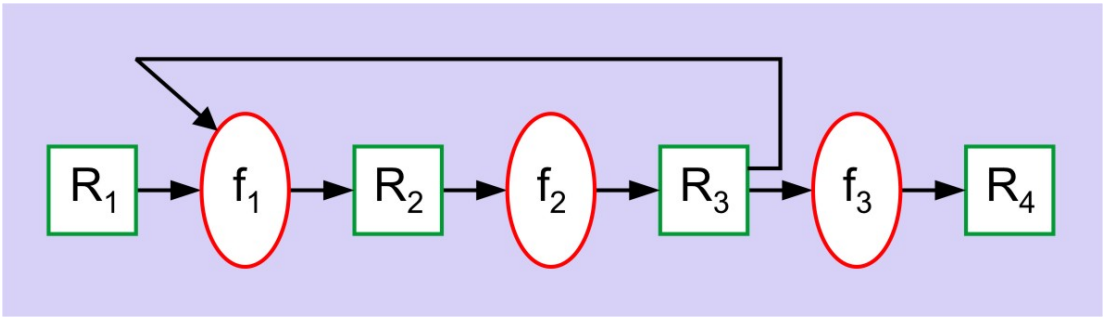
Trennung von kombinatorischer Logik und Registern

- 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)



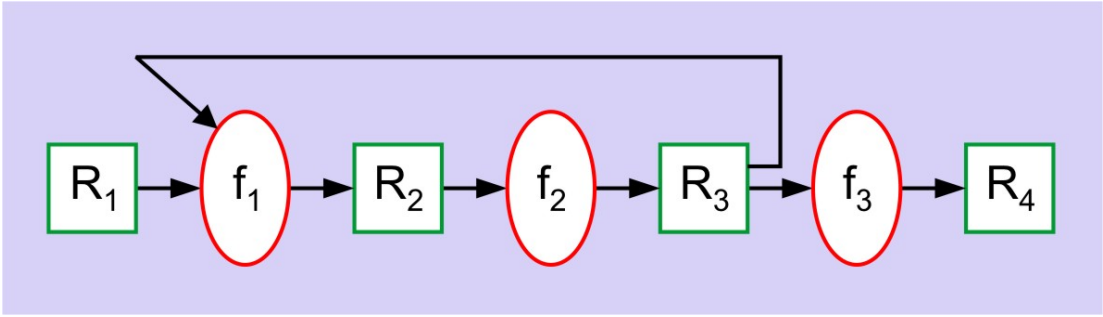
Trennung von kombinatorischer Logik und Registern

- 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)



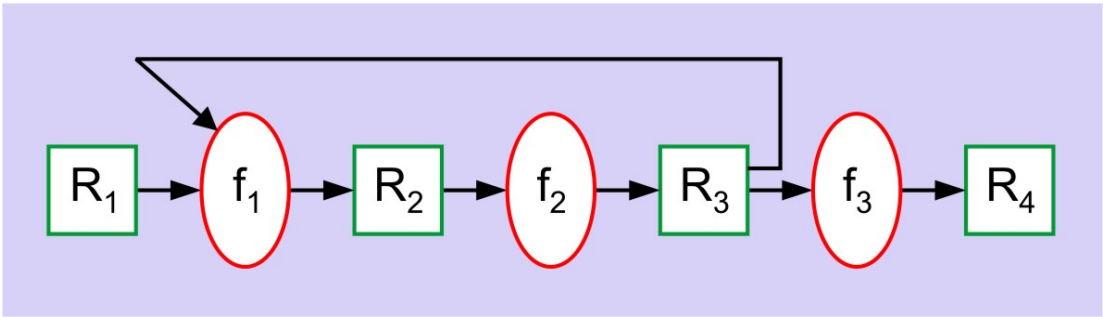
Trennung von kombinatorischer Logik und Registern

- 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)



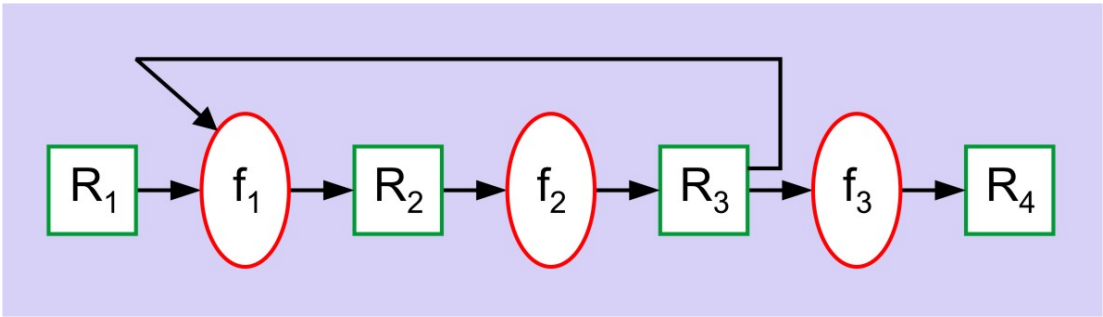
Trennung von kombinatorischer Logik und Registern

- 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)



Trennung von kombinatorischer Logik und Registern

- 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

```
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

```
module alarm_count (
  input  wire        CLOCK,              // Takt
                     RESET,              // Reset
                     INCALARM,           // =1 wenn Taste gedrückt
  output reg  [6:0]  CURRENTCOUT);       // jetziger  Zustand (Alarmanzahl bis 100)

// kombinatorische Uebergangsfunktion, die bei jeder Aenderung des jetzigen
// Zustandes CURRENTCOUT oder der Eingabe INCALARM
// den naechsten Zustand NEXTCOUNT vorlaeufig berechnet

function [6:0]  NEXTCOUNT(
  input  [6:0]  CURRENTCOUT,
  input         INCALARM);

begin
  NEXTCOUNT = CURRENTCOUT;
   if  (INCALARM == 1)       NEXTCOUNT = CURRENTCOUT+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)   CURRENTCOUT <= 0;
else                  CURRENTCOUT <= NEXTCOUNT (CURRENTCOUT, INCALARM);

endmodule
```

3. Möglichkeit: Keine saubere Trennung

Alles in einem **always**-Block

```
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

3. Möglichkeit: Keine saubere Trennung

Alles in einem **always**-Block

```
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

3. Möglichkeit: Keine saubere Trennung

Alles in einem `always`-Block

```
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

Zusammenfassung: Potenzielle Register

Zusammenfassung Definitionen

- 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 Definitionen

- 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 Definitionen

- 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 Definitionen

- 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 Definitionen

- 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 Definitionen

- 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

- 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: Getaktete `always`-Blöcke

- 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: Getaktete `always`-Blöcke

- 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: Getaktete `always`-Blöcke

- 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: Getaktete `always`-Blöcke

- 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: Getaktete `always`-Blöcke

- 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

- 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

Zusammenfassung: Flankenfreie `always`-Blöcke

- 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

Zusammenfassung: Flankenfreie `always`-Blöcke

- 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

Zusammenfassung: Flankenfreie `always`-Blöcke

- 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

Zusammenfassung: Flankenfreie `always`-Blöcke

- 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

`for`-Anweisung

Synthese von `for`-Anweisungen

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

Synthese von `for`-Anweisungen

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

Synthese von `for`-Anweisungen

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

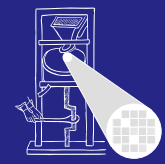
Synthese von `for`-Anweisungen

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

Synthese von `for`-Anweisungen

- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

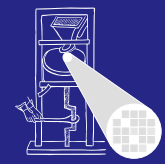
Synthese von `for`-Anweisungen



- 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
```

Synthese von `for`-Anweisungen



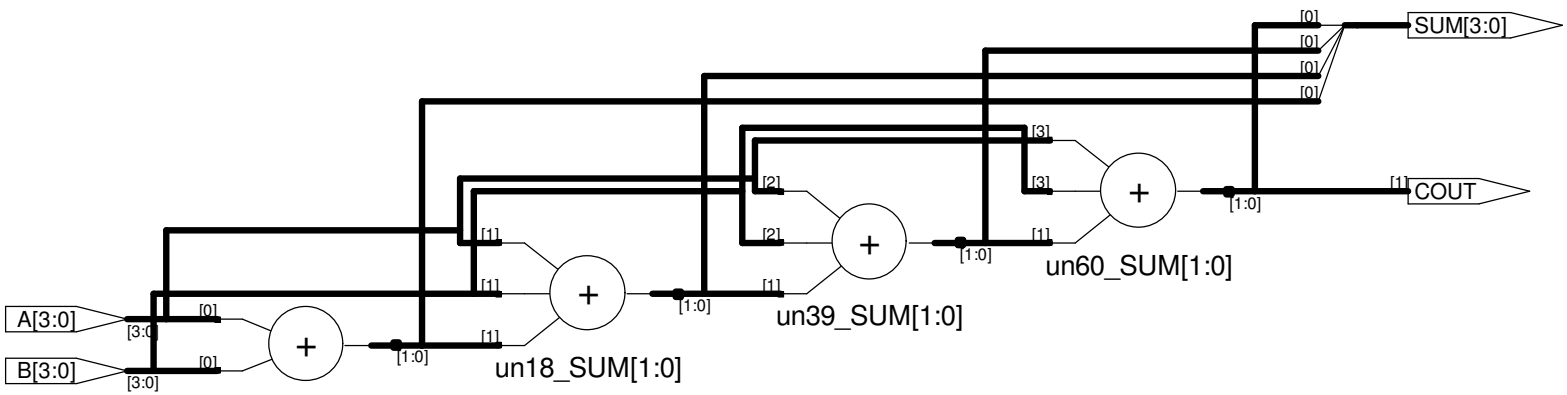
- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

```
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
```

Synthese von `for`-Anweisungen

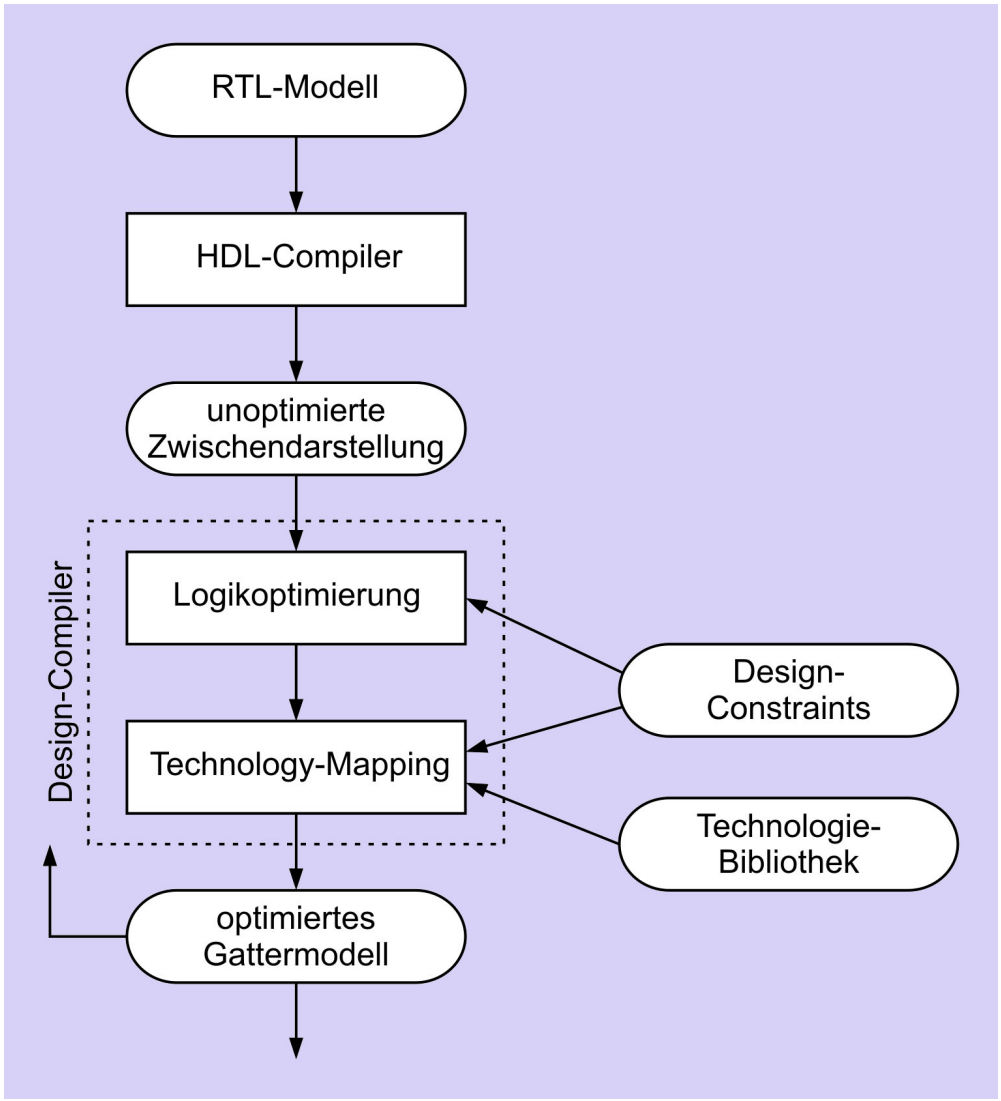
- Nicht als sequentielle Schleife
  - Wie in normaler Programmiersprache
- Stattdessen: Räumlich “ausgerollt”
  - Parallele Abarbeitung

```
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
```

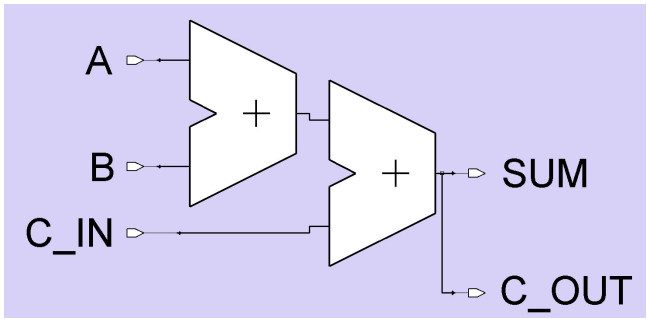


Verfeinerter Ablauf der Synthese

Verfeinerter Entwurfsablauf der Synthese



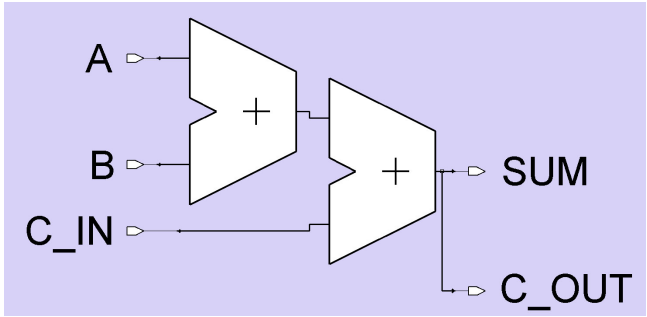
Unoptimierte Zwischendarstellung eines 1b-Addierers



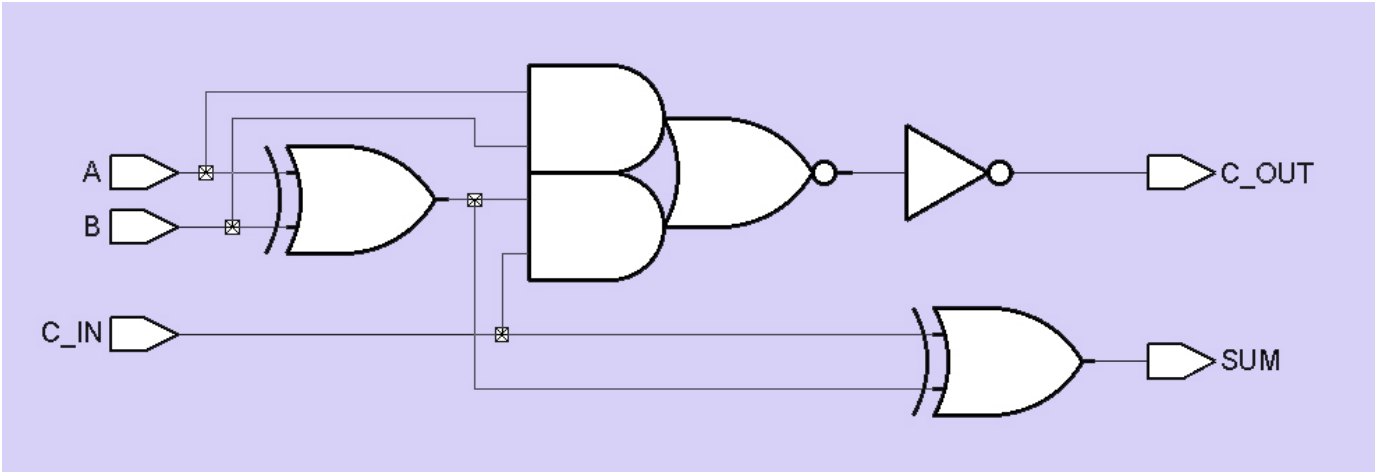
Ergebnis für ASIC-Zielbibliothek (LSI 10K)

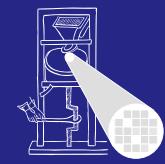
Syntheseablauf: Technologieabbildung
technology mapping

Unoptimierte Zwischendarstellung eines 1b-Addierers

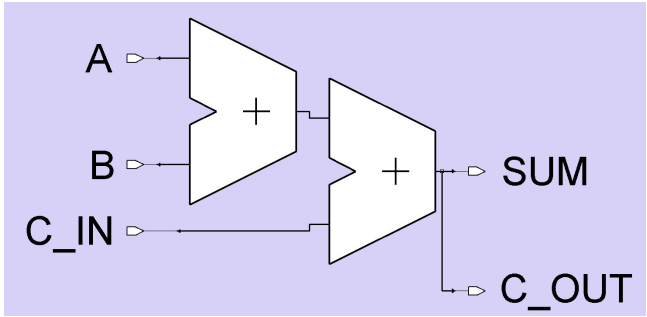


Ergebnis für ASIC-Zielbibliothek (LSI 10K)





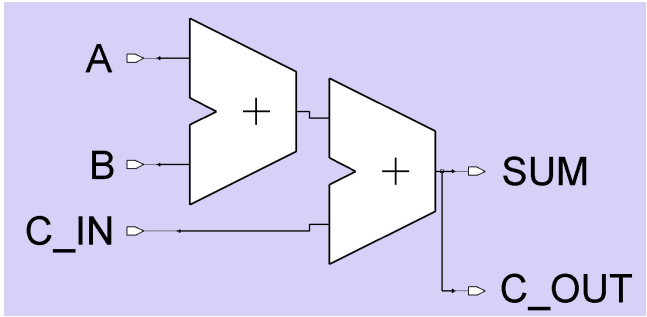
Unoptimierte Zwischendarstellung eines 1b-Addierers



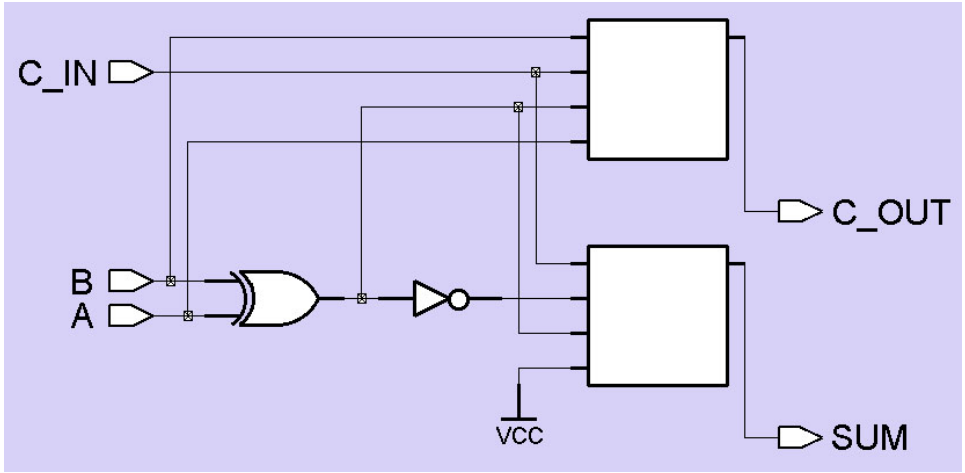
Ergebnis für FPGA (Xilinx XC4000)

Syntheseablauf: Technologieabbildung
technology mapping

Unoptimierte Zwischendarstellung eines 1b-Addierers

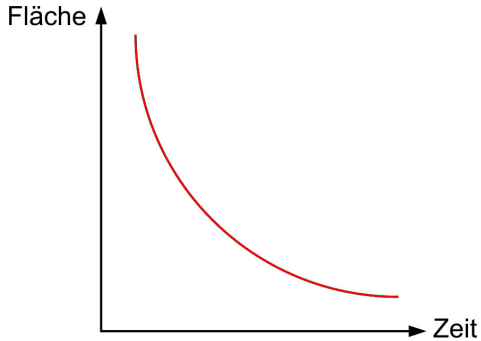


Ergebnis für FPGA (Xilinx XC4000)



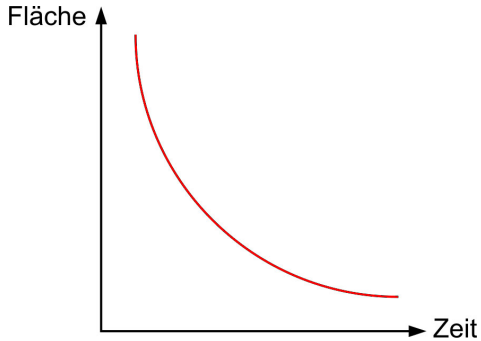
Design-Constraints

- 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



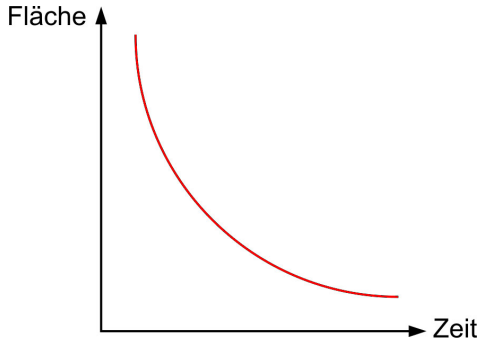
Design-Constraints

- 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



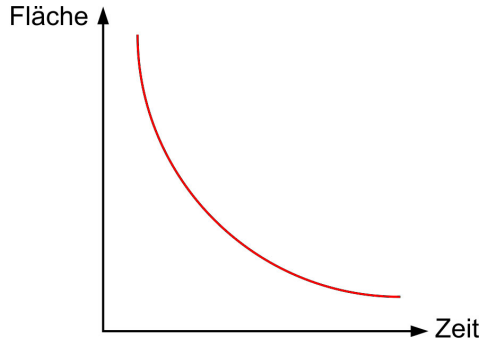
Design-Constraints

- 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



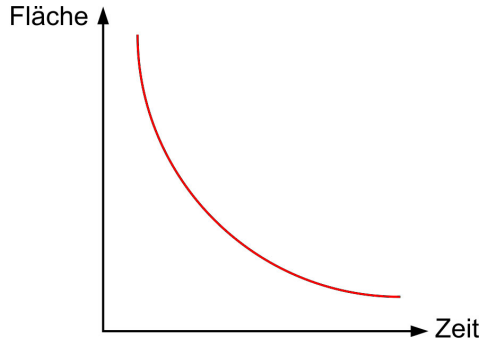
Design-Constraints

- 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



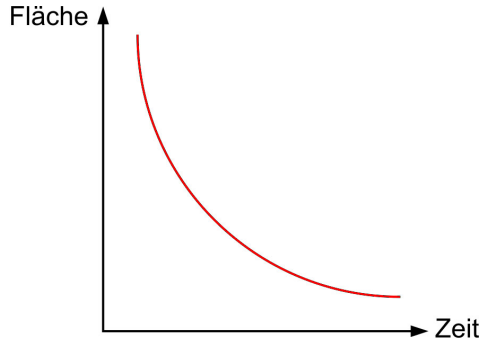
Design-Constraints

- 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



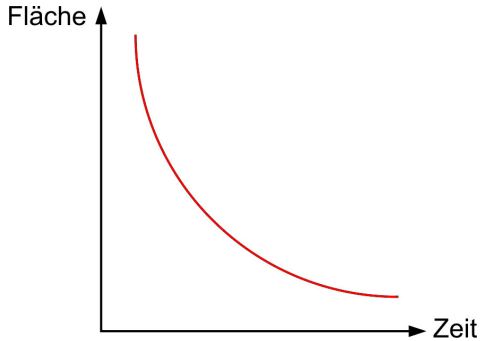
Design-Constraints

- 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



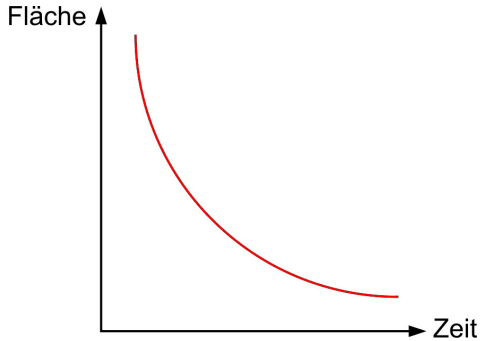
Design-Constraints

- 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



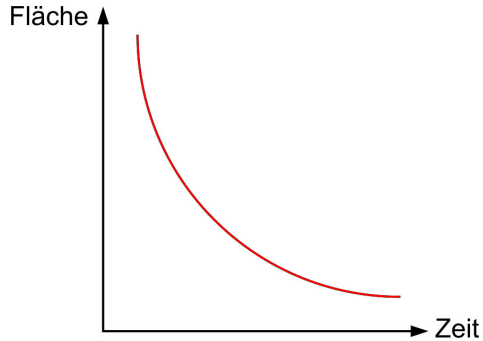
Design-Constraints

- 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



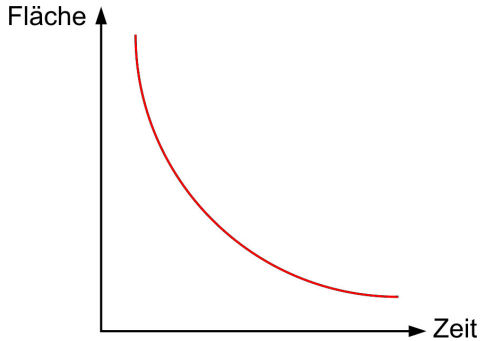
Design-Constraints

- 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



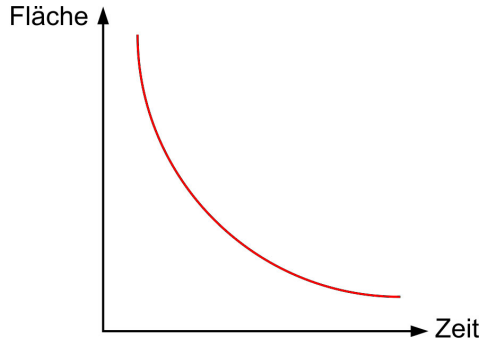
Design-Constraints

- 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



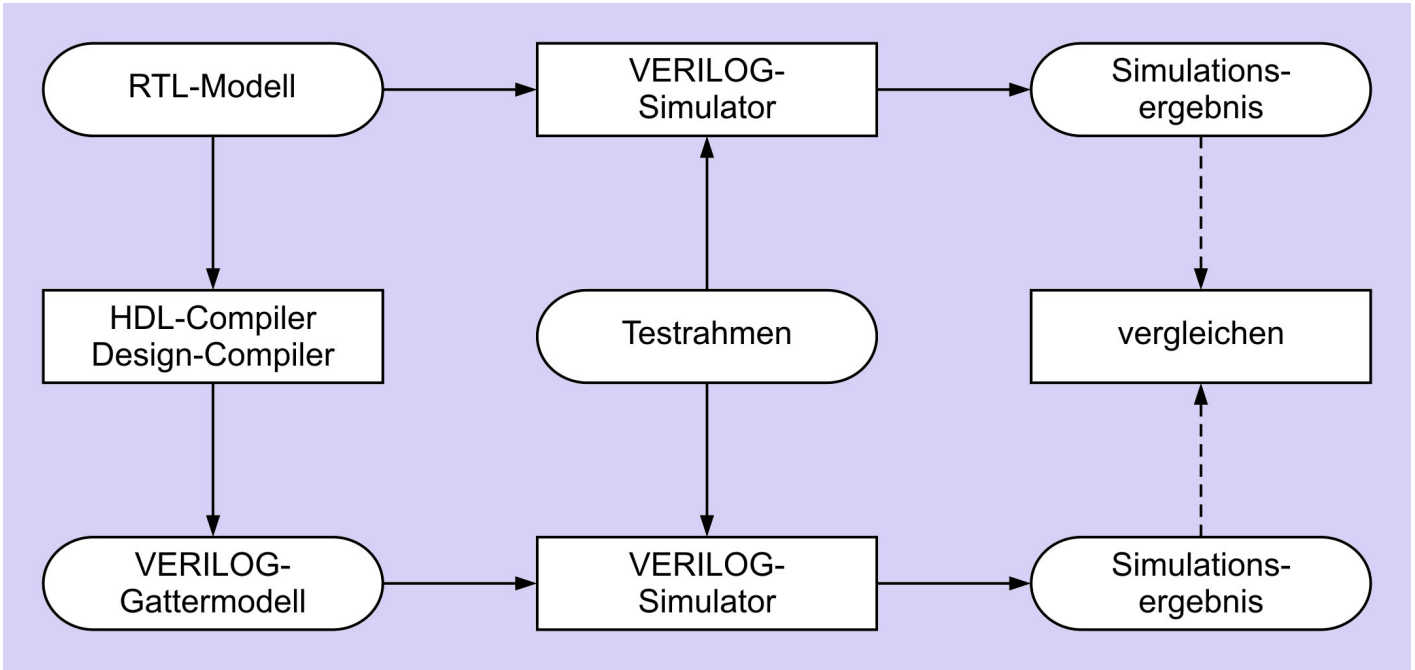
Design-Constraints

- 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



Verifikation

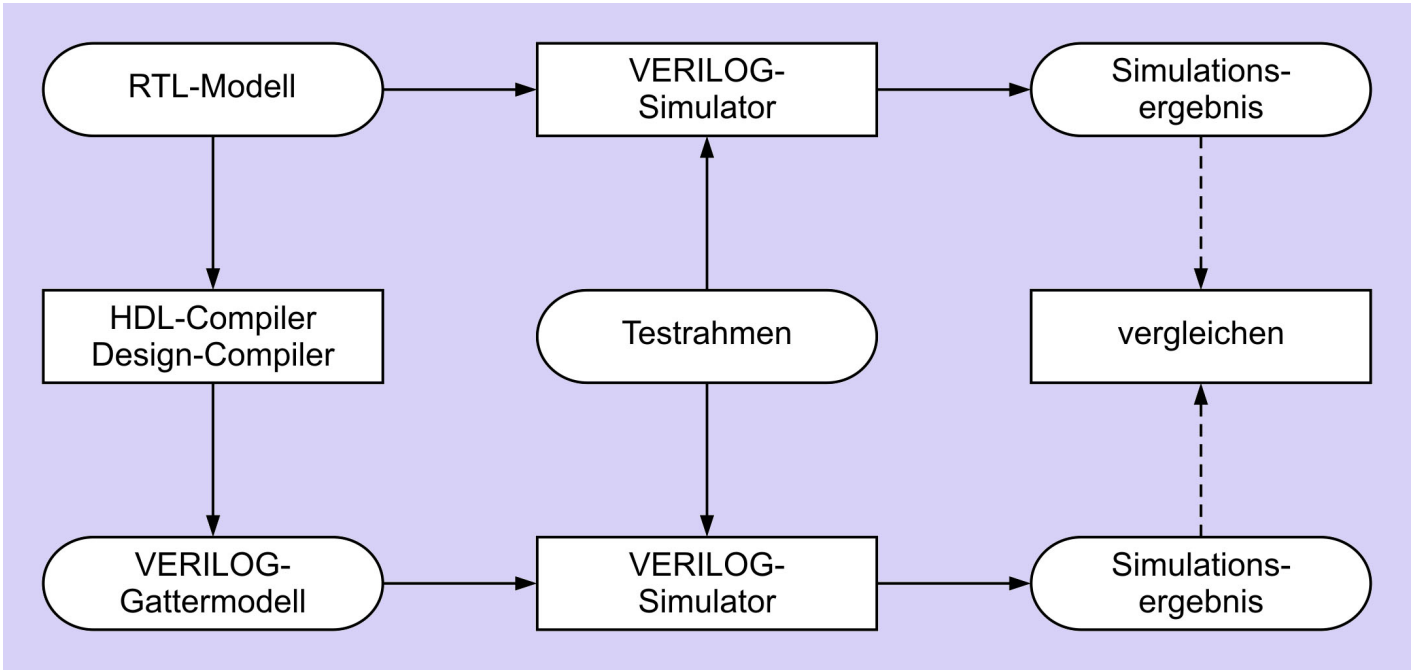
Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



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

Verifikation

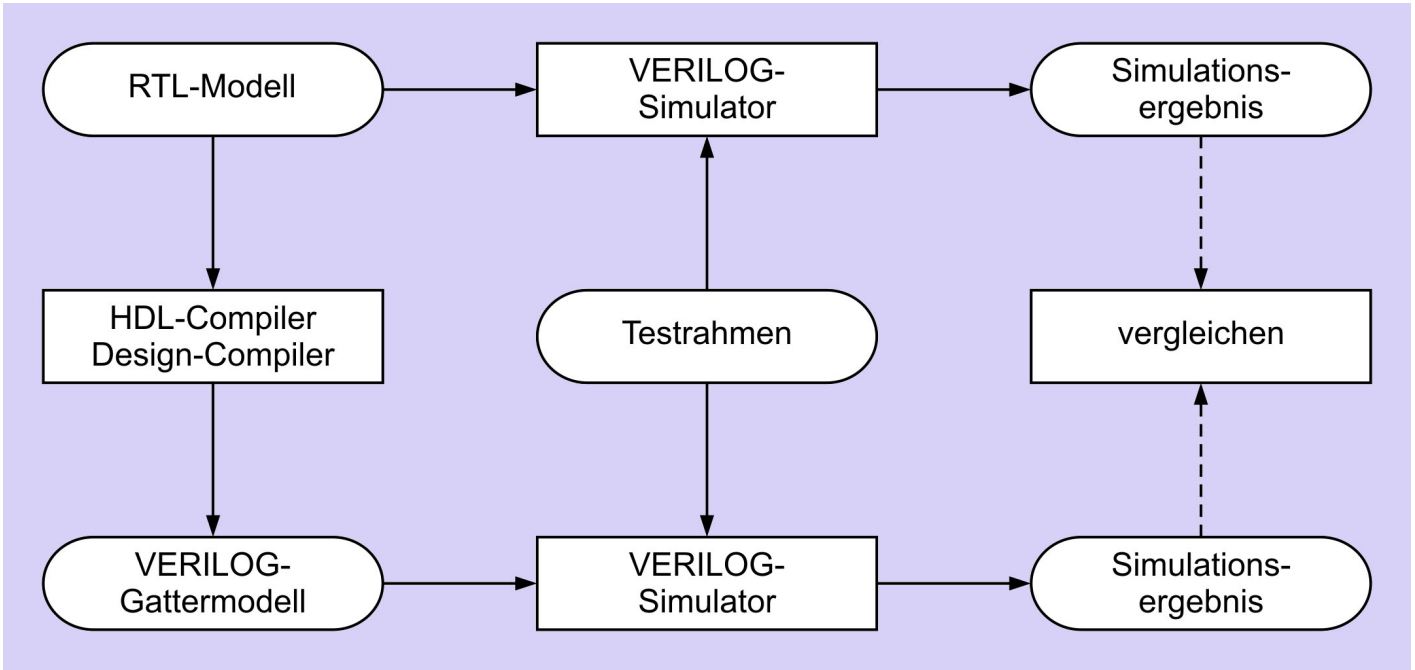
Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



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

Verifikation

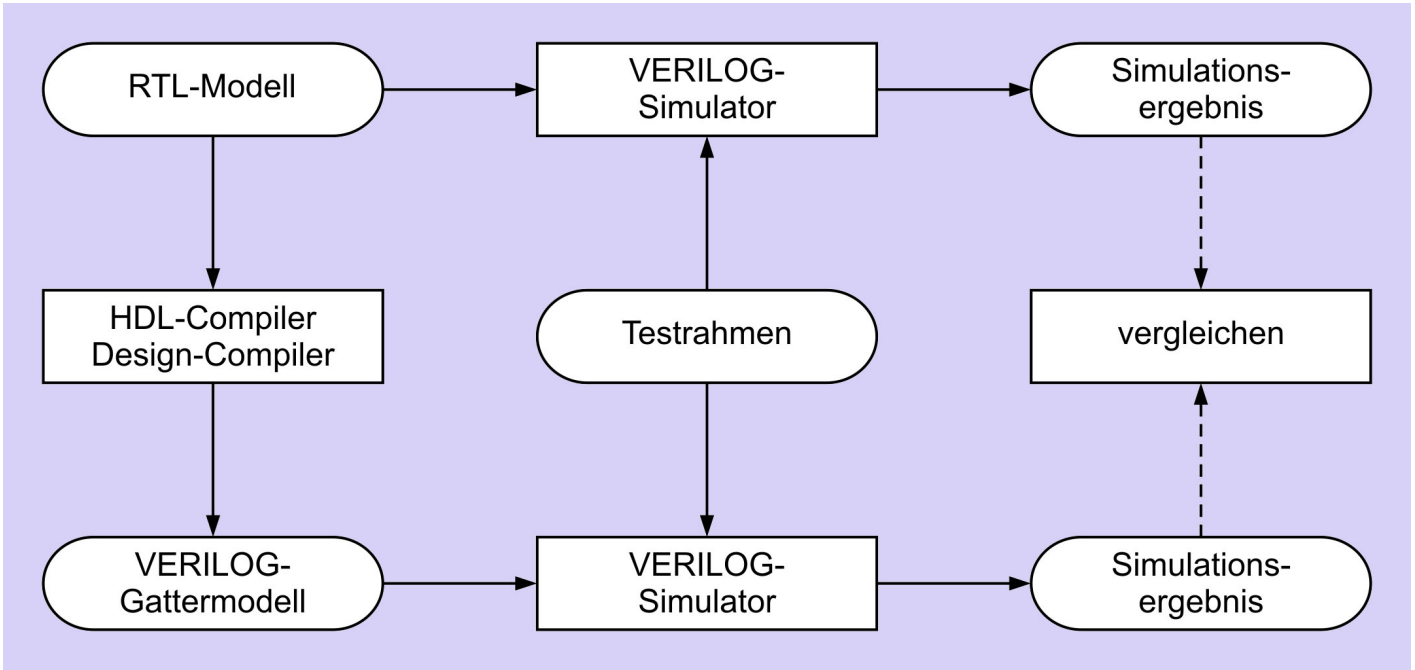
Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



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

Verifikation

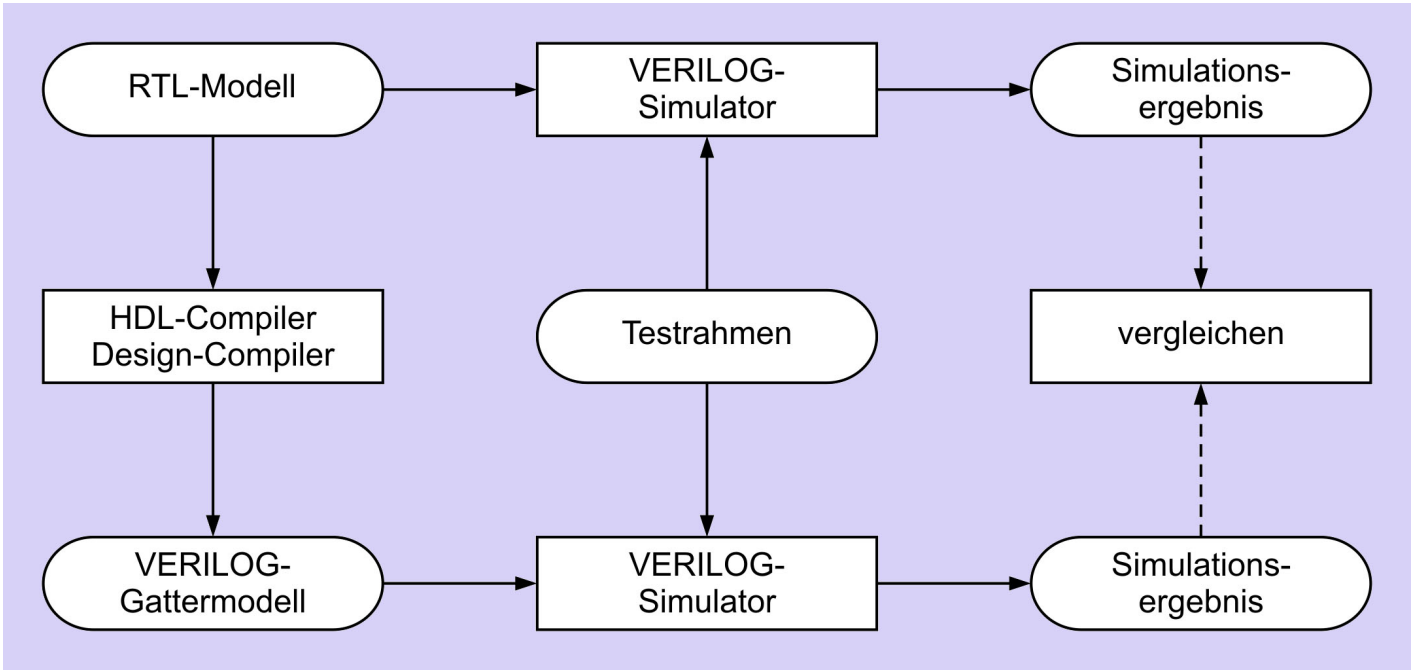
Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



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

Verifikation

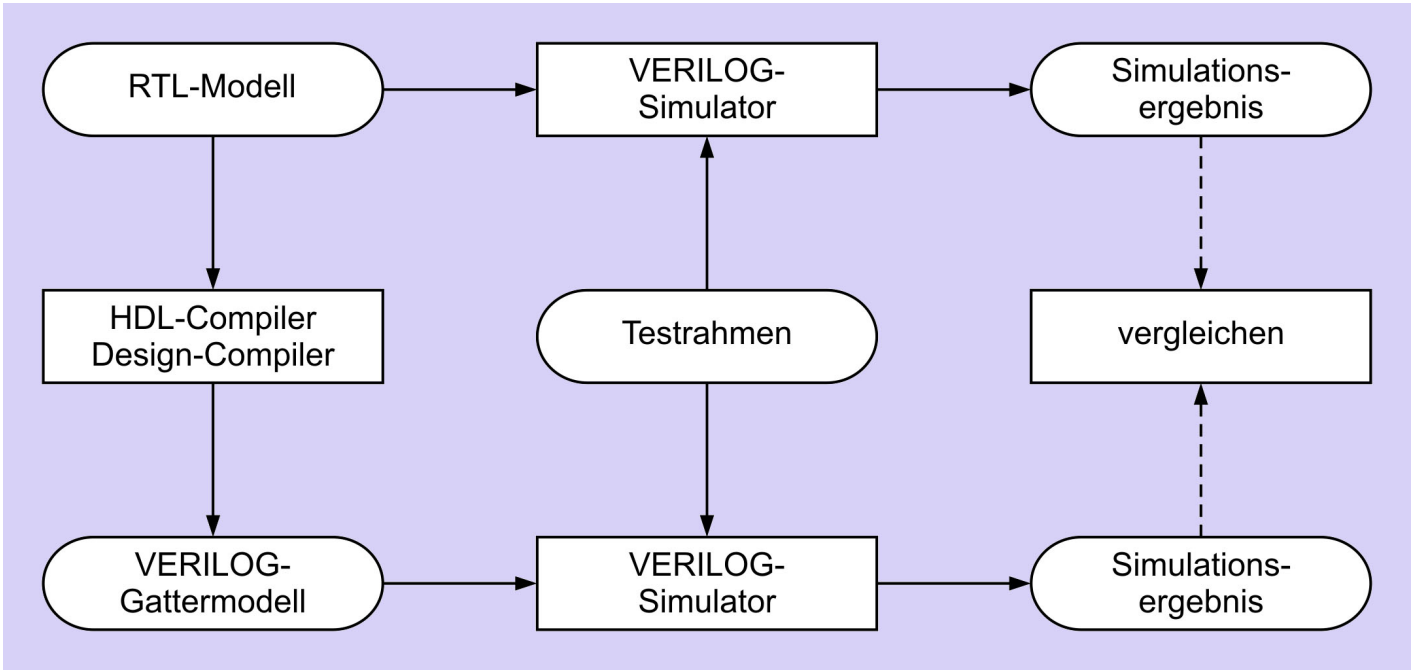
Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



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

Verifikation

Verhält sich synthetisierte Schaltung noch so wie Simulationsmodell?



- Prä-Synthese ./. Post-Synthese-Simulation
- Gleiche Testdaten
  - Bei Post-Synthese aber genauere Tests möglich
  - Hier nun genaueres 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

```
// Vergleicher
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
```

Beispiel: 4b-Vergleicher

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

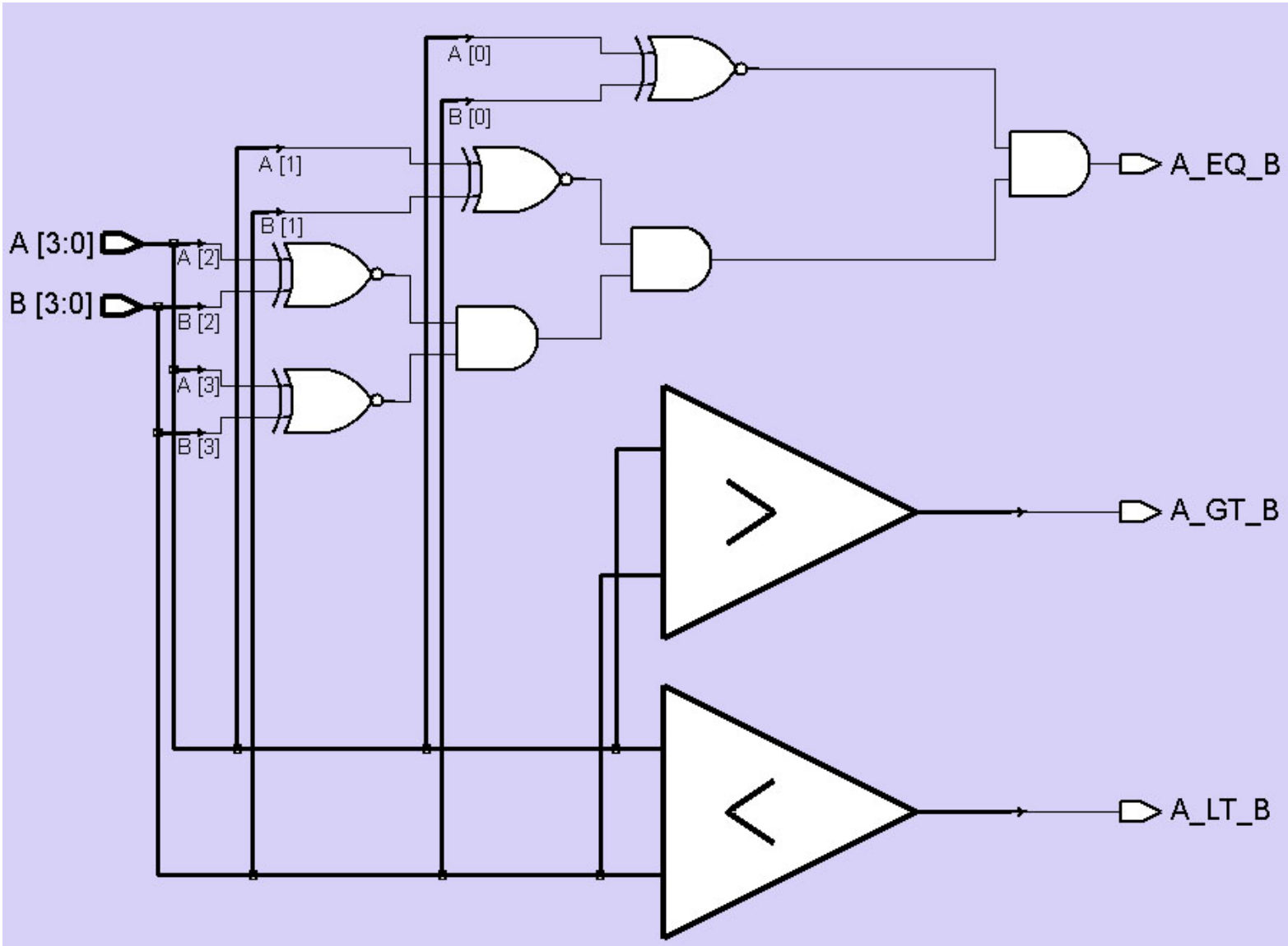
```
// Vergleicher
module mag_comp (
  input  wire [3:0]  A, B,
  output wire        A_GT_B, A_LT_B, A_EQ_B);

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
```

Beispiel: 4b-Vergleicher

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

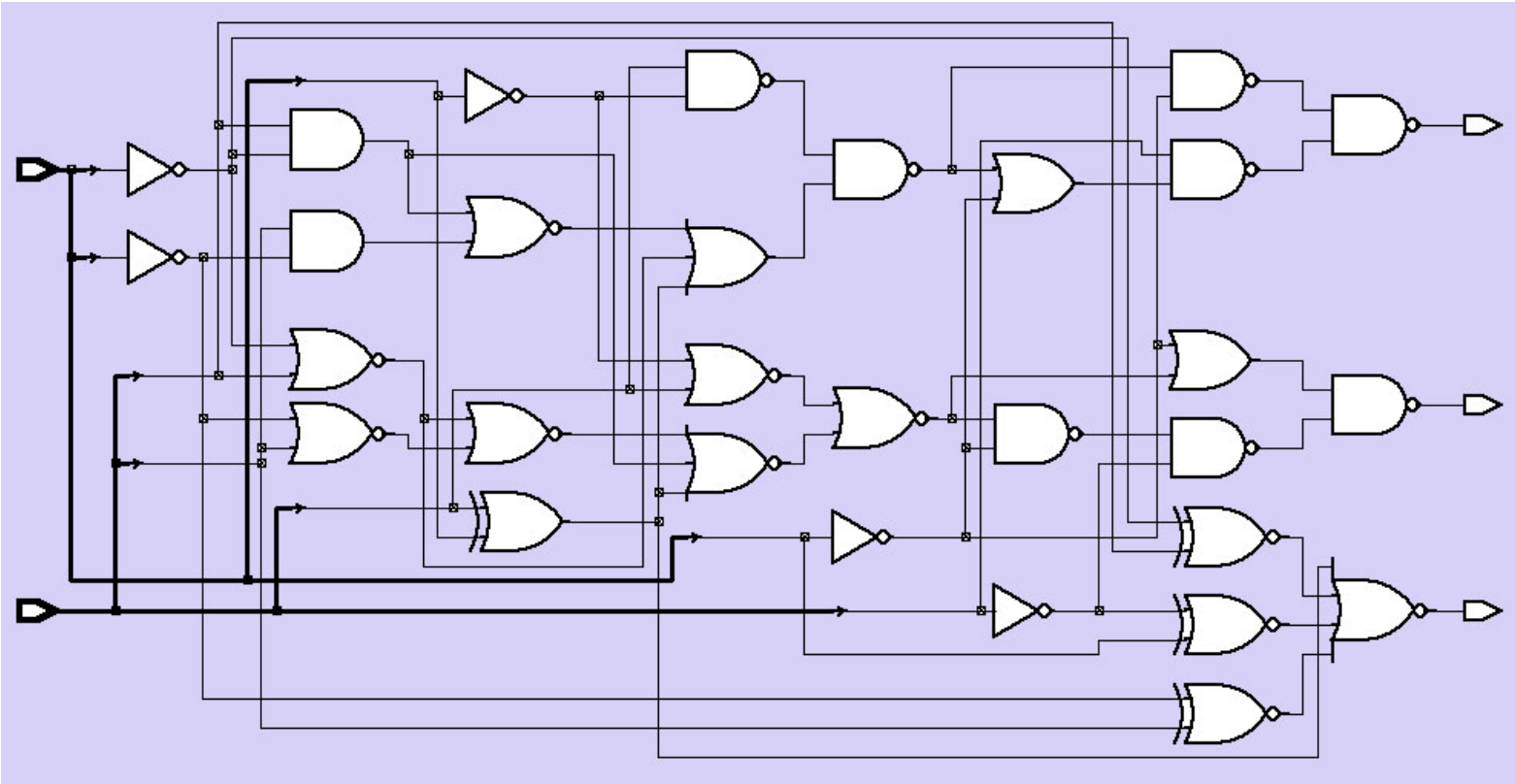
```
// Vergleicher
module mag_comp (
  input  wire [3:0]  A, B,
  output wire        A_GT_B, A_LT_B, A_EQ_B);

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



4b-Vergleicher in LSI Logic 10K ASIC-Technologie

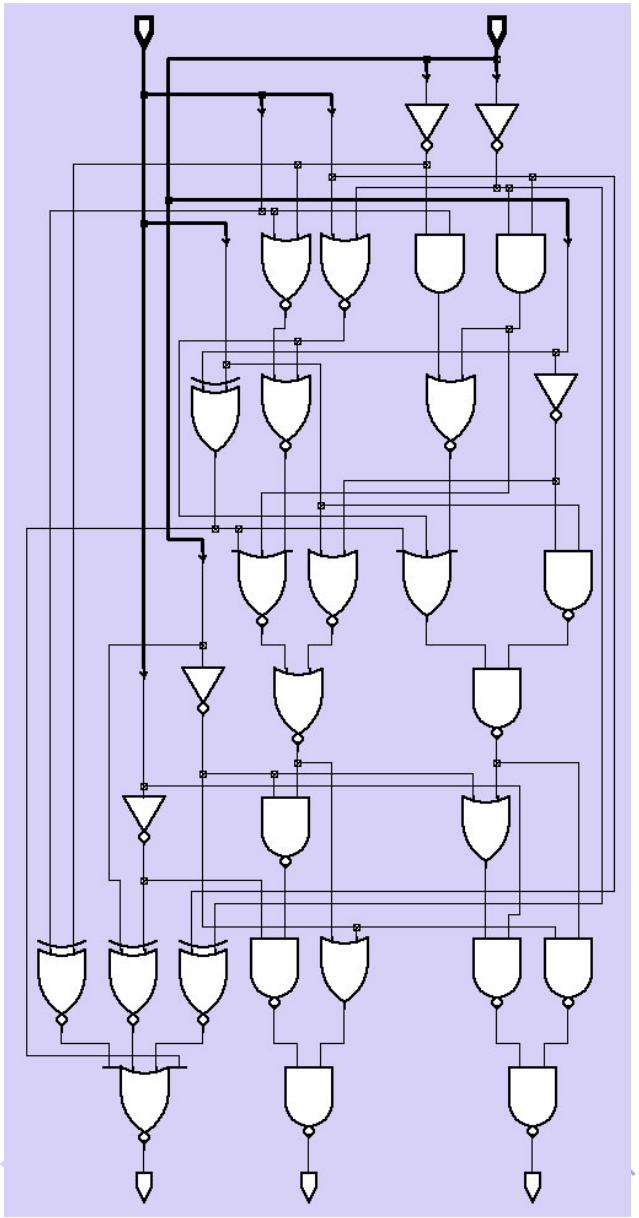
Benutzt um 1993



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
```



Schlüssellochblick in Technologie-Bibliothek

an2: AND-Gatter mit zwei Eingängen

```
// 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

```
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
```

Prä-Synthese-Ergebnis

Test des 4b-Vergleichers: Prä-Synthese

Testrahmen

```
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 eben

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
```

Vergleich: Prä-Synthese mit Post-Synthese

Gleiche Stimuli wie eben

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

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

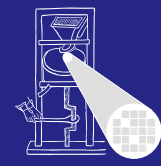
Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation



- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

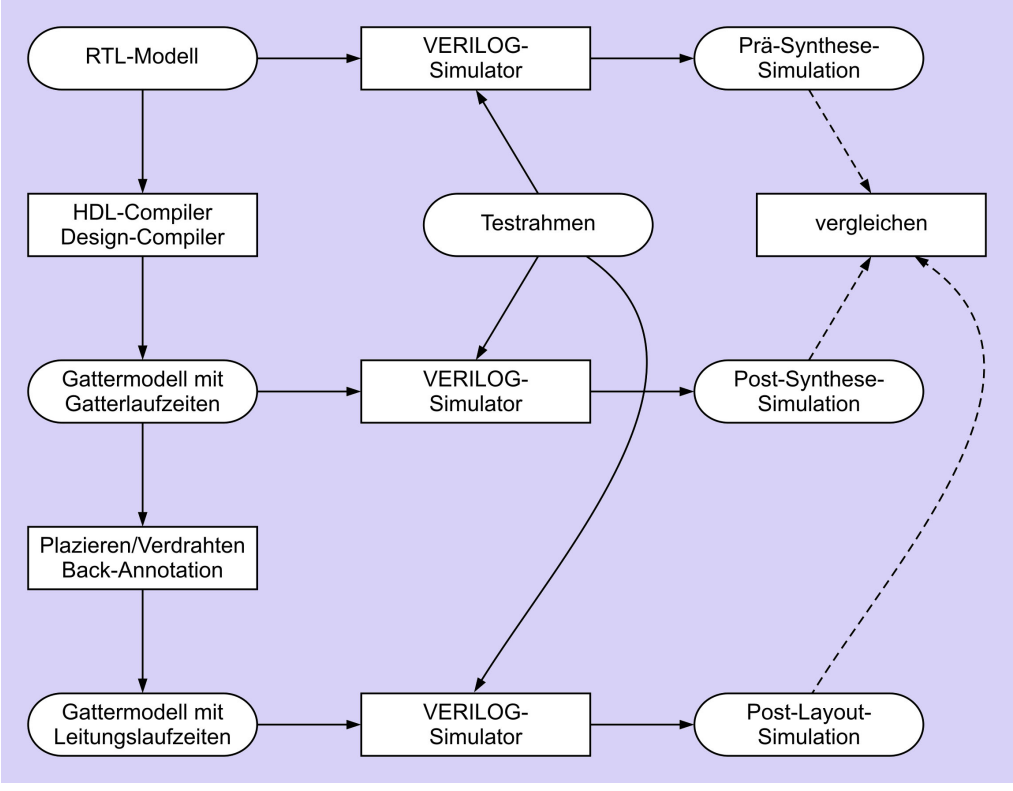
Diskussion: Prä- ./. Post-Synthese-Simulation

- 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 (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Diskussion: Prä- ./. Post-Synthese-Simulation

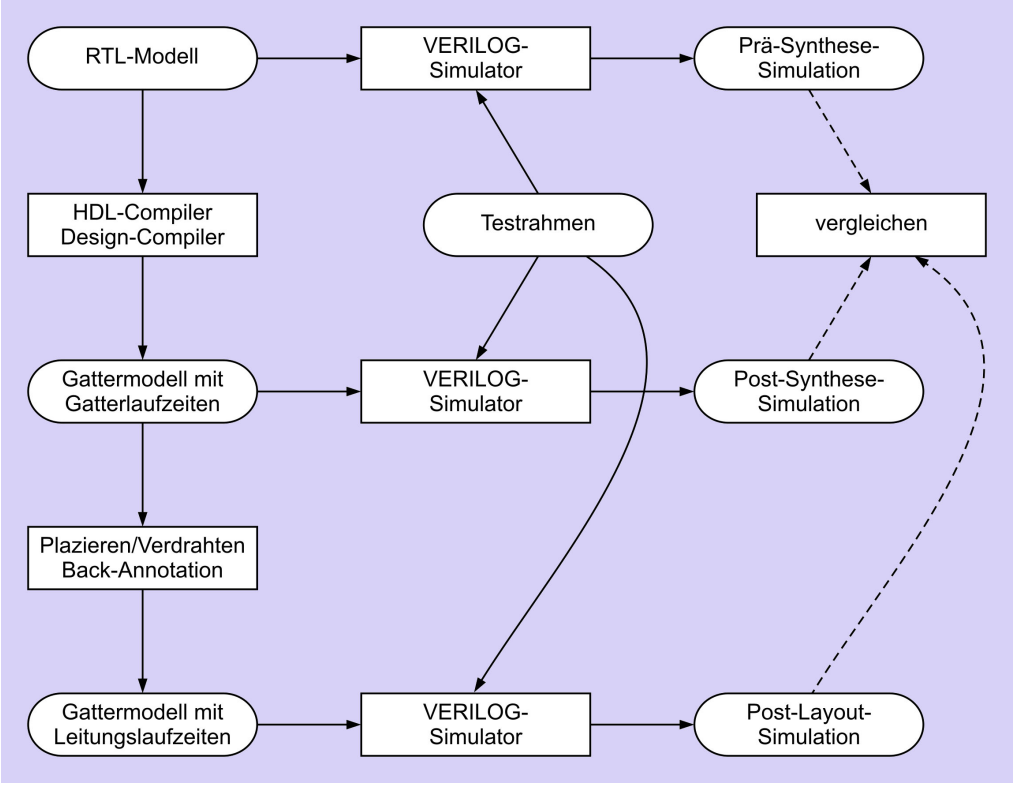
- Unterschiedlich viele Ergebnisse
- Verschiedene Werte
- Unterschiedliche Zeiten
  - Vorher gar keine ausser den im Testrahmen
- Manche Ergebnisse schlicht falsch (z.B. bei t=32)
- Interpretation nötig
  “Wenn man lange genug wartet, ist das Ergebnis richtig”!
- Was ist “lange genug”?
- Antwort: Kritischer Pfad (TGDI2)
- Damit passender Takt für RTL wählbar zwischen
  - Eingangsregistern
  - Ausgangsregistern

Weitere Verfeinerung der Verifikation
Nun bis auf Layout-Ebene



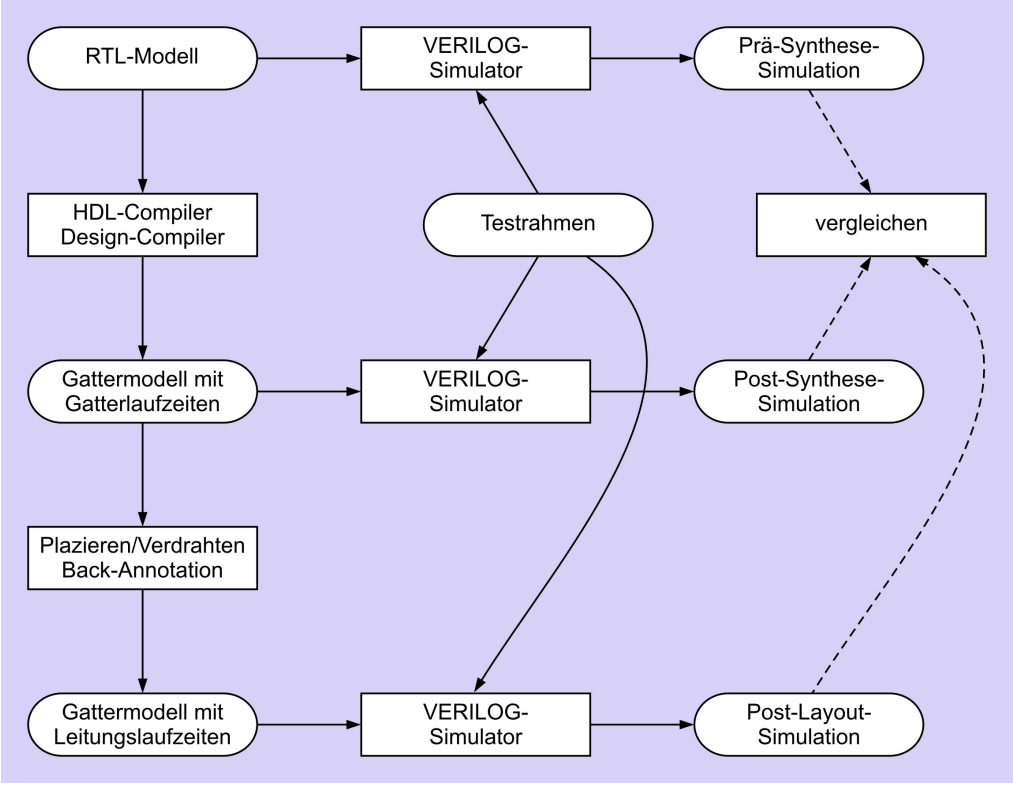
- Post-Layout-Simulation schliesst ein
  - Gatterverzögerungen
  - Leitungsverzögerung
  - Kann umfassen: Widerstände, Kapazitäten, Induktivitäten

Weitere Verfeinerung der Verifikation
Nun bis auf Layout-Ebene



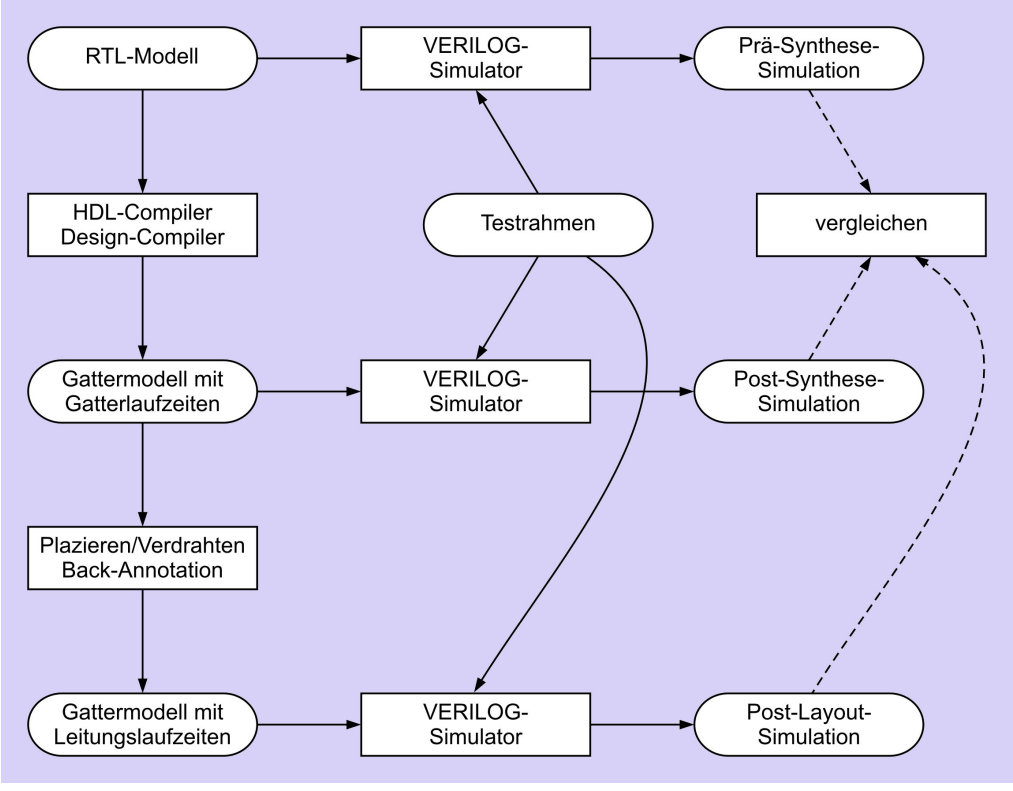
- Post-Layout-Simulation schliesst ein
  - Gatterverzögerungen
  - Leitungsverzögerung
  - Kann umfassen: Widerstände, Kapazitäten, Induktivitäten

Weitere Verfeinerung der Verifikation
Nun bis auf Layout-Ebene



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

Weitere Verfeinerung der Verifikation
Nun bis auf Layout-Ebene



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

Synthesebeispiel: Zero-Counter

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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

Synthesebeispiel: Zero-Counter

- 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: Intuitive Lösung

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

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird

Zero-Counter: Intuitive Lösung

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

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird

Zero-Counter: Intuitive Lösung

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

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird

Zero-Counter: Intuitive Lösung

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

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird

Zero-Counter: Intuitive Lösung

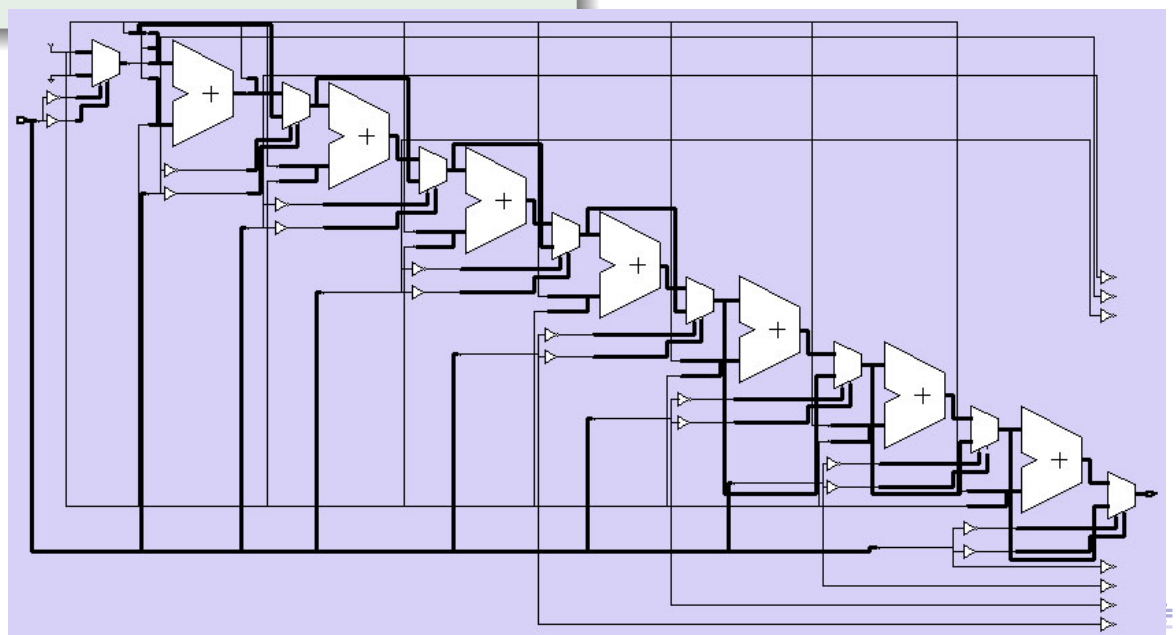
```
module count(
  input  wire [7:0] IN,
  output reg  [3:0] OUT);

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird

Zero-Counter: Intuitive Lösung

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

integer I;

always @(IN) begin
  OUT = 0;
  for (I=0; I<=7; I=I+1)
    if (IN[I]==0) OUT = OUT + 1;
end

endmodule
```

- `for`-Schleife wird räumlich abgerollt
- Addierer-Kaskade
- Multiplexer wählen bei jedem Bit, ob addiert wird



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

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

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

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

Zero-Counter: Vereinfachte Lösung

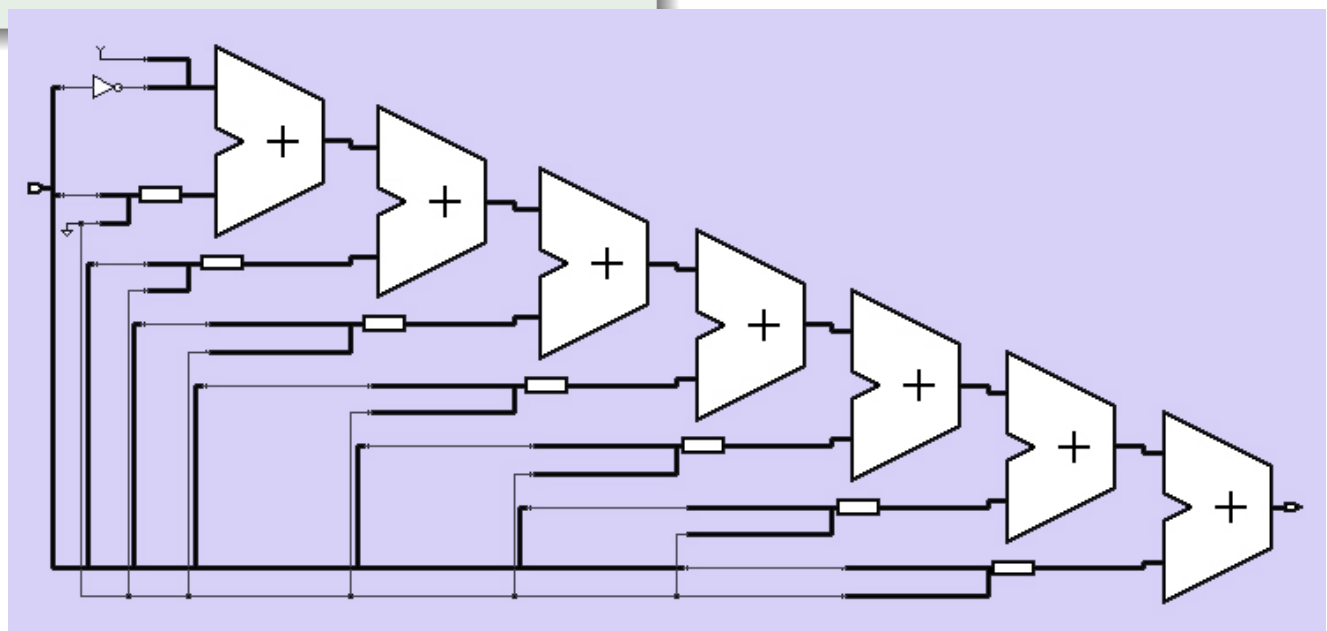
```
module count(
  input  wire [7:0]  IN,
  output reg  [3:0]  OUT);

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

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



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

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

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

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

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

Zero-Counter: Schlaue Lösung

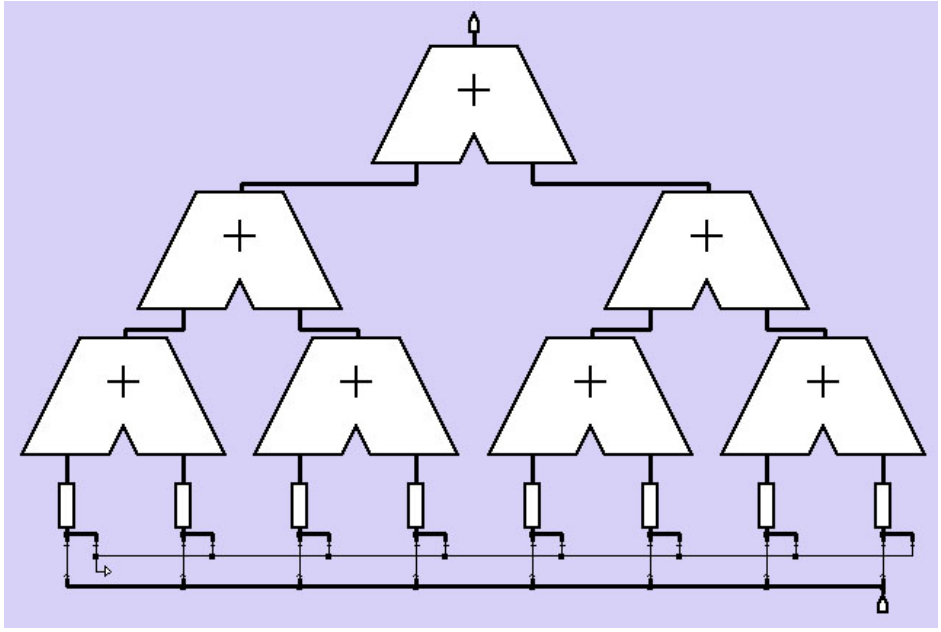
```
module count(
  input  wire [7:0] IN,
  output reg  [3:0] OUT);

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

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



Einfluß von Design-Constraints

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

Einfluß von Design-Constraints

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

Einfluß von Design-Constraints

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

Einfluß von Design-Constraints

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

Einfluß von Design-Constraints

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

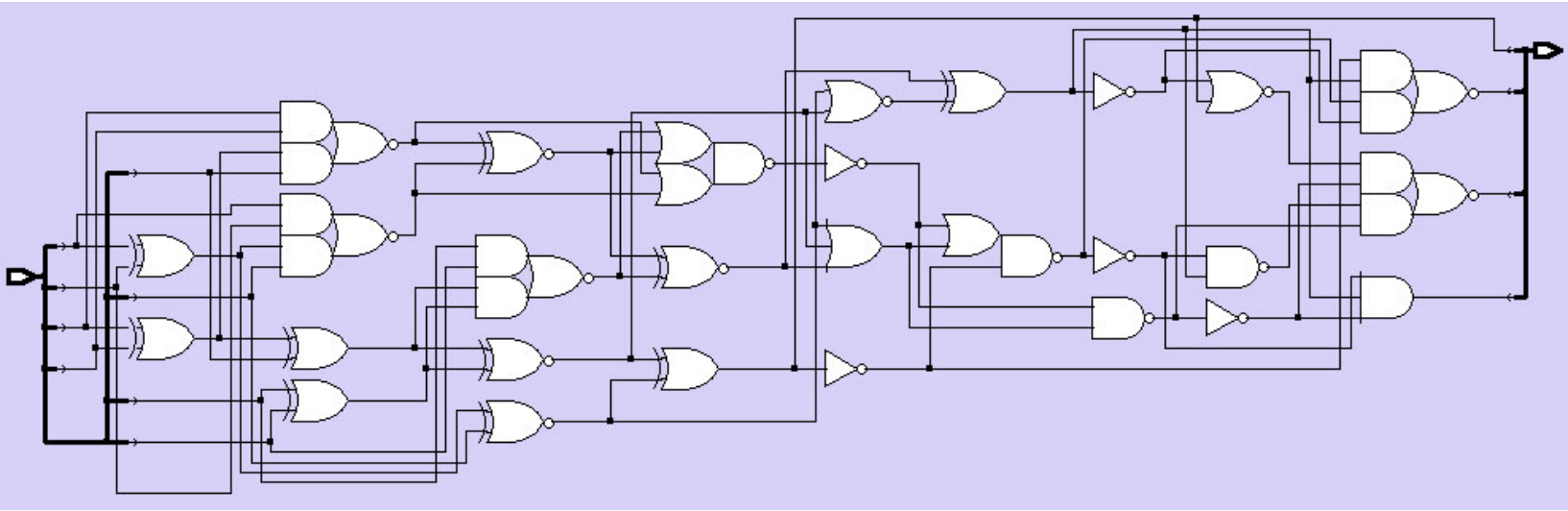
Einfluß von Design-Constraints

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

Einfluß von Design-Constraints

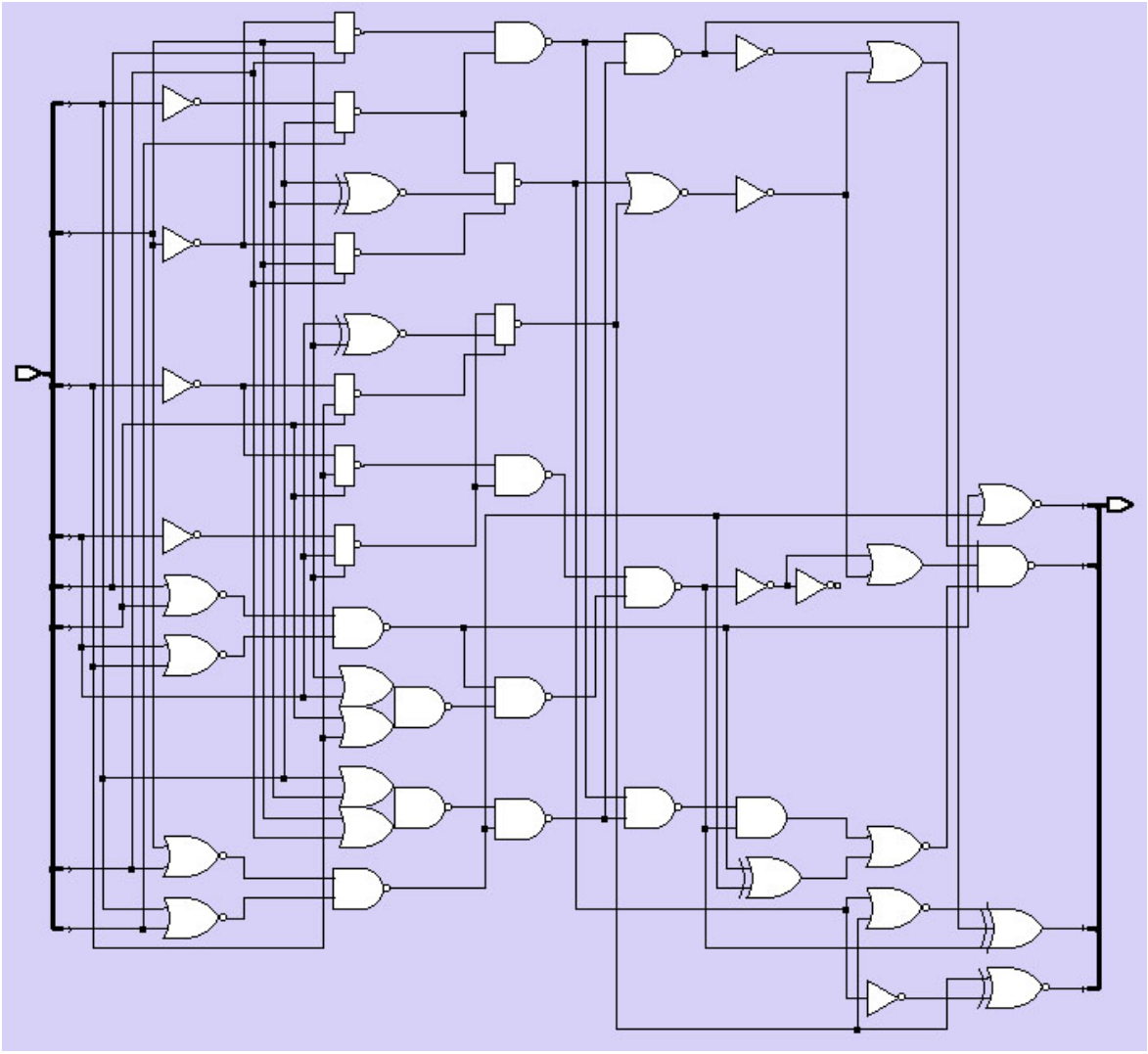
- Festlegen unterschiedlicher Optimierungsziele
- Üblich
  - Flächenbedarf
  - Geschwindigkeit (niedrige Verzögerung)
- Noch seltener
  - Energieverbrauch
  - Ausfallsicherheit

Optimierung auf Fläche
8b-Zero-Counter



52 Gatter, 17.8ns Verzögerung

Optimierung auf Geschwindigkeit
8b-Zero-Counter



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
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer

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
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer

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
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer

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
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer

Zero-Counter: Noch bessere Lösung

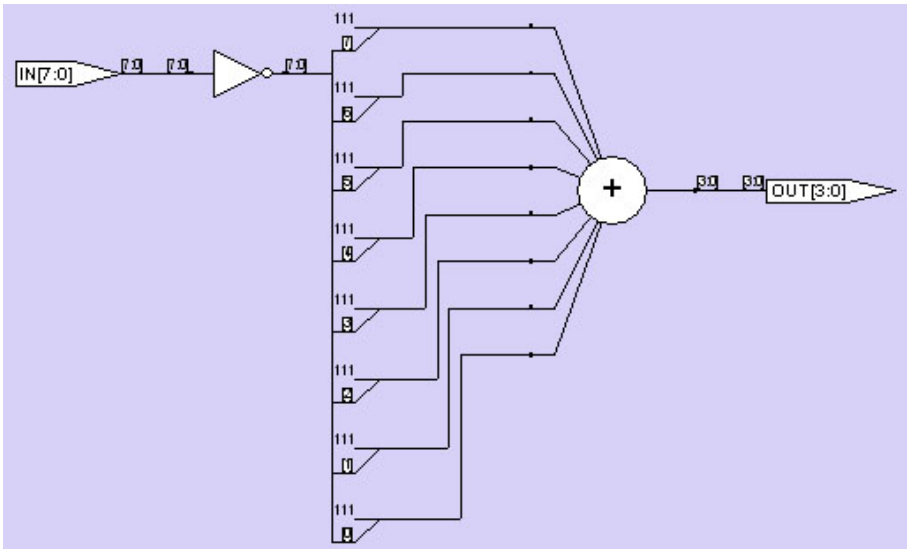
```
module count(
  input  wire [7:0] IN,
  output reg  [3:0] OUT);

always @(IN)
  OUT = ((~IN[0]+~IN[1]) + (~IN[2]+~IN[3]))
      + ((~IN[4]+~IN[5]) + (~IN[6]+~IN[7]));

endmodule
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer

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
```

- Schlaueres Synthesewerkzeug
- Erkennt Natur der Berechnung
- Addiert alle Bits gleichzeitig mit einzelnem 8b-Addierer



Synthesebeispiel: Schaltwerk

Entwurfsbeispiel: Verkaufsautomat

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

Entwurfsbeispiel: Verkaufsautomat

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

Entwurfsbeispiel: Verkaufsautomat

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

Entwurfsbeispiel: Verkaufsautomat

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

Entwurfsbeispiel: Verkaufsautomat

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

Entwurfsbeispiel: Verkaufsautomat

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

Spezifikation des Verkaufsautomaten

- 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

Spezifikation des Verkaufsautomaten

- 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

Spezifikation des Verkaufsautomaten

- 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

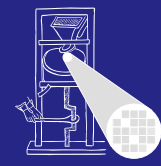
Spezifikation des Verkaufsautomaten

- 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

Spezifikation des Verkaufsautomaten

- 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

Spezifikation des Verkaufsautomaten



- 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

Spezifikation des Verkaufsautomaten

- 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

- Eingabesignal `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`

➥Zustandsautomat in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

- Eingabesignal `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`

➥Zustandsautomat in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

- Eingabesignal `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`

➥Zustandsautomat in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

- Eingabesignal `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`

➥Zustandsautomat in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

- Eingabesignal `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`

➥Zustandsautomat in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

- Eingabesignal `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`

➥Zustandsautomat in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

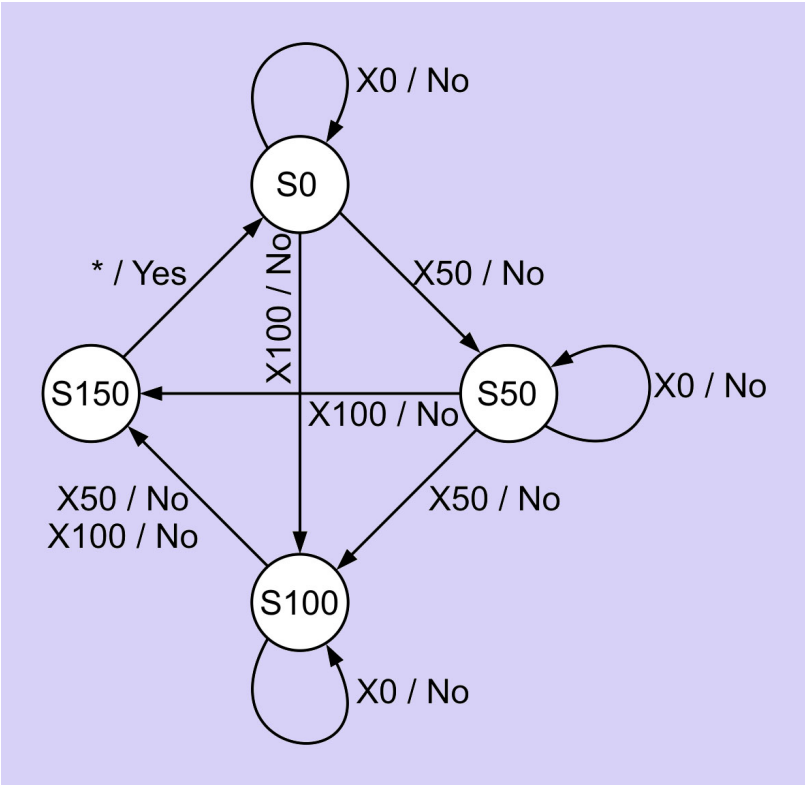
- Eingabesignal `COIN` aus zwei Bits
  - Kann symbolische Werte `X0`, `X50`, `X100` annehmen
- Ausgabesignal `ICE` steuert Eisausgabe
  - Symbolische Werte `Yes` und `No`
- Zustände `S0`, `S50`, `S100`, `S150`

➥Zustandsautomat
in Mealy-Form

Grobentwurf der Steuerung des Verkaufsautomaten

- Eingabesignal **`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`**

➥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
```

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
```

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
```

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
```

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

RTL-Modell: Zustandsspeicher

```
// 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
```

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
```

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
```

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
```

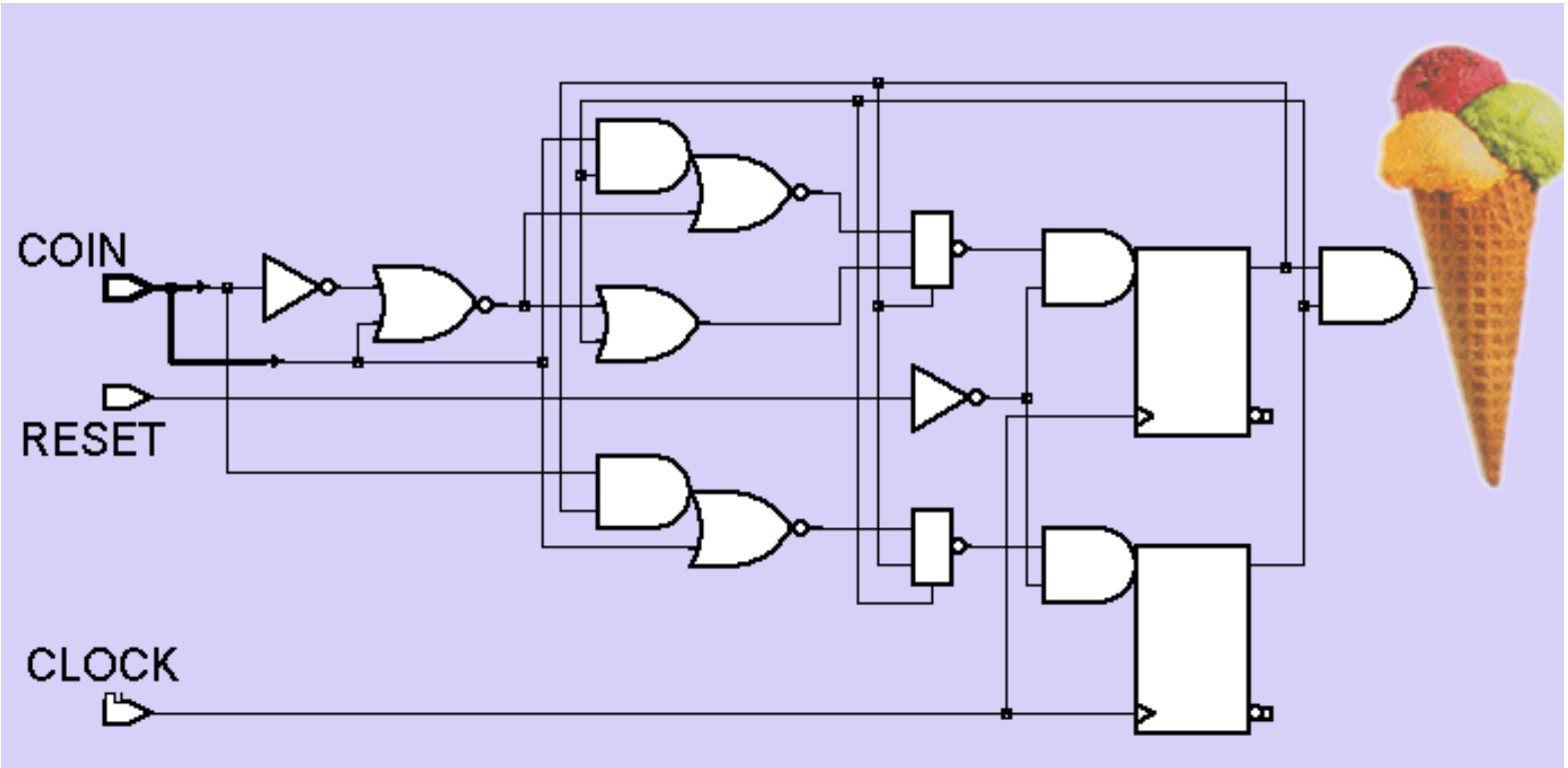
Testrahmen für Verkaufsautomat

```
// 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
```

Simulationsergebnis

```
                    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



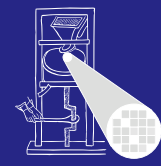
Modellierungsstil

- RTL-Modellierungsstil beeinflußt Gattermodell
  - effizient oder schlecht, im schlimmsten Fall falsch
  - Designer muss eingesetzten Stil genau beobachten
- Zielkonflikt
  - Abstraktes RTL-Modell
    - Vorgestellte Synthesewerkzeug soll sich um Details kümmern
    - Nur kann gelegentlich unerwünschte Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Aufwändig, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigkeit von Zieltechnologie kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Synthesezellen

Modellierungsstil

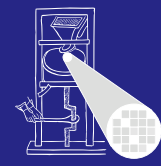
- RTL-Modellierungsstil beeinflußt Gattermodell
  - effizient oder schlecht, im schlimmsten Fall falsch
  - Designer muss eingesetzten Stil genau beobachten
- Zielkonflikt
  - Abstraktes RTL-Modell
    - Vorgestellte Synthesewerkzeug soll sich um Details kümmern
    - Nur kann gelegentlich ineffizienten Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Aufwändig, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigkeit von Zieltechnologie kann verloren gehen
    - insbesondere bei direkter Instanziierung von Synthesezellen

Modellierungsstil



- RTL-Modellierungsstil beeinflußt Gattermodell
  - effizient oder schlecht, im schlimmsten Fall falsch
  - Designer muss eingesetzten Stil genau beobachten
- Zielkonflikt
  - Abstraktes RTL-Modell
    - Vorgestellte Synthesewerkzeug soll sich um Details kümmern
    - Nur kann gelegentlich ineffizientes Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Muß sehr präzise geschrieben
    - gute Kenntnis über spätere Schaltungsstruktur
    - Unabhängigkeit von Zielarchitektur kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Synthesezellen

Modellierungsstil



- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - 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

Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - 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

Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - 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

Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - 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

Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Mühsam, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigheit von Zieltechnologie kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Spezialblöcken

Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - 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

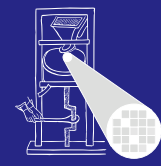
Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Mühsam, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigheit von Zieltechnologie kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Spezialblöcken

Modellierungsstil

- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Mühsam, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigheit von Zieltechnologie kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Spezialblöcken

Modellierungsstil



- 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
    - Nur kann gelegentlich unerfreuliche Hardware produzieren
  - Hardware-nahes RTL-Modell
    - Mühsam, ineffizient zu schreiben
    - gute Kontrolle über spätere Schaltungsstruktur
    - Unabhängigheit von Zieltechnologie kann verloren gehen
    - Insbesondere bei direkter Instanziierung von Spezialblöcken