Grundlagen der Digitaltechnik Foliensatz 3: Synthese und Schaltungsoptimierung

G. Kemnitz

Institut für Informatik, TU Clausthal (EDS_F3)

23. Februar 2021

Inhalt F3: Synthese und Schaltungsoptimierung

Synthese

Synthese

Synthese versucht, für eine Funktionsbeschreibung eine funktionsgleiche Schaltung zu finden. Als lösbar gelten:

1. die Synthese einer Schaltung ohne Gedächtnis, mit derselben logischen Verarbeitungsfunktion unter Vorgabe einer maximalen Verzögerungszeit.
2. die Synthese für Register-Transfer-Funktionen unter Vorgabe der Mindesttaktfrequenz.
3. die Generierung laufzeitkritischer Schaltungen für spezielle Funktionen und
4. Handentwurf und Einbindung als Teilschaltung.

Für 1 und 2 ist die Funktion in VHDL ohne Zeitverhalten zu beschreiben (ohne After-Klausel, Warteanweisungen, ...).

Bei 3 und 4 erfolgt die Einbindung der generierten oder vorentworfenen Schaltung in VHDL als Instanz.

Die Beschreibung der Zeit-Constraints (max. Verzögerung, Mindesttaktfrequenz) erfolgt in unserem System in der ucf-Datei (User Constraint File) in einer von VHDL abweichenden Sprache.

Zeit-Constraints werden befriedigt, indem bei Verletzung die Synthese mit anderen Steuerparametern neu startet. Überflüssige Constraints vergrößern der Rechenaufwand.

Die Beschränkung auf gebräuchliche Beschreibungsmuster ist dringend zu empfehlen. Denn Synthese ist ein schlecht gestelltes Problem. Geringe Beschreibungsänderungen eröffnen sprunghaft neue Interpretations- und Fehlermöglichkeiten.

Nach der Synthese, Platzierung und Verdrahtung kann mit einer Post-Place-and-Route-Simulation überprüft werden, ob das Syntheseergebnis mit den daraus abschätzbaren Verzögerungen verspricht zu funktionieren.

VHDL-Beschreibungsmittel für die Synthese

- Prozesse zur Beschreibung von Verarbeitungsfunktionen
  - ohne Verzögerung, ohne Zuweisung ungültiger Werte,
  - ohne Warteanweisungen, ...
- Prozesse zur Beschreibung von Register-Transfer-Funktionen
  - ohne Verzögerungen,
  - ohne Zuweisung ungültiger Werte, ...
- Instanziierung handentworfener oder generierter Schaltungen.

Zulässige Datentypen:

- Bittypen: `bit`, `boolean`, `std_logic`
- Bitvektortypen: `bit_vector`, `std_logic_vector`, `unsigned`, `signed`
- Zahlentypen: `integer`, `natural`, `positive`
- selbst definierte Aufzählungstypen für Bitvektorwerte,
- Feld und Verbund aus diesen Typen.

Beschreibungsmittel für das logische Verhalten von Verarbeitungsfunktionen (Abbildung Eingaben → Ausgaben ohne Gedächtnis):

- bitorientierte Ausdrücke mit den Operatoren **not**, **and**, **nand**, ...
- bitvektororientierte Ausdrücke mit Logik-, Verschiebe-, Vergleichs- und den arithmetischen Operatoren (+, -, *)[1].
- Kontrollstrukturen: Fallunterscheidung, Generierungsschleifen, Unterprogramme.

Die Synthese bildet die Funktion einer so beschriebenen Schaltung aus Gattern, Multiplexern, Rechenwerken, Komparatoren und Registern nach und optimiert dann.

Die Gatter, Multiplexer, Rechenwerke, ... werden durch vorentworfene Bausteine nachgebildet oder von Schaltungsgeneratoren aus diesen zusammengesetzt.

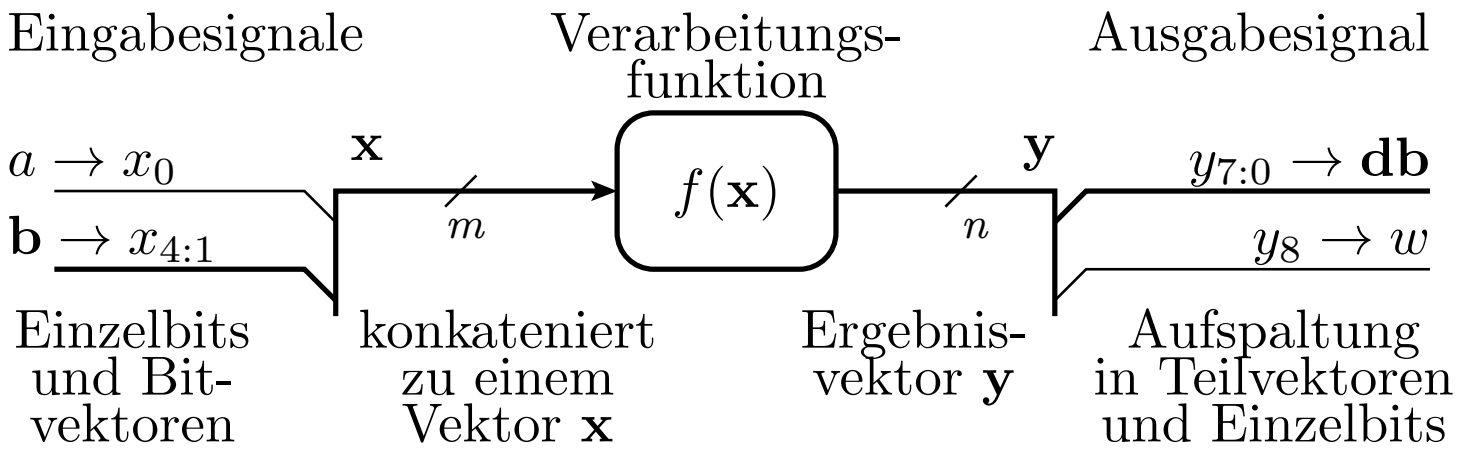
[1] Arithmetischen Operatoren nur `signed`, `unsigned`, `integer`, `natural`, `positive` mit WB-Beschränkung. Keine Division, siehe später Foliensatz F4.

Verarbeitungsfunktionen

Digitale Verarbeitungsfunktionen

Eine digitale Verarbeitungsfunktion bildet ganz allgemein ein m-Bit-Eingabesignal $\mathbf{x}$ auf ein n-Bit-Ausgabesignal $\mathbf{y}$ ab[2].

Beide Signalvektoren können sich aus Einzelbits oder Teilbitvektoren zusammensetzen.



[2] Außer der Zielfunktion $f(\mathbf{x})$ können Obergrenzen für die Verzögerung, den Stromverbrauch, ... vorgegeben sein.

Beschreibungsschablonen in VHDL

Das Beschreibungsmodell einer Verarbeitungsfunktion für die Synthese ist ein Prozess mit allen Eingabesignalen in der Weckliste, der bei jedem Durchlauf allen Ausgabesignalen einen neuen Wert ohne After-Klausel zuweist. Die Synthese übersetzt:

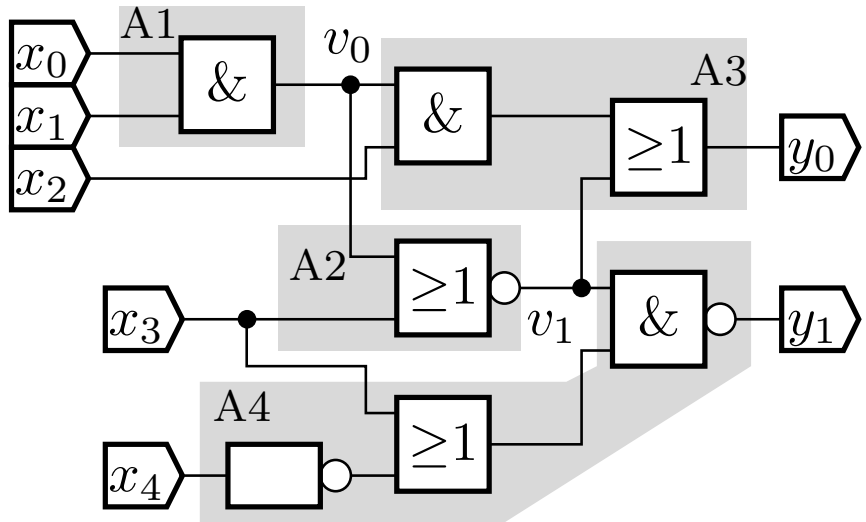
- logische Operatoren in Gatter,
- arithmetische Operatoren in Rechenwerke,
- Vergleichsoperatoren in Komparatoren und
- Fallunterscheidungen in Multiplexer.

Zwischenergebnisse sind in Variablen zu speichern[3]. Unterprogramme werden Teilschaltungen und Schleifen werden aufgerollt. Aufteilung in mehrere Prozesse ist möglich. Die Synthese übersetzt Prozesse einzeln und fügt die Schaltungen zusammen.

[3] Signale übernehmen zugewiesene Werte erst nach Weiterschalten der Simulationszeit. Ohne Verzögerungsangabe, wenn der Prozess schläft.

Beschreibung & Extraktion einer Gatterschaltung

```
signal x: std_logic_vector(4 downto 0);
signal y: std_logic_vector(1 downto 0);
...
process(x)
 variable v: std_logic_vector(1 downto 0);
begin
 A1: v(0):= x(0) and x(1);
 A2: v(1):= v(0) nor x(3);
 A3: y(0)<=(v(0) and
     x(2)) or v(1);
 A4: y(1)<=((not x(4))
     or x(3)) nand v(1);
end process;
```

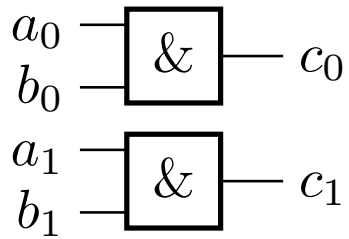


Ausführungsreihenfolge, Bitvektoren

VHDL hat die logischen Operatoren **not**, **and**, **nand**, **or**, **nor**, **xor** und **xnor**. **Not** hat Vorrang. Die Ausführungsreihenfolge der zweistelligen Operationen ist mit Klammern festzulegen.

Außer auf die Bittypen `bit`, `boolean` und `std_logic` können Bitoperatoren bitweise auf die Vektortypen `bit_vector`, `std_logic_vector`, `unsigned` und `signed` angewendet werden.

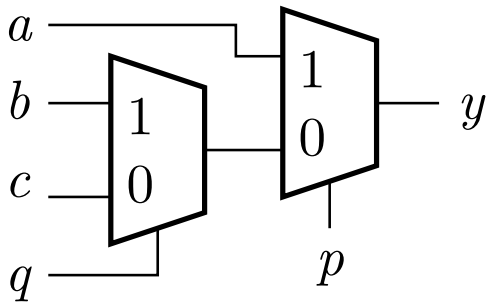
```
signal a, b, c: std_logic_vector(1 downto 0);
...
process(a, b)
begin
   c <= a and b;
end process;
```



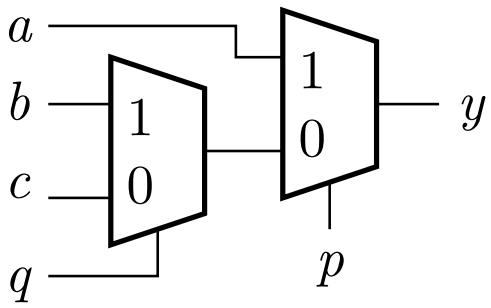
Aus Fallunterscheidungen werden Multiplexer

Bei einer Fallunterscheidung muss dem Zuweisungsziel in jedem Kontrollpfad ein Wert zugewiesen werden. If-Elsif wird durch eine Multiplexerkette nachgebildet:

```
signal a, b, c, p, q, y: std_logic;
...
process(a, b, c, p, q)
begin
  if     p='1' then y <= a;
  elsif  q='1' then y <= b;
  else              y <= c;
  end if;
end process;
```



Statt Else-Zweig, ist auch ein bedingtes Überschreiben möglich. Wichtig ist nur, dass jedem Ausgabesignal im Berechnungsfluss ein neuer Wert zugewiesen wird. Denn das Speichern eines Wertes wird durch ein Latch nachgebildet.

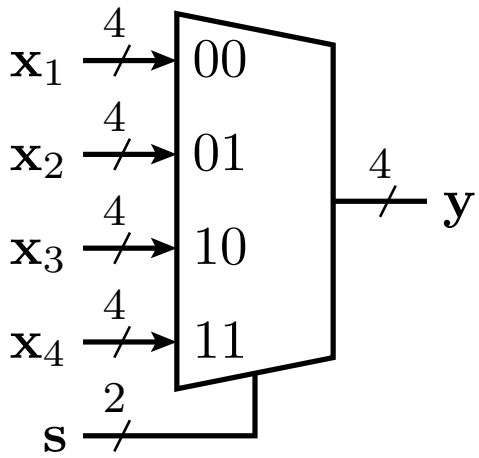



```
signal a, b, c, p, q, y: std_logic;
...
process(a, b, c, p, q)
begin
 y <= c;
 if     p='1' then y <= a;
 elsif  q='1' then y <= b;
 end if;
end process;
```

Wenn einem Signal zum selben Simulationzeitpunkt mehrere Werte zugewiesen werden, gilt der letzte.

Case-Anweisung für große Multiplexer

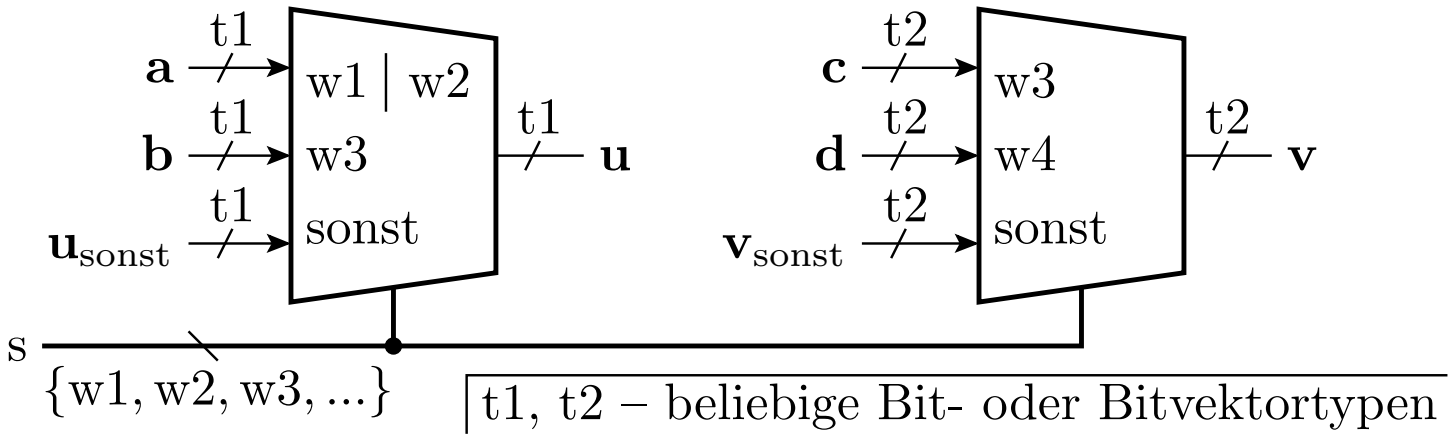
```
signal s: std_logic_vector(1 downto 0);
signal x1,x2,x3,x4,y: std_logic_vector(3 downto 0);
...
process(s, x1, x2, x3, x4)
begin
 case s is
  when "00"     => y <= x1;
  when "01"     => y <= x2;
  when "10"     => y <= x3;
  when others   => y <= x4;
 end case;
end process;
```



Im Beispiel hat der Multiplexer je einen Dateneingang für drei Auswahlwerte und den Sonst-Fall. Der letzte Fall »s="11"« muss **others** sein. In der Simulation erfasst **others** auch "0X", ...

Eine Auswahlanweisung, mehrere Multiplexer

```
                                  u <= u_sonst;                    v <= v_sonst;
case s is
  when w1 | w2 =>                 u <= a;
  when w3      =>                 u <= b;                          v <= c;
  when w4      =>                                                  v <= d;
  when others  =>                                 null;
end case;                         Mux1                             Mux2
```



Zuweisung eines »Standardwertes« vor der Auswahlanweisung.
Überschreiben für einen Teil der Auswahlwerte.

Beschreibung einer Wertetabelle mit einer Auswahlanweisung

```
signal x: std_logic_vector(3 downto 0);
signal y: std_logic;
...
process(x)
begin
 case x is
  when "1000"|"0100"|"0010"|"0001"
                  => y <= '1';
  when others     => y <= '0';
 end case;
end process;
```

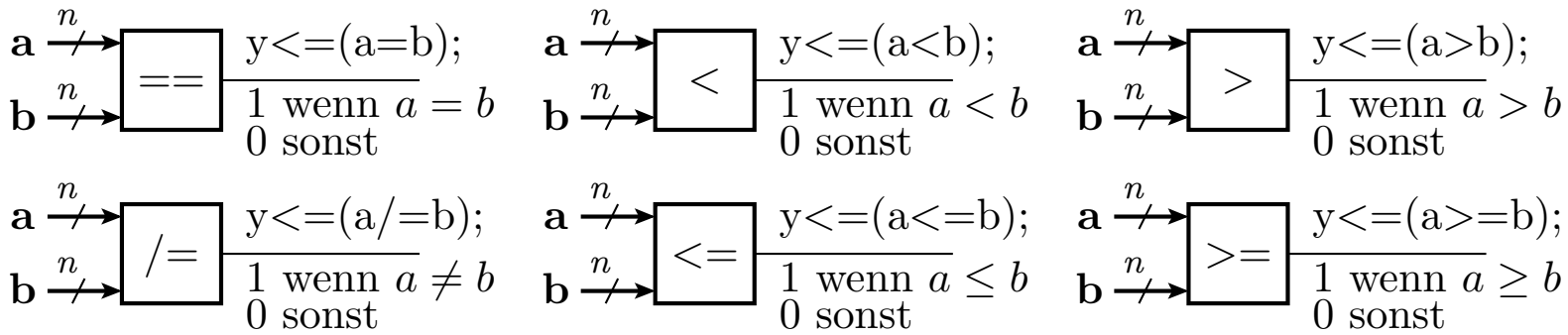
x_3	x_2	x_1	x_0	y
0	0	0	1	1
0	0	1	0	1
0	1	0	0	1
1	0	0	0	1
sonst				0

Vergleichsoperatoren und Komparatoren

Die Vergleichsoperatoren für Bedingungen »=«, »/=« (ungleich), »>« etc. werden durch Komparatoren nachgebildet. Tests auf Gleich- oder Ungleichheit sind für alle Bit- und Bitvektortypen möglich, Größenvergleiche nur für Zahlentypen (`integer`, `natural`, `positive`) und die Bitvektortypen zur Zahlendarstellung:

```
--fuer ganzzahlige vorzeichenfreie Zahlen
unsigned is array (natural range <>) of std_logic;
-- fuer ganzzahlige vorzeichenbehaftete Zahlen
signed is array (natural range <>) of std_logic;
```

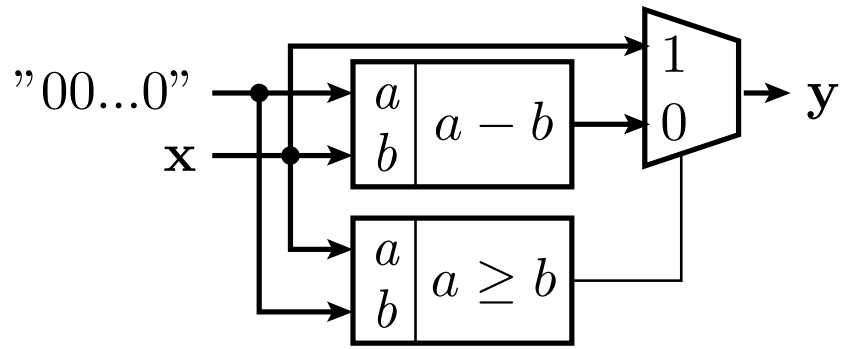
signal a, b: [un]signed(n-1 **downto** 0); **signal** y: std_logic;



Addition, Subtraktion und Multiplikation

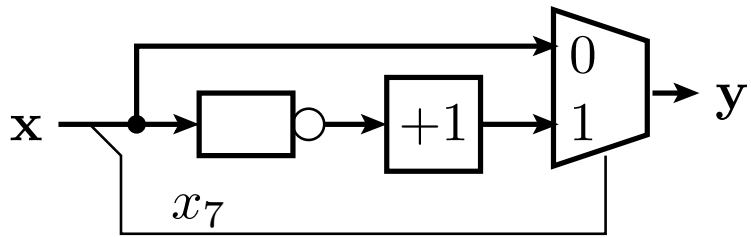
Arithmetische Operationen sind kompliziertere Algorithmen, die in einem oder mehreren Schritten ausgeführt werden. Für die Ausführung in einem Schritt eignen sich Addition, Subtraktion, Inkrement (+1), Dekrement (-1), Negation, Betrag und Multiplikation für positive ganze Zahlen und vorzeichenbehaftete Zahlen im Zweierkomplement. Beispiel Bildung des Betrags:

```
signal x: signed(7 downto 0);
signal y, unsigned(7 downto 0);
...
process(x)
begin
  if (x>=0) then
    y<=unsigned(x);
  else
    y<=unsigned(-x);
end process;
```



Die Funktion `unsigned(x)` ändert den Typ in vorzeichenfrei[4]. Der Betrag lässt sich mit weniger Schaltungsaufwand bilden. Im Zweierkomplement ist das führende Bit das Vorzeichenbit und die Negation ist die bitweise Negation plus eins. Bei führendem Bit eins ist das Ergebnis der negierte Eingabevektor plus eins.

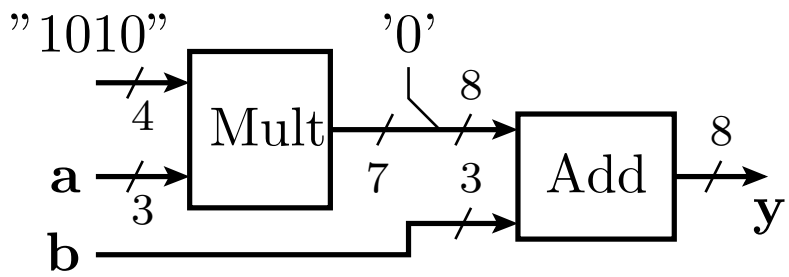
```
signal x: signed(7 downto 0);
signal y, unsigned(7 downto 0);
...
process(x)
begin
  if (x(7)='0') then
    y<=unsigned(x);
  else
    y<=unsigned((not x)+'1');
end process;
```



[4] Für eng verwandte Typen, `integer`, `natural` und `positive` untereinander und `std_logic_vector`, `unsigned` und `signed` untereinander, ist der Name der Konvertierfunktion gleich dem Namen des Zieltyps.

Bei der Addition ist die Bitanzahl der Summe gleich der des längsten Summanden. Um auch ein Übertragsbit zu erhalten, muss dem längsten Summanden eine führende Null vorangestellt werden. Die Bitanzahl des Produkts ist gleich der Summe der Anzahl der Bits der Faktoren. Im Beispiel werden 3 Bits mit einer 4-Bit-Konstanten multipliziert. Das 7-Bit-Produkt wird um eine führende Null erweitert, damit die nachfolgende Addition den Übertrag als achtes Bit berechnet.

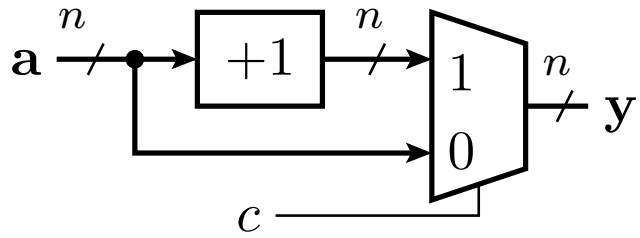
```
signal a, b: unsigned(2 downto 0);
signal y: unsigned(7 downto 0);
...
process(a, b)
 variable v: unsigned(6 downto 0);
begin
 A1: v:= a * "1010";
 A2: y<= ('0'&v) + b;
end process;
```



Bedingte Zähloperation

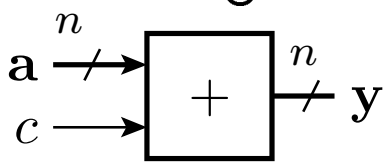
Eine naheliegende Beschreibung einer bedingten Zähloperation ist, dass nur bei erfüllter Bedingung gezählt wird:

```
signal a, y: unsigned(n-1 downto 0);
signal c: unsigned(0 downto 0);
...
if c="0" then y <= a;
else y <= a +"1";
end if;
```



Eine funktionsgleiche Beschreibung einer einfacheren Schaltung für eine bedingte Inkrement-Operation:

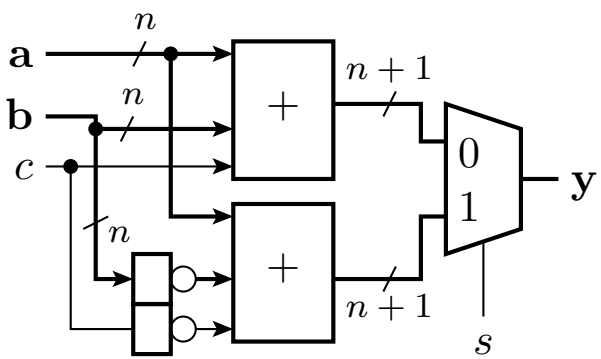
```
y <= a + c;
```



Die Synthese führt viele Optimierungen selbstständig aus. In Zweifelsfällen Syntheseergebnis ansehen und gegebenenfalls die Beschreibung der Zielfunktion nachbessern.

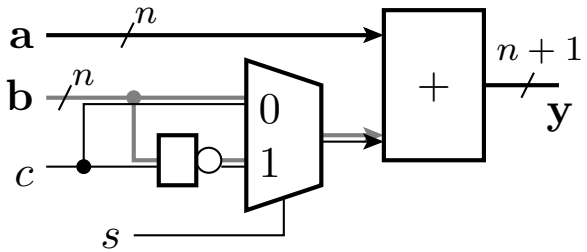
Schaltung zur wahlweisen Addition und Subtraktion

```
signal a, b:unsigned(n-1 downto 0);
signal y: unsigned(n downto 0);
signal c, s: std_logic;
...
if s='0' then  y <= ('0'& a)+b+c;
else y <= ('0'& a)-b-c;
end if;
-- optimierte Beschreibung
variable vb:unsigned(n-1 downto 0);
variable vc:unsigned(0 downto 0)
...
if s='0' then  vb:=b; vc(0):=c;
else vb:=not b; vc(0):= not c;
end if;
y <= ('0'& a)+vb+vc;
```



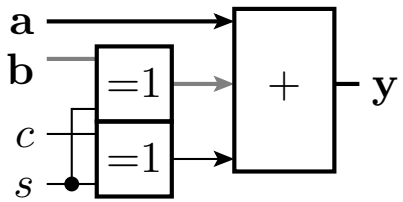
⇓

Umschalter vor dem Addierer



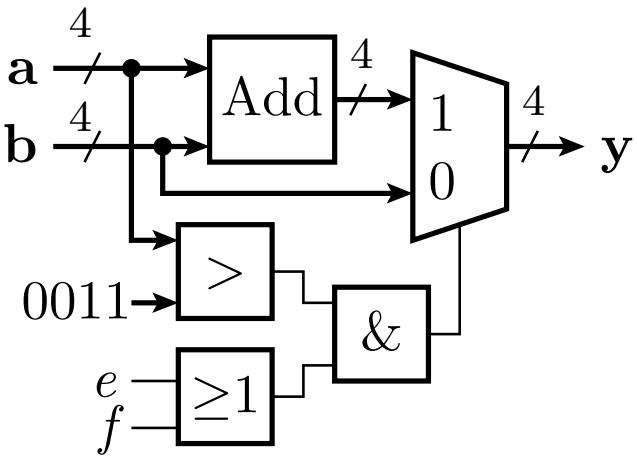
⇓

Vereinfachung der Bitscheibenop.



Eine etwas kompliziertere Berechnung

```
signal a, b, y: unsigned(3 downto 0);
signal e, f: std_logic;
...
process(a, b, e, f)
begin
 if (a > "0011") and
     (e or f) ='1' then
  y <= a + b;
 else
  y <= b;
 end if;
end process;
```



Es mangelt nicht an Fehlermöglichkeiten: falsche Bitanzahl, falsche Datentypen, Denkfehler, zu kleine Wertebereiche, unberücksichtigte Sonderfälle, ... Komplexere Funktionen simulieren!

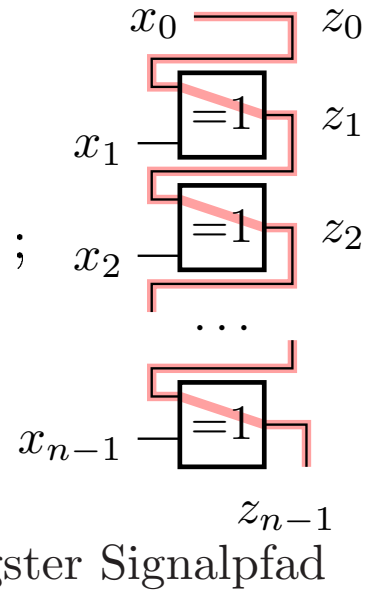
Schleifen

Schleifen zur Beschreibung von Verarbeitungsfunktionen sind mit Vorsicht anzuwenden. Sie beschreiben keine zeitliche Wiederholung, sondern eine Schaltungsgenerierung. Beispiel n-Bit-EXOR:

$$y = x_{n-1} \oplus x_{n-2} \oplus \ldots \oplus x_1 \oplus x_0$$

Schleife mit Ausführung von rechts nach links:

```
signal x: std_logic_vector (n-1 downto 0);
variable z: std_logic_vector (n-1 downto 0);
...
begin
  z(0) := x(0);
  for idx in 1 to n-1 loop
    z(idx) := z(idx-1) xor x(idx);
  end loop;
```



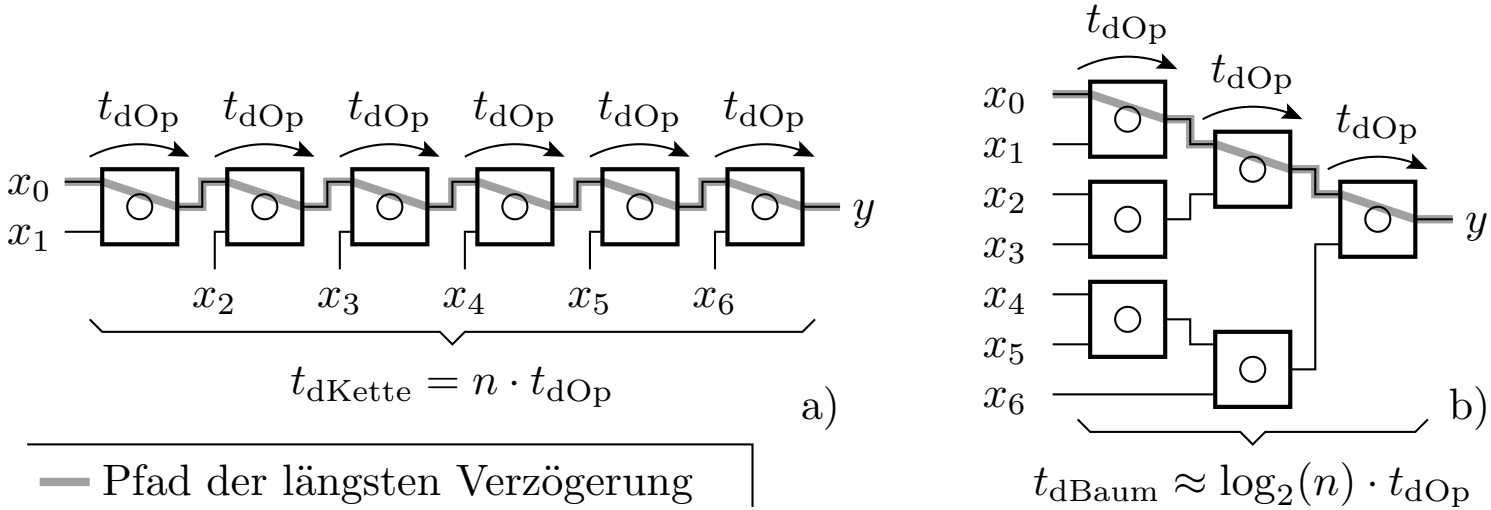
Das Ergebnis ist eine Kette, in der alle EXOR-Operationen nacheinander ausgeführt werden.

Bäume statt Ketten

Die Gesamtverzögerung ist die Summe der Einzelverzögerungen. EXOR-Verknüpfungen sind assoziativ. Geschickt geklammert

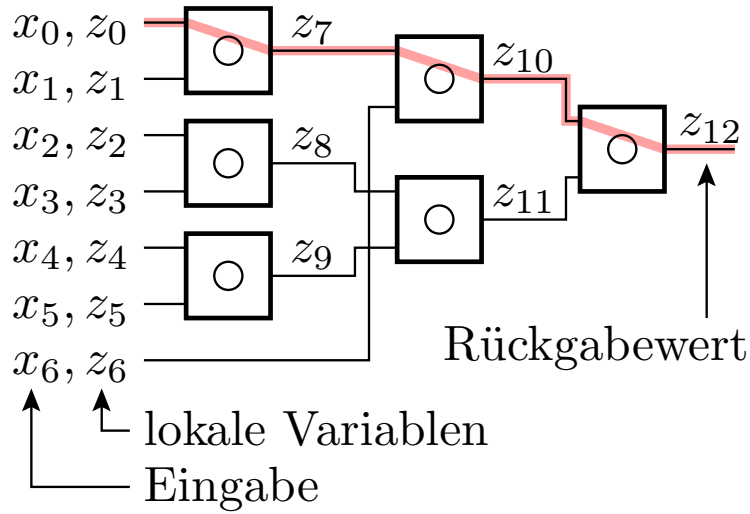
$$x_6 \circ \ldots \circ x_1 \circ x_0 = ((x_1 \circ x_0) \circ (x_3 \circ x_2)) \circ ((x_4 \circ x_5) \circ x_6)$$

($\circ$ assoziative Operation) wird aus der Kette ein Baum, bei dem die Verzögerungszeit nur noch logarithmisch mit der Anzahl der Operationen zunimmt.



Generierungsschleife für Bäume

```
type tVektortyp is (natural range <>) of tTyp;
signal x: tVektortyp(n-1 downto 0);
variable z: tVektortyp(2*n-2 downto 0);
...
begin
  z(x'range):=x;
  for idx in 0 to n-2 loop
    z(idx+n):=z(2*idx)
        <op> z(2*idx+1);
  end loop;
```

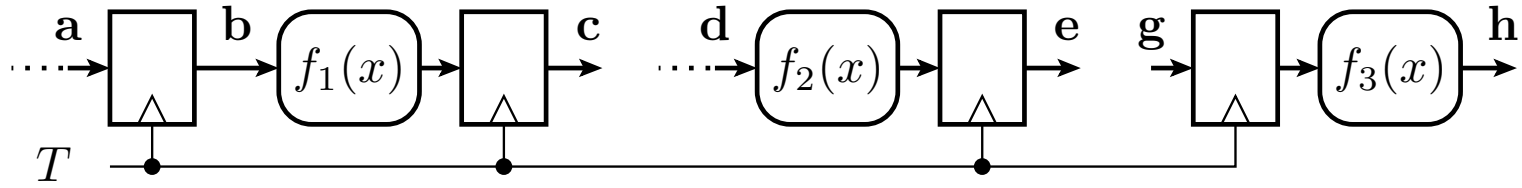


- Das Attribut **'range** liefert den Indexbereich.
- Für eine Schaltung aus 6 Gattern ist ein geklammerter Ausdruck übersichtlicher und schneller hingeschrieben.

Abtast- und RT-Funktionen

Abtast- und RT-Funktionen

Beschreibung durch einen Prozess, der nur bei Taktänderung geweckt wird. Neuberechung nur bei aktiver Taktflanke. Signalzuweisungen ohne Verzögerung und Pseudo-Werte ('X', 'U', ...).



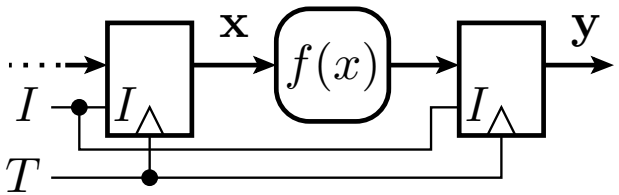

```
process(T)
begin
 if rising_edge(T) then
  b <= a;       -- Eingabeabtastung
  c <= f1(b);   -- Register-Transfer-Funktion
  e <= f2(d);   -- Verarbeitung + Abtastung
  h <= f3(g);   -- Abtastung + Verarbeitung
 end if;
end process;
```

Ein Abtastprozess kann Schaltungen mit mehreren Registern beschreiben.

Zusätzliche Registerinitialisierung

Die Initialisierung von Registern erfolgt oft zustandsgesteuert für alle Register gemeinsam. Bei aktivem Initialisierungssignal wird eine Konstante und sonst bei aktiver Taktflanke der Eingabewert übernommen. Wie später ab Folie 59 gezeigt wird, muss das Initialisierungssignal zeitlich zum Takt ausgerichtet sein.

```
process(I, T)
begin
 if I='1' then                -- *1
  x <= (others=>'0');         -- Zuweisung der
  y <= "0110000";             -- Anfangswerte
 elsif rising_edge(T) then    -- *2
  x <= ...;
  y <= f(x);
 end if;
end process;
```



*1	oder »I = '0'«
*2	oder »falling_edge(T)«

Registerextraktion

Was für Register beschreiben die nachfolgenden drei Prozesse? Welche Anschlusssignale sind Daten-, Takt- und Initialisierungseingänge und welche sind Datenausgänge? Mit welchen Werten werden die initialisierbaren Register initialisiert?

```
signal a,b,c,d: std_logic;
signal K,L,M: std_logic_vector(2 downto 0);
```

```
process(a)
begin
 if rising_edge(a)
                then
  K(0) <= d;
  K(1) <= K(0);
  K(2) <= K(1);
 end if;
end process;
```

```
process(b, c)
begin
 if c='1' then
  L <= "000";
 elsif falling_edge(b)
  then
  L <= K;
 end if;
end process;
```

```
process(a, c)
begin
  if c='0' then
   M <= "010";
  elsif rising_edge(a)
   then
    M <= K;
  end if;
end process;
```

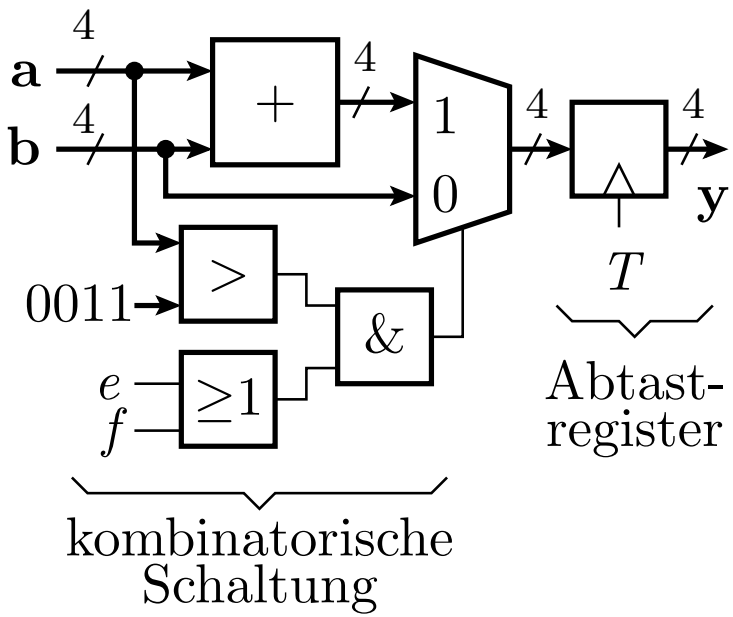
Anschlusssignale und Parameter der beschriebenen Register:

Dateneingangssignal	d	K(0)	K(1)	K	K
Datenausgangssignal	K(0)	K(1)	K(2)	L	M
Bitbreite	1	1	1	3	3
Taktsignal	a ↑	a ↑	a ↑	b ↓	a ↑
Initialisierungssignal	–	–	–	c (H)	c (L)
Initialisierungswert	–	–	–	000	010

↑ – steigende Flanke; ↓ – fallende Flanke; H – high-aktiv; L – low-aktiv

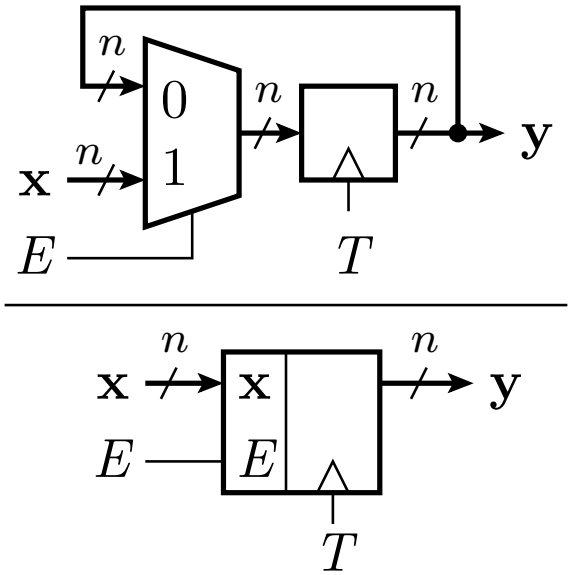
Kombinatorische Schaltung mit Ausgaberegister

```
signal a, b, y: std_logic_vector(3 downto 0);
signal T, e, f: std_logic;
...
process(T)
begin
 if rising_edge(T) then
  if (a>"0011") and
     (e or f) ='1' then
   y <= a+b;
  else
   y <= b;
  end if;
 end if;
end process;
```



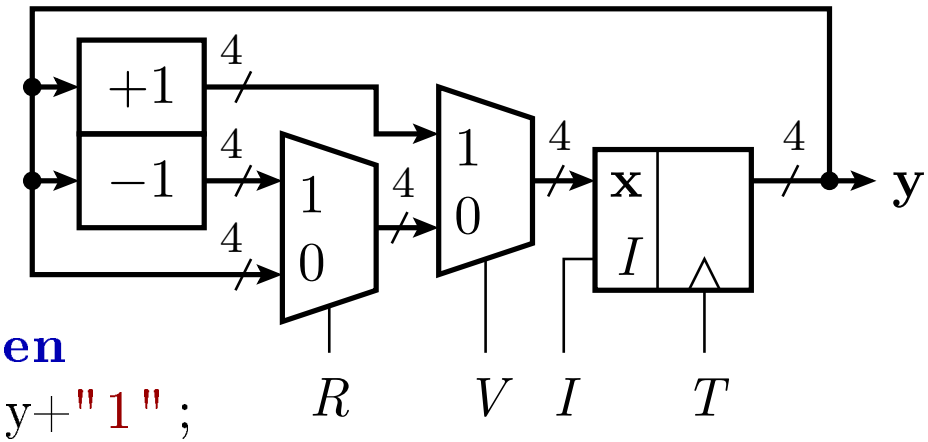
Abtastregister mit bedingter Übernahme

```
signal x, y: std_logic_vector(n-1 downto 0);
signal T, E: std_logic;
...
process(T)
begin
  if rising_edge(T) then
    if E='1' then
      y <= x;
    end if;
  end if;
end process;
```



Eine bedingte Übernahme wird mit einem Multiplexer nachgebildet. Bei erfüllter Übernahmebedingung, im Beispiel $E =$ '1', Weiterleitung des Eingabewerts und sonst des Istwerts zum Registereingang.

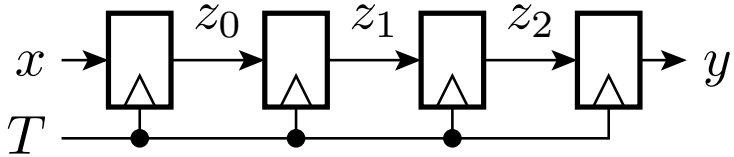
```
signal y: unsigned(3 downto 0);
signal T, I, V, R: std_logic;
...
process(T, I)
begin
 if I='1' then
  y<="0000";
 elsif rising_edge(T) then
  if    V='1' then y <= y+"1";
  elsif R='1' then y <= y-"1";
  end if;
 end if;
end process;
```



In einem kombinatorischen Prozess sind Rückführungen der Form »y<=y+"1"« verboten. Denn sie würden ein Weiterzählen in Zeitschritten undefinierter Dauer, d.h. ein undefiniertes nicht synthetisierbares Verhalten beschreiben. In einem Abtastprozess, der in jeder Taktperiode genau einmal geweckt wird, ist das ein wohldefiniertes, durch eine Schaltung nachbildbares Verhalten.

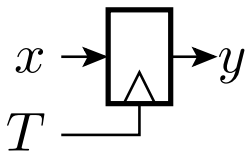
Speicherverhalten von Variablen

```
signal T, x, y: std_logic;
...
process(T)
 variable z: std_logic_vector(2 downto 0);
begin
 if rising_edge(T) then
  y <= z(2);   z(2):=z(1);
  z(1):=z(0);  z(0):=x;
 end if;
end process;
```



Ein Signal übernimmt zugewiesene Werte erst, wenn der Prozess schläft, eine Variable sofort. Variablen, die nach dem Wecken zuerst gelesen und dann beschrieben werden, verzögern um eine Taktperiode und beschreiben Register. Im Beispiel werden aus den drei Variablenzuweisungen drei Register.

```
signal T, x, y: std_logic;
...
process(T)
 variable z: std_logic_vector(2 downto 0);
begin
 if rising_edge(T) then
  z(0) := x;  z(1) := z(0);
  z(2) := z(1);  y <= z(2);
 end if;
end process;
```

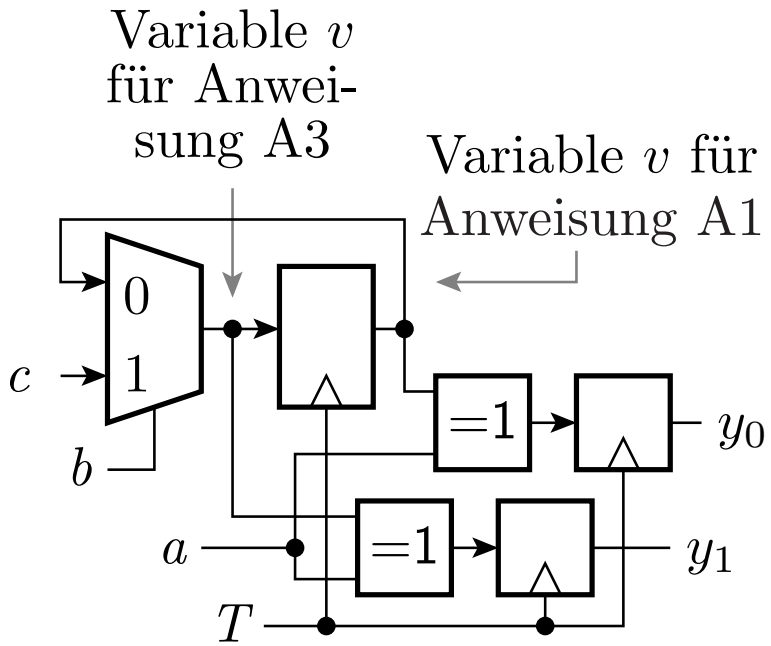


- Bei Umkehrung der Zuweisungsreihenfolge wird bei jeder aktiven Taktflanke der aktuelle Wert von x, statt des von drei Takten zuvor an y zugewiesen. Eines statt drei Register.
- Auf Prozessvariablen haben andere Prozesse keinen Zugriff.
- Für Variablen, auch solche, die in Register übersetzt werden, berechnet die Simulation keine Zeitverläufe.

Empfehlung: Keine Register durch Variablen beschreiben.

Speicherverhalten bedingter Variablenzuweisungen

```
signal T, a, b, c, y0, y1: std_logic;
...
process(T)
 variable v: std_logic;
begin
 if rising_edge(T) then
A1:   y0 <= v xor a;
   if b='1' then
A2:    v:= c;
   end if;
A3:   y1 <= v xor a;
 end if;
end process;
```



Variablenzuweisungen in Fallunterscheidungen, denen im Berechnungsfluss davor kein Wert zugewiesen wird, haben eine nicht einfach zu verstehende Wirkung. Besser nicht benutzen!

Typische Beschreibungsfehler

Entwurfsfehler und Fehlervermeidung

Synthese liefert nicht immer eine funktionierende Schaltung.
Übersetzungsnachrichten, die auf Unzuverlässigkeiten hinweisen:

- kombinatorisch erzeugtes Taktsignal,
- Latches gefunden,
- maximale Taktfrequenz unter der geforderten, ...

Weitere gefährliche Fallen (siehe nachfolgende Abschnitte):

- fehlende Abtastung asynchroner Eingabesignale,
- nicht synchronisiertes Initialisierungssignal,
- fehlende und falsche Constraints,
- ungewolltes Schwing- oder Speicherverhalten, ...

Syntheseprogramme sind nicht fehlerfrei:

- Bevorzugung bewährter Beschreibungsmittel.

Speicherverhalten in kombinatorischen Prozessen

signal x1, x2, sz, y, yF1, yF2, yF3: std_logic ;

korrekt

```
process (x1, x2);
  variable vz: std_logic ;
begin
  vz := x1 and x2;
  y <= not vz;
end process;
```

F1

```
process (x1);
  variable vz: std_logic ;
begin
  vz := x1 and x2;
  yF1 <= not vz;
end process;
```

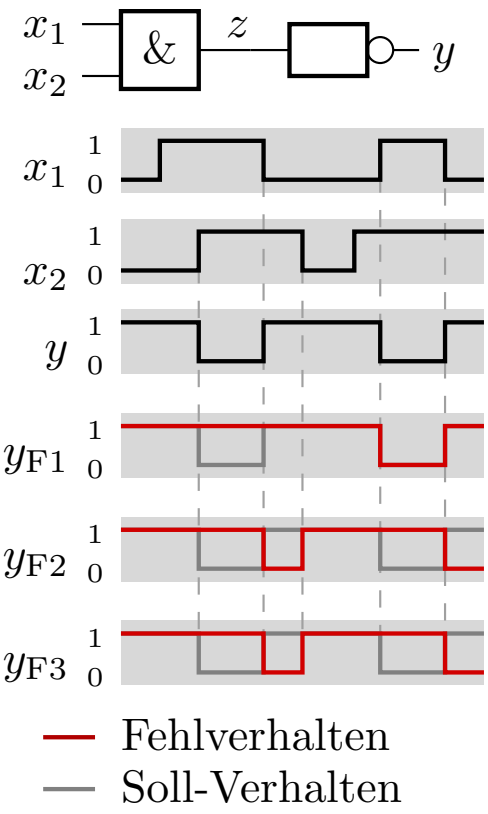
F2

```
signal sz: std_logic ;
process (x1, x2);
begin
  sz <= x1 and x2;
  yF2 <= not sz;
end process;
```

F3

```
process(x1, x2);
  variable vz: std_logic ;
begin
  yF3 <= not vz;
  vz := x1 and x2;
end process;
```

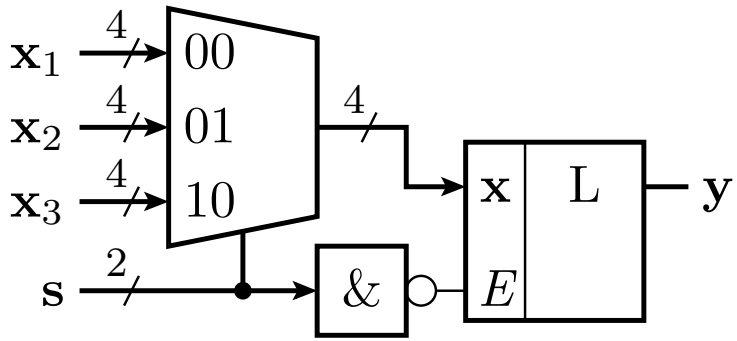
warte

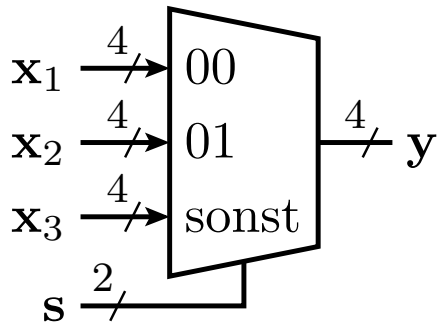


F1: fehlendes Signal in der Weckliste; F2: Signal statt Variable als Zwischenspeicher; F3: Variablenwert vor der Zuweisung ausgewertet.

Fallunterscheidung mit fehlender Zuweisung

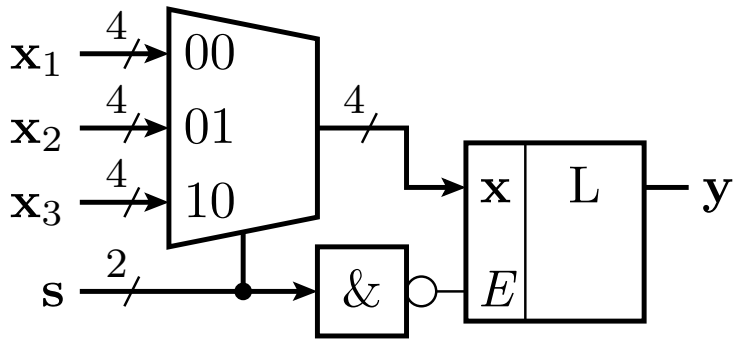
```
signal x1, x2, x3, y: std_logic_vector(3 downto 0);
signal s: std_logic_vector(1 downto 0);
  ...
 process(s, x1, x2, x3)
 begin
  case s is
   when "00" =>  y<=x1;
   when "01" =>  y<=x2;
   when "10" =>  y<=x3;
   when others => null;
 end case;
end process;
```





Die Null-Anweisung (tue nichts) im Others-Zweig bewirkt einen Latch-Einbau.

Korrektur durch Wertezuweisung, z.B. »y<=x3« im Others-Zweig.

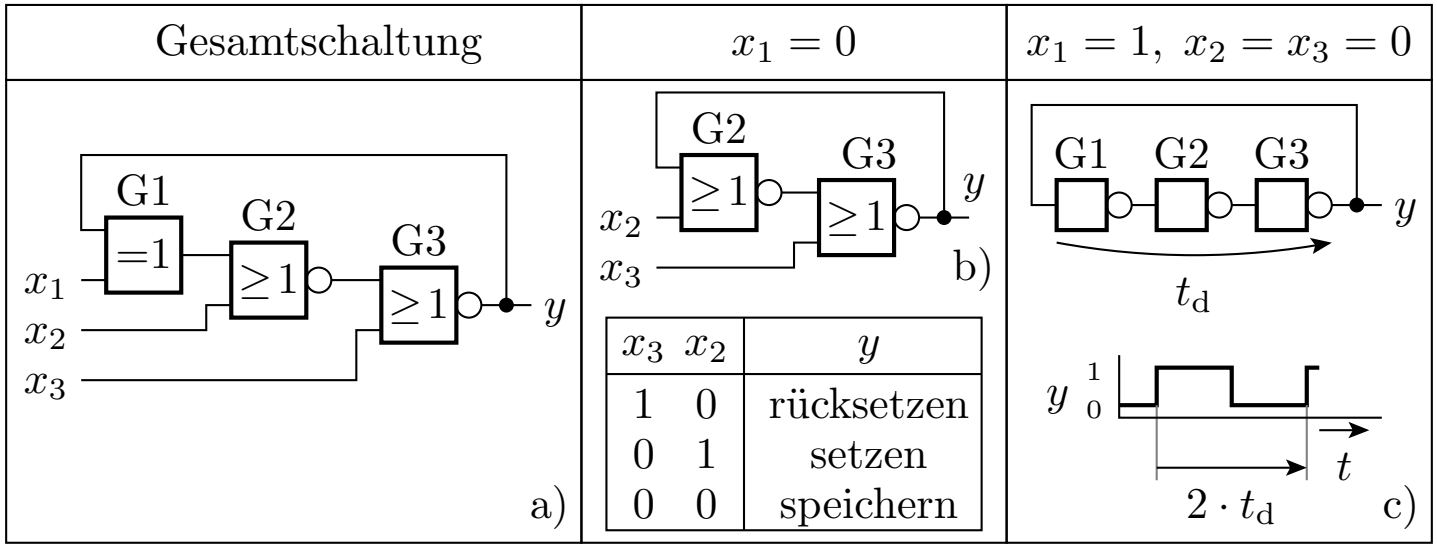


- Auch, falls nur die drei explizit beschriebenen Auswahlwerte auftreten können, ist in der Case-Anweisung eines kombinatorischen Prozesses im Others-Zweig dem Ausgang ein Wert zuzuweisen.
- Das von der Synthese zur Zustandsspeicherung für alle anderen Eingaben eingebaute Latch hat ein von einer kombinatorischen Funktion gebildetes potentiell glitch-behaftetes Freigabesignal.
- Schaltung Robustheit gegenüber Laufzeittoleranzen? [5]

[5] Im Beispiel ist die Schaltung robust gegenüber Glitches auf dem Freigabesignal, weil das Freigabesignal nach jeder Eingabeänderung wieder aktiviert und der korrekte Auswahlwert weitergereicht wird.

Rückführungen in kombinatorischen Schaltungen

Die Rückführungen von Berechnungsergebnissen auf Eingänge müssen immer über Register laufen, die alle Werteänderungen bis zur nächsten Taktflanke verzögern. Das Beispiel zeigt, dass die Rückführung eines Ausgangs auf einen Eingang bei einer kombinatorischen Schaltung ein Speicher- oder ein Schwingverhalten verursachen kann. Tatsächliches Verhalten laufzeitabhängig.



x_3	x_2	y
1	0	rücksetzen
0	1	setzen
0	0	speichern

Constraints

Constraints

Constraints sind Zusatzinformationen zur Zielfunktion für die

- Synthese, z.B.

  `USE_DSP48`

  `RAM_STYLE`

- Implementierung, z.B.

  `LOC` <*placement*> <*constraint*>

  `PERIOD` <*timing*> <*constraint*>

Synthese-Constraints weisen die Nutzung (Bevorzugung) bestimmter vorentworfener Bausteintypen an wie Block-RAM's und DSP- (Digital Signal Processor) Einheiten, die bestimmte PLD als vorgefertigte Bausteine enthalten (siehe später Foliensatz F5).

LOC und PERIOD

LOC bindet Schaltungselemente an programmierbare Hardware-Bausteine der PLDs. Für Anschlüsse zwingend. Aus »Ampel.ucf« der Laborübung:

```
NET "GCLK0" LOC = V10; # Takteingang
NET "LD<0>" LOC = U16; # Leuchtdiode LD0
NET "LD<1>" LOC = V16; # Leuchtdiode LDE
```

PERIOD beschreibt die Maximaldauer einer Taktperiode und für Register-Transfer-Funktionen die Obergrenze der Summe der Verzögerungs- und Vorhaltezeiten aller Pfade. Auf der Praktikumsbaugruppe ist am Gehäuseanschluss V10 ein 100 MHz-Oszillator angeschlossen:

```
NET "GCLK" LOC = V10;        # Takteingang
NET "GCLK" PERIOD = 10ns;    # Taktperiode 10 ns
```

Zeitkritische Signale

Signaländerungen für die Steuerung der zeitlichen Abläufe:

- Initialisierung von Speicherzellen (SR, Set/Reset),
- Registertakte (CLK, ClocK),
- Latch-Freigabe (DATA_GATE} und
- die Richtungssteuerung (OE, Output Enable) bidirektionaler Anschlüsse

müssen zeitgenau und glitchfrei sein. Dafür sind bestimmte Leitungsnetze reserviert, die über sog. BUFG-Treiber erreicht werden. Kennzeichnung von Leitungen als solche:

```
NET "<net_name>" BUFG={CLK|\OE|SR|DATA_GATE}
```

Weiterhin können Zeitschranken vorgegeben werden für

- die Verzögerungszeit kombinatorischer Funktionen,
- die Verarbeitung von Eingaben bis zur ersten Abtastung und
- die Verarbeitung von Ausgaben nach der letzten Abtastung.

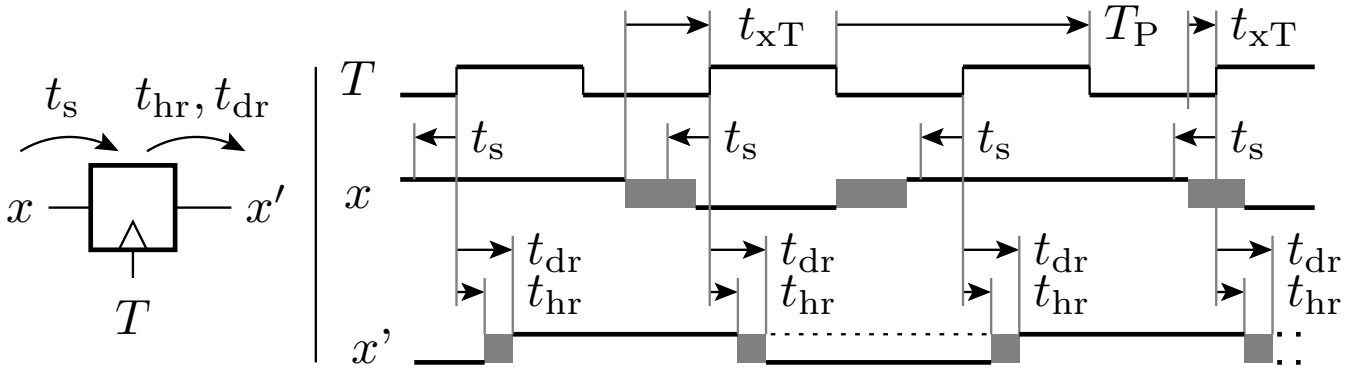
Asynchrone Eingabe

Abtastung

Asynchrone Eingabe und Abtastung

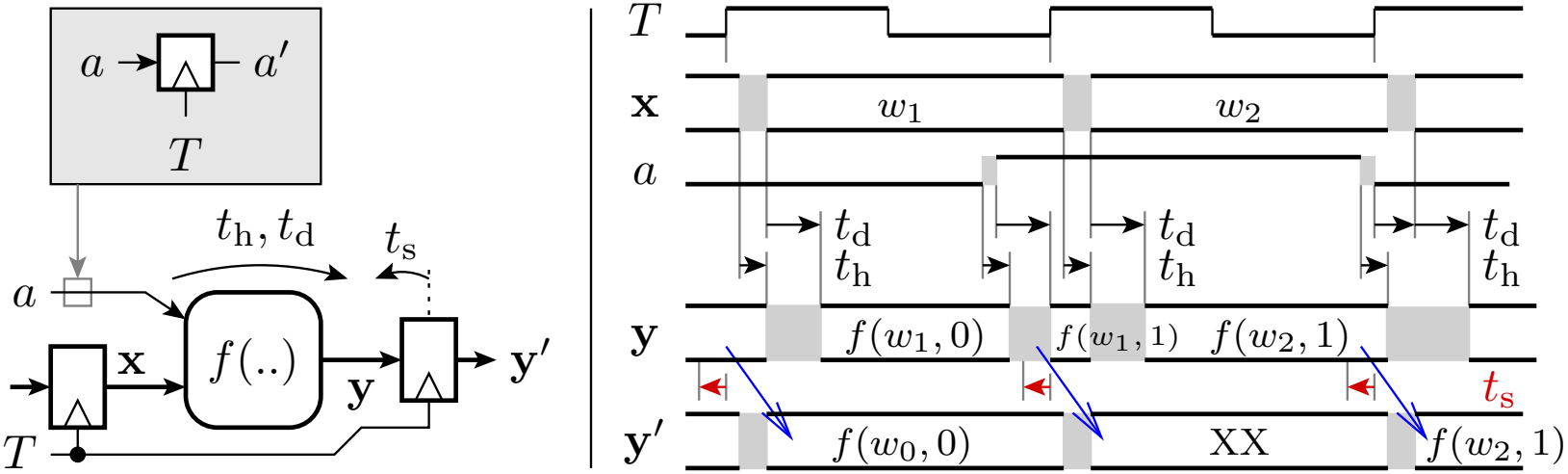
Asynchron bedeutet, dass die Eingabe nicht zeitlich zum Systemtakt ausgerichtet ist. Damit gibt es auch kein mit Constraints beschreibbares zum Takt ausgerichtetes Zeitfenster, in dem die Eingaben garantiert gültig sind.

Asynchrone Eingaben bitweise abtasten:

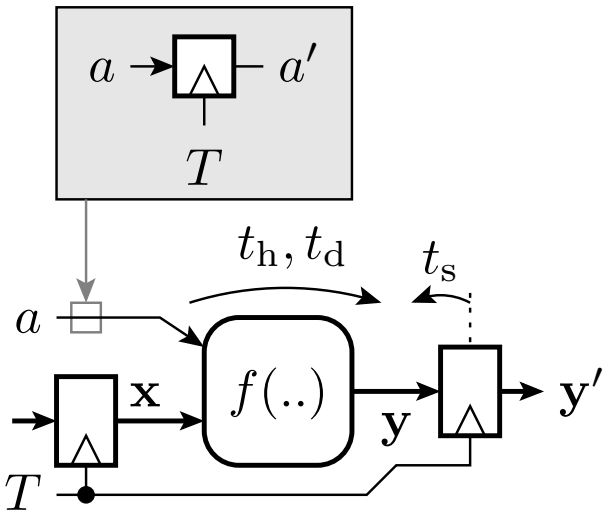


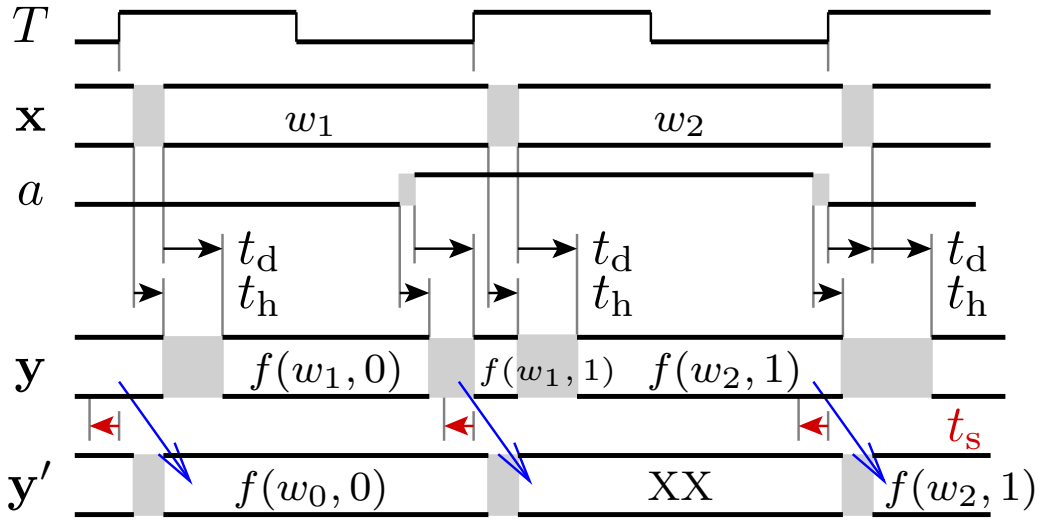
Ein abgetastetes Bitsignal hat auch, wenn es im Abtastmoment ungültig ist, für die komplette folgende Taktperiode den Wert »0« oder »1«.

Verarbeitung vor der Abtastung



Das Signal $\mathbf{y}$ wird aus einem zum Takt ausgerichteten Signal $\mathbf{x}$ und einem nicht ausgerichteten Signal a gebildet. Änderungen von $\mathbf{y}$, ausgelöst durch Änderungen von $\mathbf{x}$ sind zur nächsten aktiven Taktflanke immer gültig, aber von a ausgelöste Änderungen nicht. Wenn $\mathbf{y}$ ein Bitvektor ist, wird bei Abtastung außerhalb des Gültigkeitsfensters ein nicht vorhersehbarer Bitvektor aus Nullen und Einsen übernommen. Wirkung laufzeitabhängig.





Die skizzierte Beispielschaltung arbeitet nur zuverlässig, wenn a vor der Verarbeitung mit einem zusätzlichen Register zeitlich am Takt ausgerichtet wird.

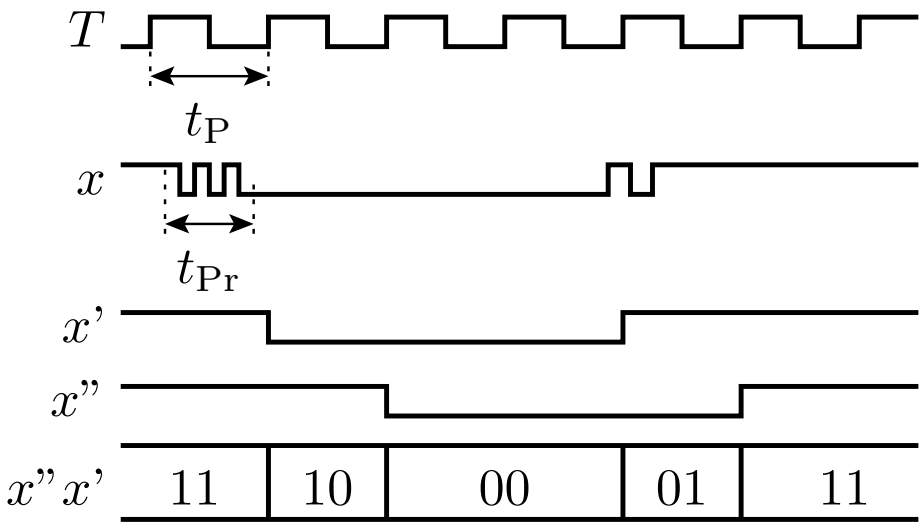
Wichtige Regel

Alle asynchronen Eingaben von Schaltern, anderen Rechnern, ... vor der Verarbeitung bitweise mit dem eigenen Systemtakt abtasten.

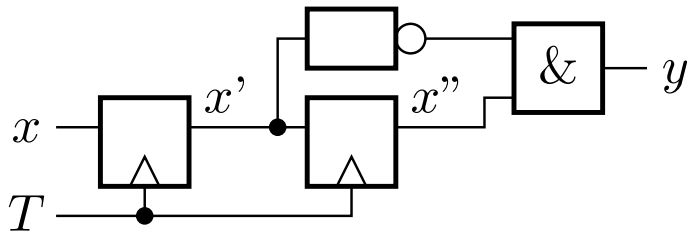
Entprellen

Entprellen von Schaltern und Tastern

Mechanische Eingabeelemente (Taster und Schalter) prellen. Bei Betätigung ist es schwer auszuschließen, dass der Kontakt durch mechanische Schwingungen mehrfach öffnet und schließt. Zur Verhinderung, dass das abgetastete Signal bei einer Einzelbetätigung mehrfach ein- und ausschaltet, ist die Abtastperiode länger als die maximale Prellzeit zu wählen.

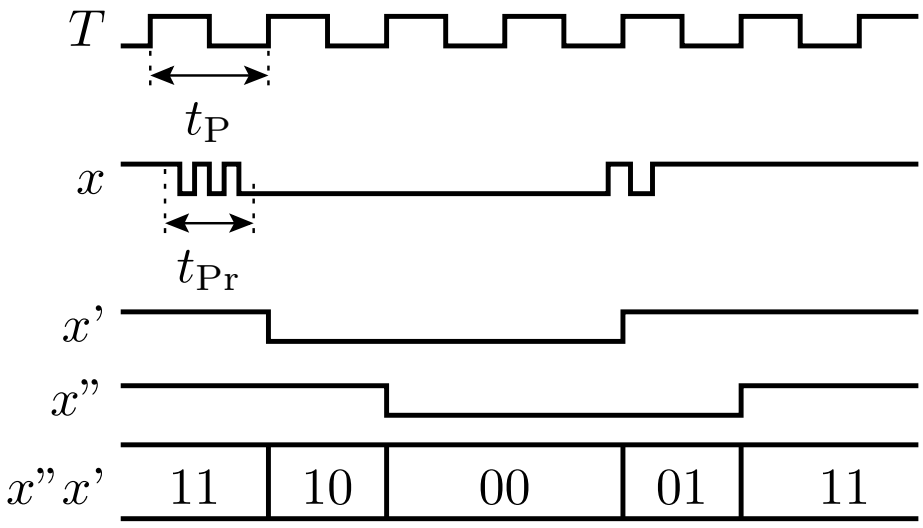


Schaltung zur Erkennung der fallenden Flanken

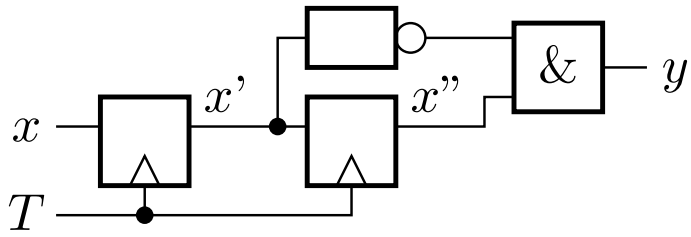


t_{Pr}	maximale Prellzeit
$t_P > t_{Pr}$	Abtastperiode

Zur Erzeugung eines Impulses bei einer Tasterbetätigung ist das Eingabesignal ein zweites mal abzutasten. Tasterbetätigungen sind dann an der Abtastfolge »01« bzw. »10« zu erkennen.



Schaltung zur Erkennung der fallenden Flanken



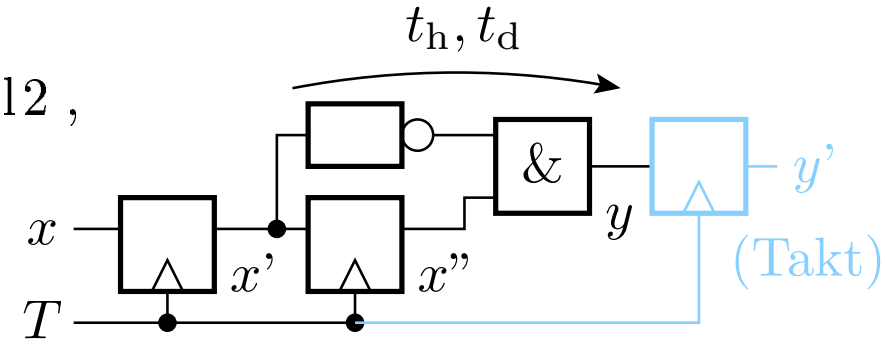
t_{Pr} maximale Prellzeit

$t_P > t_{Pr}$ Abtastperiode

(x', x'' – einmal bzw. zweimal abgetastetes Signal). Ohne die erste Abtastung ist x' und damit y nicht am Takt ausgerichtet und die nachfolgende Schaltung keine laufzeittolerante Register-Transfer-Funktion.

Beschreibung in VHDL

```
signal x, x_del, x_del2,
  T, y: std_logic;
...
process(T)
begin
 if rising_edge(T) then
  x_del  <= x;          -- Beschreibung der
  x_del2 <= x_del;      -- beiden Register
 end if;
end process;
y <= not x_del and x_del2; -- Gatterbeschreibung
```



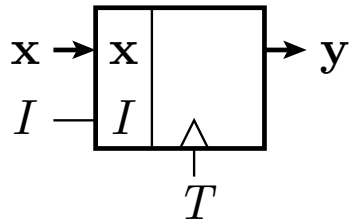
Wichtig ist, dass

- das erste Register nicht vergessen und
- die Periode des Abtasttakts nicht kleiner als die maximale Prellzeit gewählt wird. Ausreichend großer Taktteiler.

Initialisierung

Register mit asynchroner Initialisierung

Register haben meist einen asynchron wirkenden Eingang zur Initialisierung, der Vorrang vor der Übernahme mit dem Takt hat:

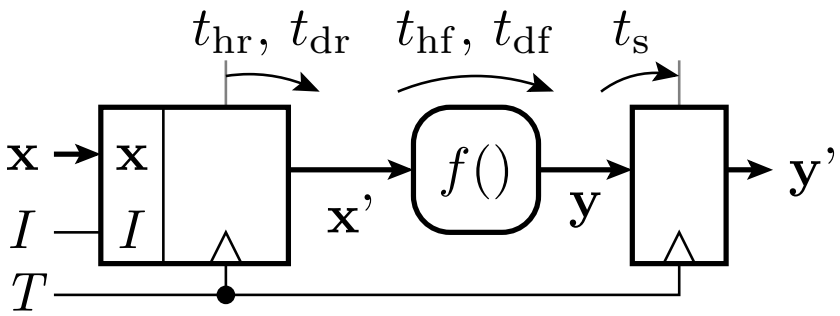


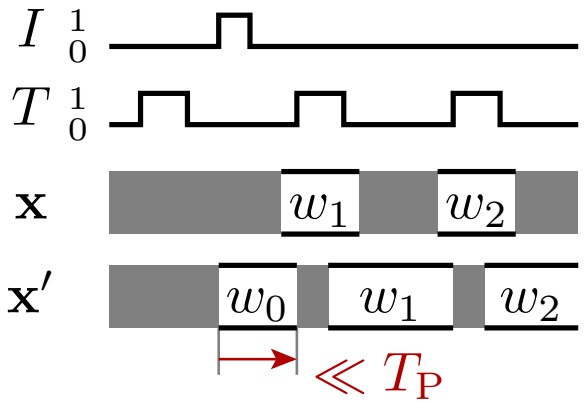

```
signal x, y: t_dat;
signal T,  I: std_logic;
...
process(T, I)
begin
  if I='1' then
    y <= <Anfangswert>;
  elsif rising_edge(T) then
    y <= x;
  end if;
end process;
```

(t_dat – Typ der abzutastenden Daten).

Potentielle Fehlfunktionen

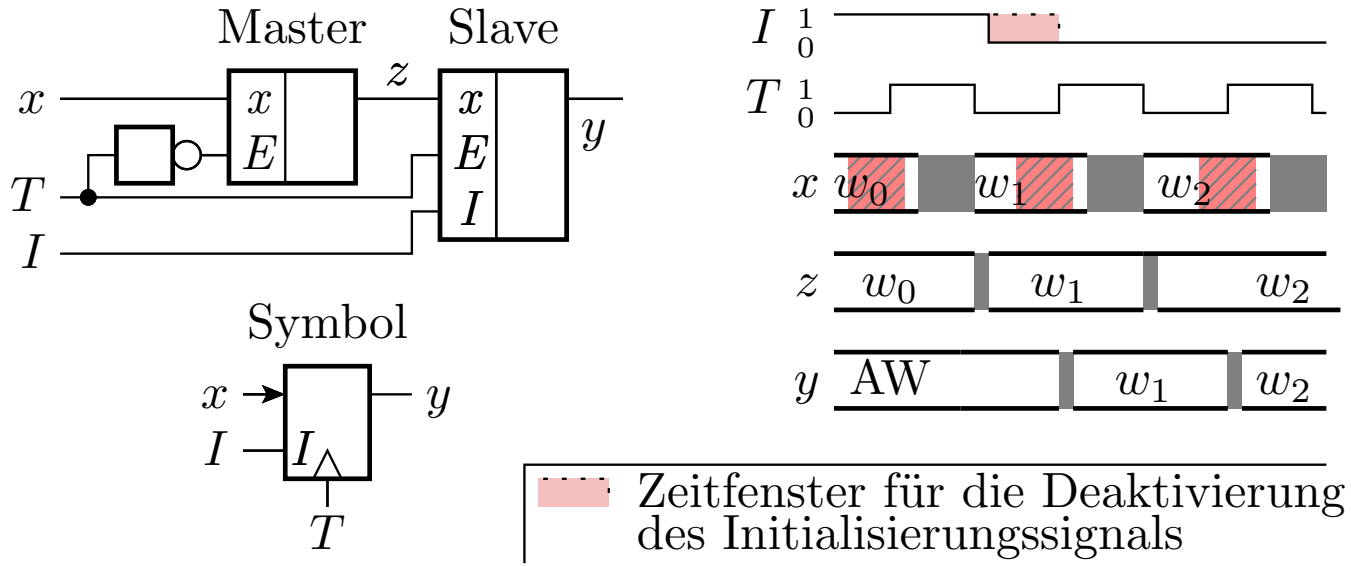
Pulsdauer des Initialisierungssignals zu kurz:



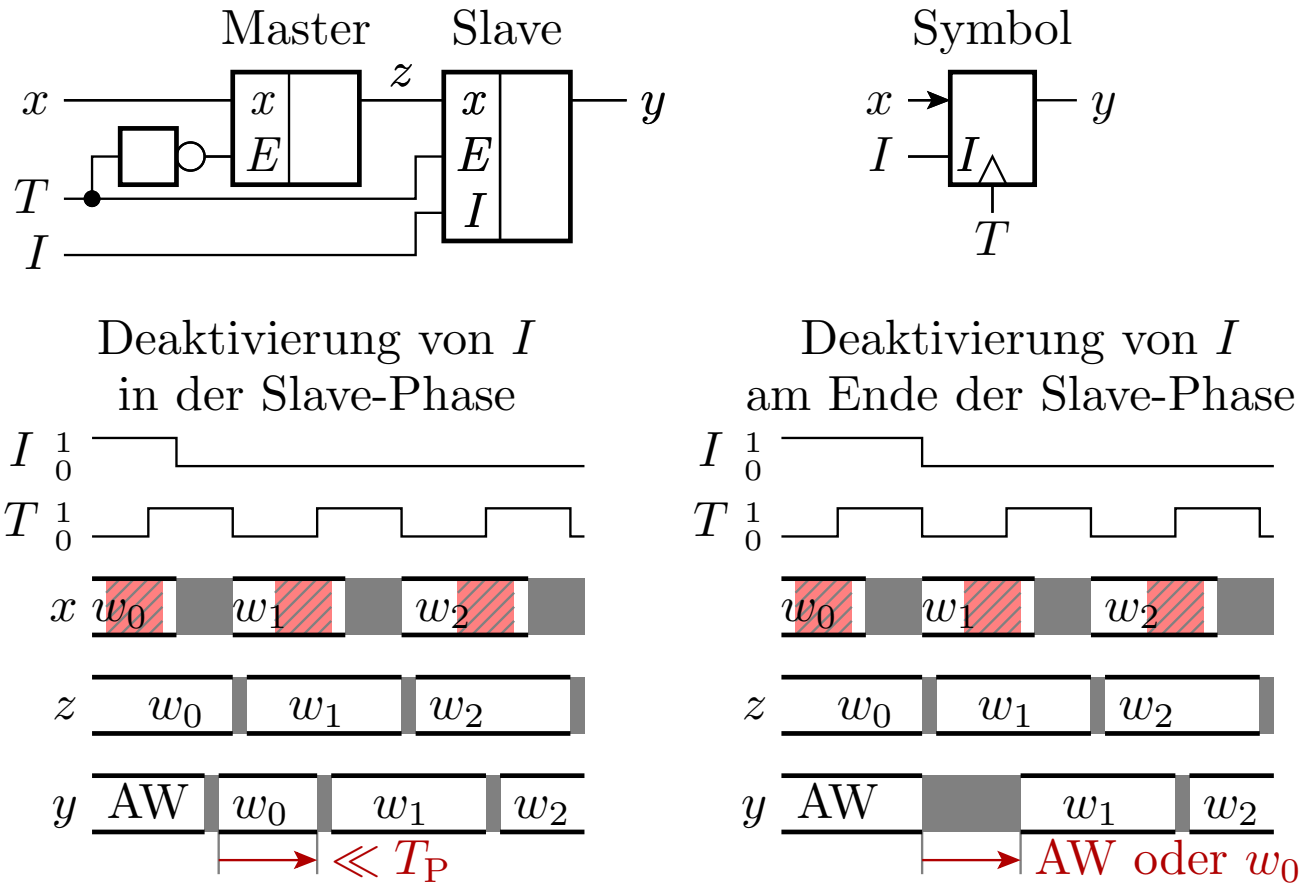


Selbst wenn das Eingaberegister seinen Anfangswert korrekt übernimmt, liegt der Initialwert am Registerausgang zu kurze Zeit stabil an, so dass das nachfolgende Register im ersten Schritt möglicherweise einen ungültigen Wert übernimmt.

Bei Master-Slave-Registern wirkt die asynchrone Initialisierung auf den Slave. Sie muss mindestens eine Taktperiode anliegen und in der Masterphase oder am Anfang der Slave-Phase deaktiviert werden.

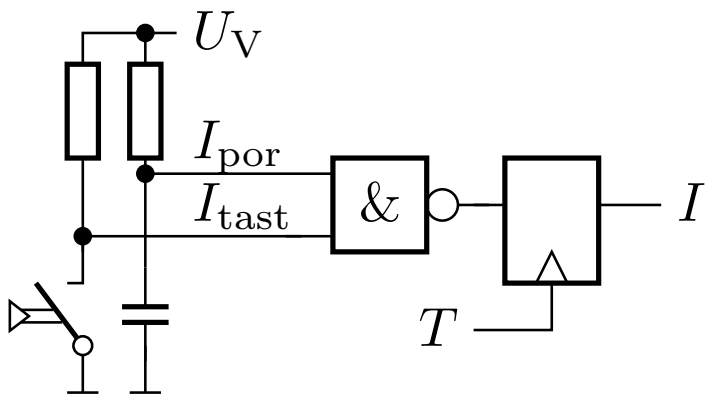


Sonst liegt der anschließend übernommene Wert zu kurz am Registerausgang an, so dass das Register hinter der nachfolgenden Verarbeitungsfunktion möglicherweise einen ungültigen Wert abtastet.



Bei Deaktivierung am Ende der Slave Phase ist es unsicher, ob der Slave den Initialisierungswert behält oder den Wert vom Master übernimmt.

Laufzeitrobuste Lösung




```
signal T, I, I_por,
    I_tast: std_logic;
...
I_por <= '0', '1'
  after 1 ms;
process(T)
begin
 if rising_edge(T) then
  if I_por='0' or I_tast='0' then I<='1';
  else I<='0';
  end if;
 end if;
end process;
```

Bildung eines Gesamtinitialisierungssignals aus Power-On-Reset und Tasten-Reset und Ausrichtung an der aktiven Taktflanke.

Mit Taktübertragung

Unterschiedlich getaktete Teilsysteme

Beim Datenaustausch zwischen Teilsystemen mit zeitlich unausgerichteten Takten benötigt der Empfänger eine Zusatzinformation über die Gültigkeitsfenster:

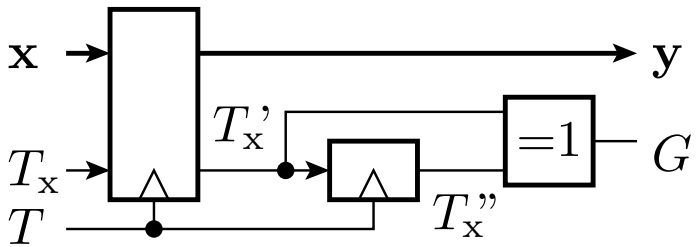
- Mitübertragung des Sendetakts zum Empfänger oder
- Rekonstruktion der Gültigkeitsfenster aus den Änderungszeitpunkten der empfangenen Daten.

Bei einer parallelen Schnittstelle wird meist der Sendetakt mit übertragen.

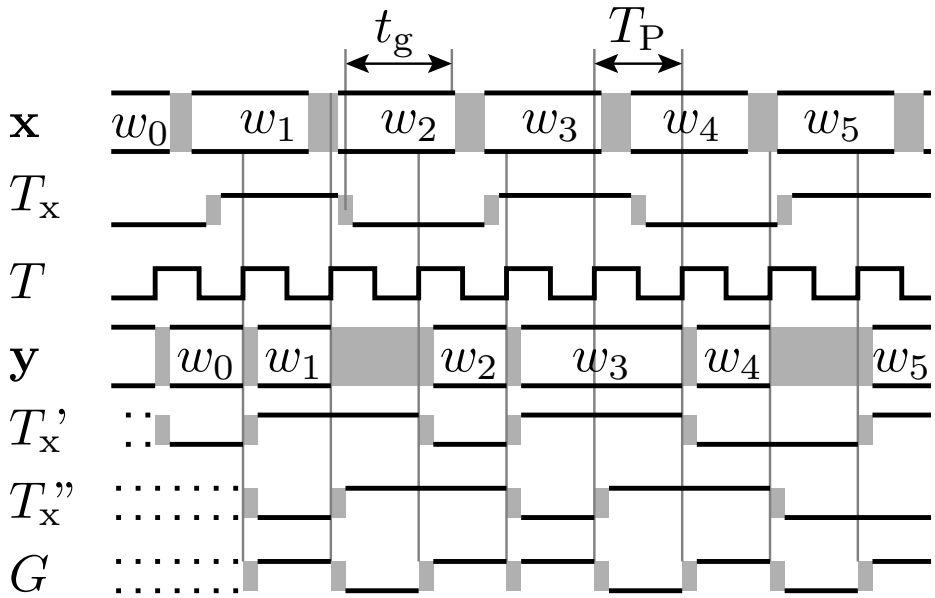
Empfangene Daten werden zuerst abgetastet. Im Anschluss wird die Gültigkeit der abgetasteten Signale mit einer laufzeittoleranten Register-Transfer-Funktion bestimmt.

Parallele Schnittstelle

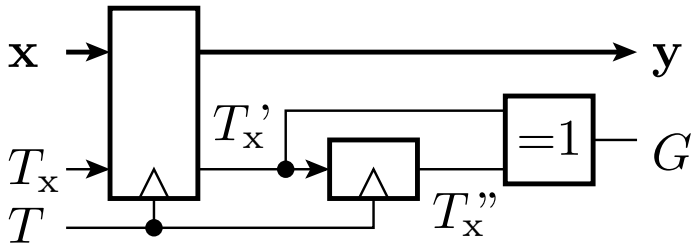
Im nachfolgenden Beispiel wird mit beiden (der steigenden und der fallenden) Sendetaktflanke ein Datenwort übertragen. Die Daten seien jeweils ab der Taktflanke von T_x für eine Dauer t_g gültig.



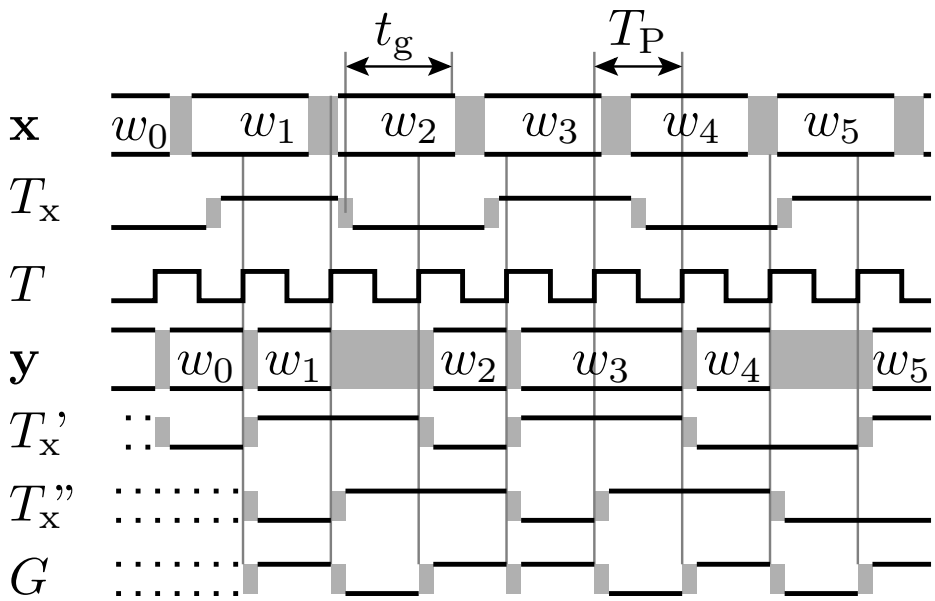
x	asynchrones Eingabesignal
T_x	Sendetakt
T	Systemtakt
y	abgetastetes Eingabesignal
G	Gültigkeitssignal für **x**'



Der Empfänger muss Daten und Takt mit einer Taktperiode $T_\mathrm{p} < t_\mathrm{g}$ abtasten, um jeden gültigen Wert mindestens einmal zu erfassen.



x	asynchrones Eingabesignal
T_x	Sendetakt
T	Systemtakt
y	abgetastetes Eingabesignal
G	Gültigkeitssignal für **x**'

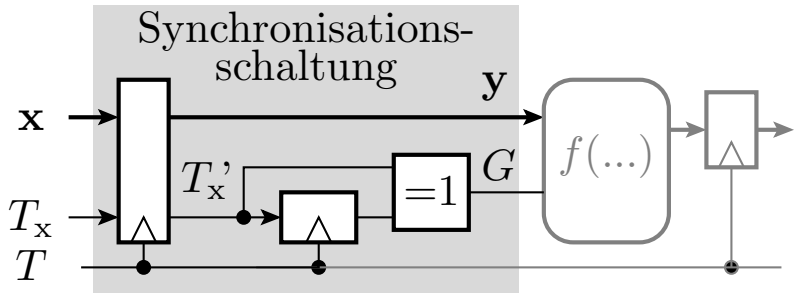


Die Eingabedaten sind immer genau dann neu und gültig, wenn sich der Abtastwert des Sendetakts vom vorherigen Abtastwert unterscheidet d.h., wenn die EXOR-Verknüpfung des ein- und zweimal abgetasteten Sendetakts eins ist.

VHDL-Beschreibung der Synchronisationsschaltung

Bildung eines Gesamtinitialisierungssignals und Ausrichtung an der aktiven Taktflanke durch Abtasten.

```
signal T, Tx, Tx_del, Tx_del2, G: std_logic;
signal x, y: std_logic_vector(...);
  ...
process(T)
begin
 if rising_edge(T) then
  y <= x;
  if Tx='1' then Tx_del <= '1';
  else Tx_del <= '0';
  end if;
  Tx_del2 <= Tx_del;
 end if;
end process;
G <= Tx_del xor Tx_del2;
```

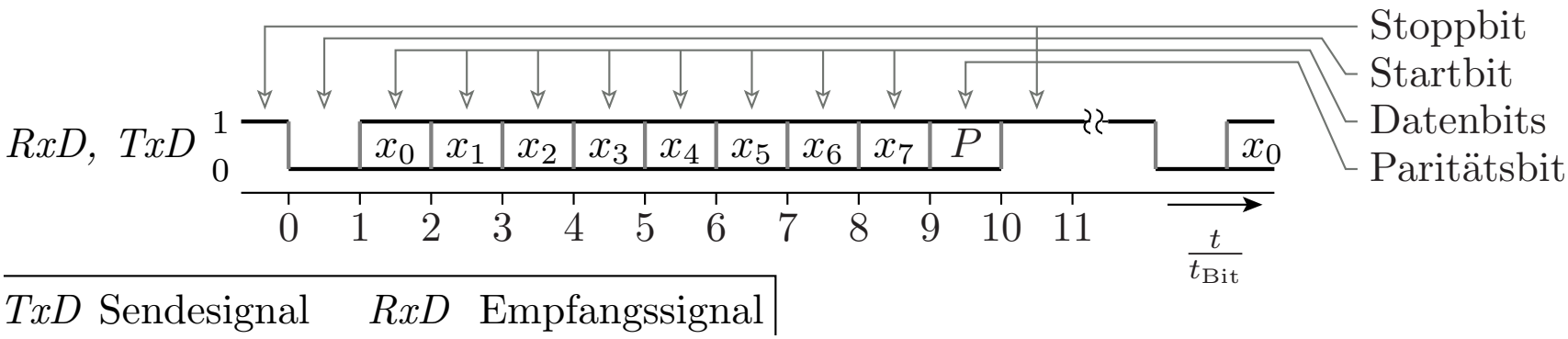


Ohne Taktübertragung

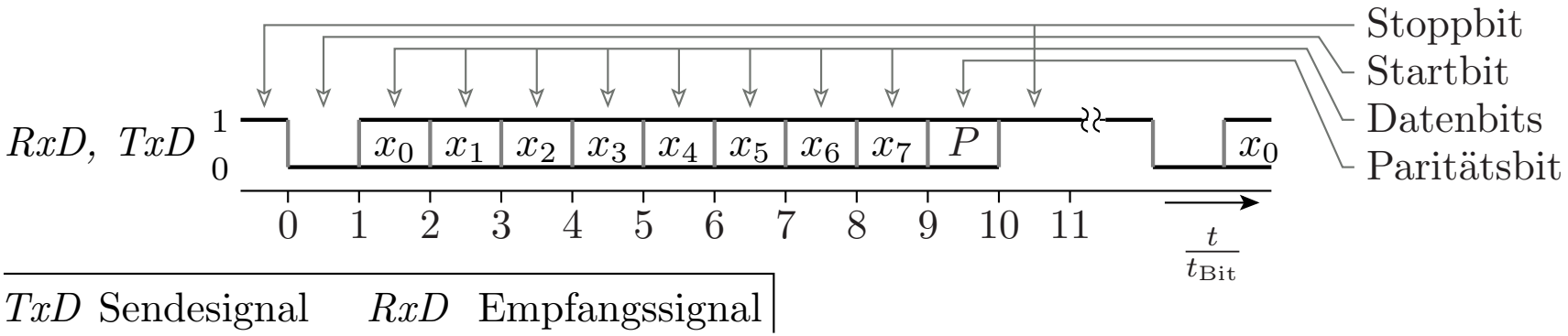
Gültigkeitszeitpunkte aus den Signalflanken

Beispiel sei das UART[6]-Protokoll. Der Sender sendet

- nach der letzten Übertragung für mindestens t_{Bit} eins,
- für genau t_{Bit} null (Startbit),
- für je genau t_{Bit} ein Datenbit,
- optional für t_{Bit} ein Paritätsbit und
- anschließend für $\geq 1 \cdot t_{\text{Bit}}$ oder $\geq 2 \cdot t_{\text{Bit}}$ eins (Stoppbit(s)).



[6] Universal Asynchronous Receiver and Transmitter, genormte Schnittstelle für den Datenaustausch zwischen PCs und Mikrorechnern.



Der Empfänger tastet das ankommende Bitsignal mit einer Periode

$$T_{\mathrm{P}} \ll t_{\mathrm{Bit}}$$

(typ. $T_{\mathrm{P}} = \frac{t_{\mathrm{Bit}}}{16}$). Nach Erkennung einer fallenden Flanke werden nach $1{,}5 \cdot t_{\mathrm{Bit}}$ (24 Takte), $2{,}5 \cdot t_{\mathrm{Bit}}$ (40 Takte) etc. die abgetasteten Bitwerte für gültig erklärt und weiter verarbeitet (siehe später Foliensatz EDS-F4).

Voraussetzungen für Datenübertragungen ohne Takt:

- eindeutige Startkennung, hinreichend häufig Signalflanken,
- Empfängertakt ein bekanntes Vielfaches der Bitzeit, ...

Zusammenfassung

Die asynchrone Datenübernahme erfordert spezielle laufzeitrobuste Schaltungsstrukturen:

- bitweise Abtastung vor der Verarbeitung,
- Abtastperiode größer Prelldauer,
- Synchronisation aller asynchron wirkenden Signale (Initialisierungssignale, Freigabesignale von Latches, ...
- Bei der Datenübertragung zuerst abtasten und dann die Gültigkeit prüfen.

Die asynchrone Übertragung verlangt die Definition eines Protokolls, dass u.a. Signalmerkmale zur Erkennung des Übertragungsbeginns und der Gültigkeitszeitpunkte aus dem abgetasteten Signal definiert.

Vergessene Abtastung hat manchem Studierenden im Arbeitsbereich eine wochenwährende Fehlersuche beschert.

Schaltungsoptimierung

Kenngrößen und Ziele für die Optimierung

Entwurfsziele für digitale Schaltungen (Minimierung / Obergrenzen):

- Aufwand in Transistoren, Chipfläche, ...
- Taktfrequenz, Vor- und Nachhaltezeiten, Verzögerungen, ...
- Energieverbrauch.

Obergrenze der Verzögerung der meisten Datenpfade ist die Taktperiode, vorgebbar als Constraint. Automatische Lösungssuche durch mehrfache Synthese, Implementierung und Verdrahtung.

Obergrenze der Schaltungsgröße: In den Laborübungen Resourcen des PLDs. Wenn die Schaltung nicht auf den Chip passt, Nachbesserung der Beschreibung ...

Für den Energieverbrauch gibt es ein Schätztool. Wenn zu hoch, Nachbesserung der Beschreibung ...

Energieverbrauch

Leistungsaufnahme

Elektronische Schaltungen wandeln elektrische Energie in Wärme um:

$$W \approx U_\mathrm{V} \cdot \bar{I}_\mathrm{V} \cdot t_\mathrm{B}$$

(U_V – Versorgungsspannung; $\bar{I}_\mathrm{V}$ – mittlerer Versorgungsstrom, t_B – Betriebsdauer).

Die heutigen digitalen Schaltungen bestehen aus FCMOS-Gattern. FCMOS-Gatter verbrauchen hauptsächlich während der Schaltvorgänge Strom. Beim Entwurf optimierbare Einflussfaktoren auf die Leistungsaufnahme:

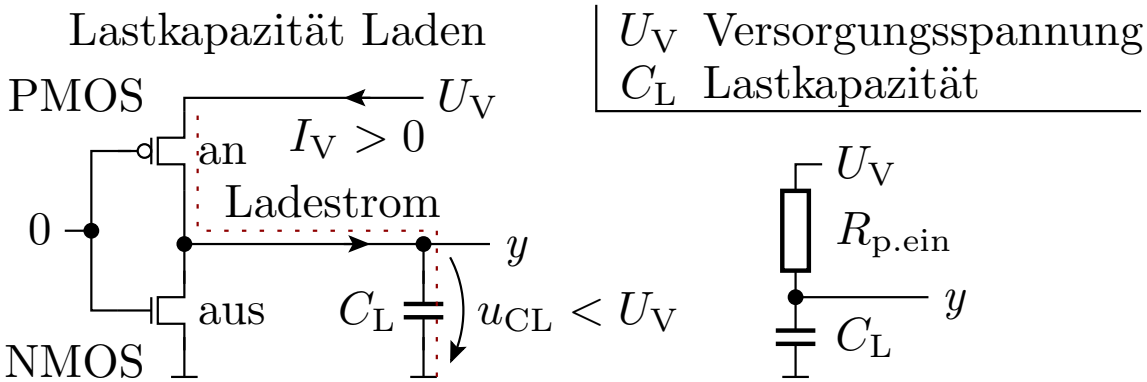
- Anzahl der Gatter + Schaltvorgänge je Gatter und Zeit oder
- Anzahl der Berechnungen und
- Anzahl der Schaltvorgänge je Berechnung.

Für batteriebetriebene Geräte ist geringer Energieverbrauch wichtig.

Leistungsumsatz an einem FCMOS-Inverter

Ein FCMOS-Inverter besteht aus

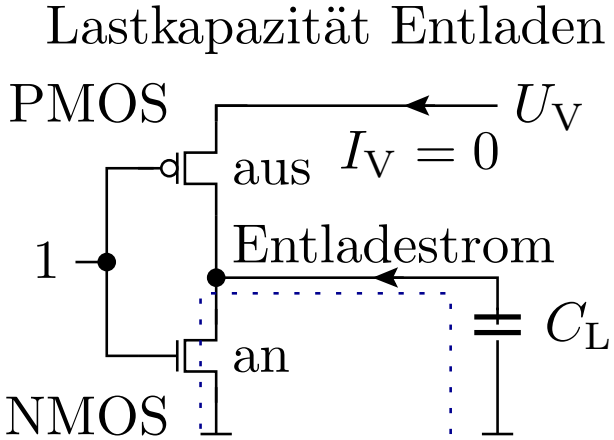
- einem PMOS-Transistor, der bei einer '0' am Eingang einschaltet und die Lastkapazität auf U_V auflädt und
- einem NMOS-Transistor, der bei einer '1' am Eingang einschaltet und die Lastkapazität entlädt.



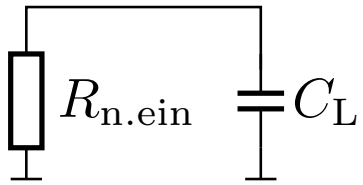
Energie zum Aufladen Lastkapazität (50% gespeichert, 50% Wärme):

$$W_\mathrm{CL} \approx U_\mathrm{V}^2 \cdot C_\mathrm{L}$$

$U_\mathrm{V} = 3{,}3\,\mathrm{V}$, $C_\mathrm{L} = 100\,\mathrm{fF} \Rightarrow W_\mathrm{CL} \approx 1\,\mathrm{pWs}$; 10^{12} Schaltvorgänge je Ws.



U_V Versorgungsspannung
C_L Lastkapazität



Beim Entladen der Lastkapazität wird nur gespeicherte Energie in Wärme umgesetzt.

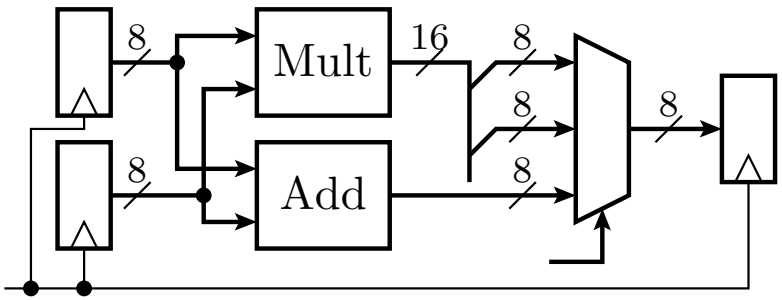
Logische Verknüpfungen werden durch Reihen- und Parallelschaltung von Transistoren im PMOS- und NMOS-Zweig realisiert (siehe später Foliensatz F5). Energieumsatz pro Logikgatter pro Schaltvorgang bei gleichem C_L wie beim Inverter.

- Minimierung der Anzahl der Schaltvorgänge,
- Taktfrequenz nur so hoch wie nötig, Glitches vermeiden,
- Minimierung der Anzahl der Rechenschritte, ...

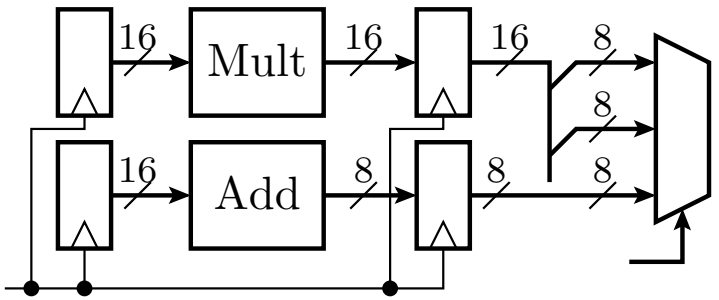
Schaltungserweiterung zur Energieeinsparung

Nachfolgende Schaltung berechnet von zwei 8-Bit-Operanden Summe und Produkt und wählt für die Übernahme in das Zielregister mit »op« zwischen Summe, niederwertigem und höherwertigem Produktbyte.

weniger Schaltungsaufwand



weniger Stromverbrauch



Für den Stromverbrauch wäre es günstiger, wenn nicht zu jeder benötigten Operation eine Parallelberechnung eines ungenutzten Ergebnisses erfolgt und nicht jedes Produkt zweimal gebildet werden muss, einmal für das niederwertige und einmal für das höherwertige Produktbyte. Das verlangt getrennte Operanden und Ergebnisregister für beide Rechenwerke, d.h. in Summe 56, statt 24 Registerbits.

Schaltungsumformung

Schaltungsumformungen

Gleiche Aufgaben lassen sich mit verschiedenen Funktionen lösen, gleiche Funktionen mit unterschiedlichen Schaltungen nachbilden.

Schaltungsumformung: Überführung in eine funktionsgleiche andere Schaltung mit anderen Aufwandskenngrößen (Größe, Geschwindigkeit, Energieverbrauch.). Es gibt

- algorithmische Umformungen, z.B. Berechnung von Winkelfunktionen mit dem Cordic-Algorithmus statt über Taylor-Reihen (siehe später Foliensatz F6).
- arithmetische Umformungen, z.B. Ausklammern zur Einsparung von Operationen und Rechenwerken:

$$a \cdot b + a \cdot c = a \cdot (b + c)$$

- und logische Umformungen.

Konstantenelimination und Verschmelzung

Zwei Grundregeln der Aufwandsoptimierung, die auch bei jeder Code-Optimierung für Software eingesetzt werden, sind:

- Konstantenelimination: Ersatz von Berechnungsschritten mit konstantem Ergebnis durch ihren Wert bei der Übersetzung, und
- Verschmelzung: Mehrfach genutzte Zwischenergebnisse nur einmal berechnen.

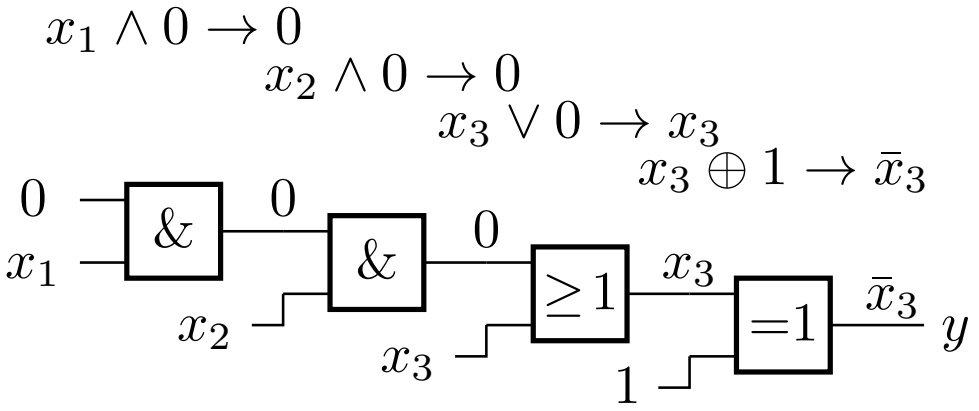
Beispiel für die Anwendung der ersten Regel sei die Vereinfachung eines logischen Ausdrucks mit konstanten Operanden:

$$\begin{aligned}((((x_1 \wedge 0) \wedge x_2) \vee x_3) \oplus 1) &= (((0 \wedge x_2) \vee x_3) \oplus 1) \\ &= ((0 \vee x_3) \oplus 1) \\ &= (x_3 \oplus 1) \\ &= \bar{x}_3\end{aligned}$$

$$((((x_1 \wedge 0) \wedge x_2) \vee x_3) \oplus 1) = \bar{x}_3$$

Der ursprüngliche Ausdruck hat vier logische Operatoren, aus denen bei der Übersetzung in eine Schaltung vier Gatter werden. Der vereinfachte Ausdruck $\bar{x}_3$ verlangt nur einen einzigen Inverter:

Schaltung für den Originalausdruck



vereinfachte Schaltung

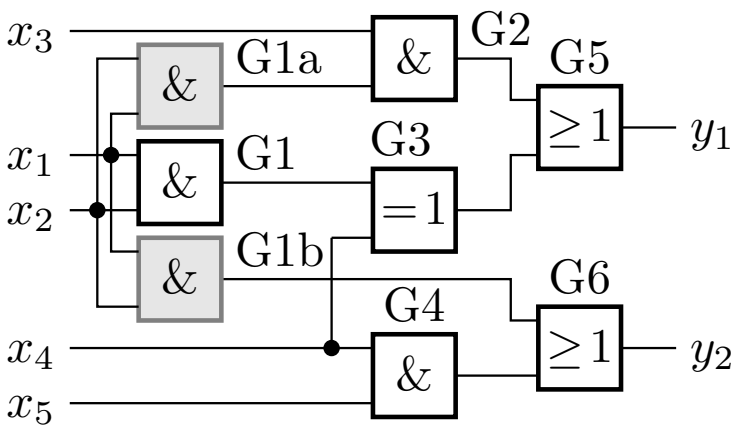
Als Beispiel für die Mehrfachnutzung von Zwischenergebnissen soll folgende Berechnung von zwei Ausgabe- aus vier Eingabebits dienen:

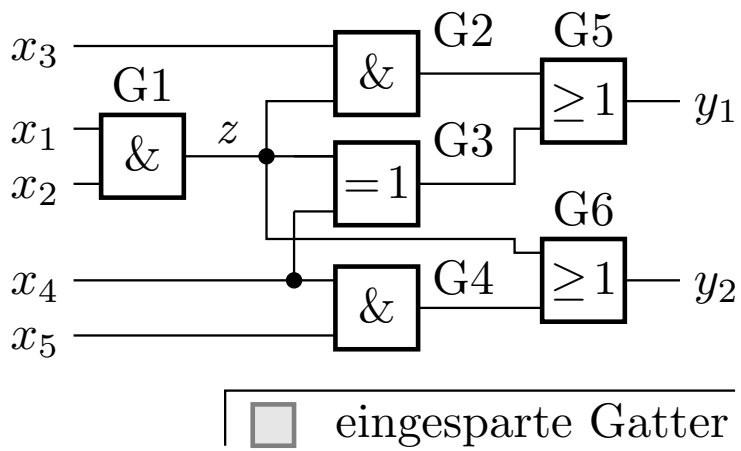
$$\begin{aligned} y_1 &:= x_3x_2x_1 \vee (x_4 \oplus x_2x_1) \\ y_2 &:= x_5x_4 \vee x_2x_1 \end{aligned}$$

Im ersten Ausdruck auf der rechten Zuweisungsseite ist der Term x_2x_1 zwei- und im zweiten einmal vorhanden. Vereinfachter Berechnungsfluss:

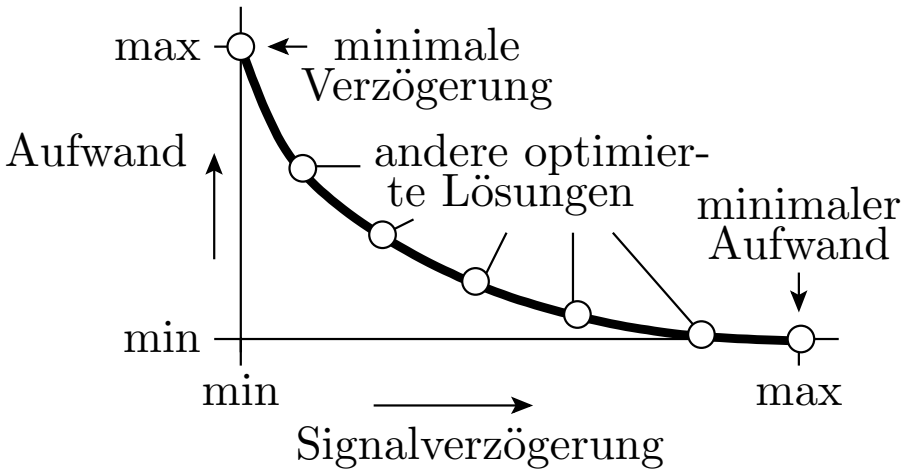
$$\begin{aligned} z &:= x_2x_1 \\ y_1 &:= x_3z \vee (x_4 \oplus z) \\ y_2 &:= x_5x_4 \vee z \end{aligned}$$

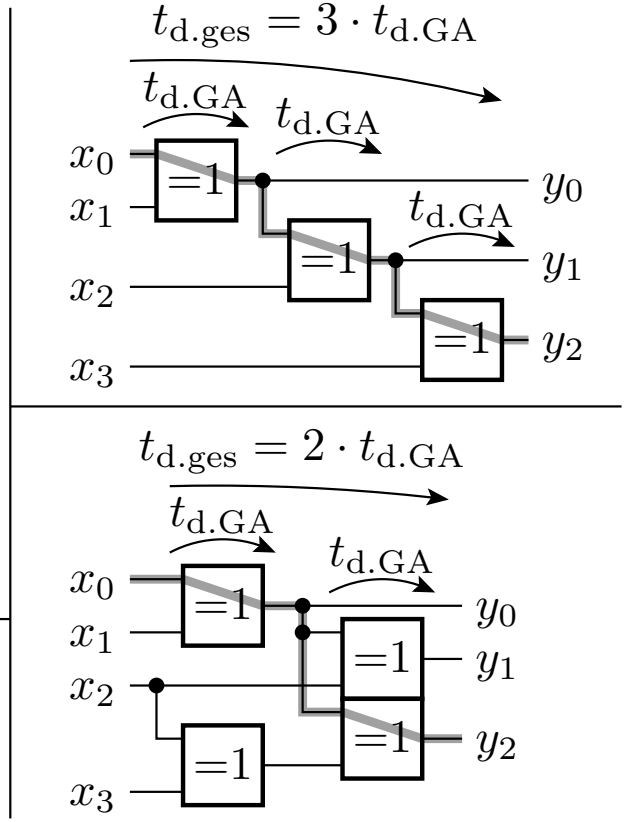
Vereinfachung auf der Schaltungsebene:





Die kleinste ist nicht immer die schnellste Schaltung





längster Pfad

$t_{d.GA}$ maximale Gatterverzögerung

$t_{d.ges}$ maximale Gesamtverzögerung

Die Realisierung mit drei Gattern hat einen längsten Pfad von drei Gatterverzögerungszeiten und die mit vier Gattern nur von zwei Gatterverzögerungszeiten.

Umformungsregeln für logische Ausdrücke

Umformungsregel	Bezeichnung
$\bar{\bar{x}} = x$	doppelte Negation
$x \vee 1 = 1$ $x \vee \bar{x} = 1$ $x \wedge 0 = 0$ $x \wedge \bar{x} = 0$	Eliminationsgesetze
$x_1 \vee (x_1 \wedge x_2) = x_1$ $x_1 \wedge (x_1 \vee x_2) = x_1$	Absorbtionsgesetze
$\overline{x_1 \wedge x_2} = \bar{x}_1 \vee \bar{x}_2$ $\overline{x_1 \vee x_2} = \bar{x}_1 \wedge \bar{x}_2$	De Morgansche Regeln[7]

[7] Nach Augustus De Morgan (1806 - 1871) englischer Mathematiker.

Umformungsregeln für logische Ausdrücke

Umformungsregel	Bezeichnung
$x_1 \wedge x_2 = x_2 \wedge x_1$ $x_1 \vee x_2 = x_2 \vee x_1$	Kommutativgesetze
$(x_1 \vee x_2) \vee x_3 =$ $x_1 \vee (x_2 \vee x_3)$ $(x_1 \wedge x_2) \wedge x_3 =$ $x_1 \wedge (x_2 \wedge x_3)$	Assoziativgesetze
$x_1 \wedge (x_2 \vee x_3) =$ $(x_1 \wedge x_2) \vee (x_1 \wedge x_3)$ $x_1 \vee (x_2 \wedge x_3) =$ $(x_1 \vee x_2) \wedge (x_1 \vee x_3)$	Distributivgesetze

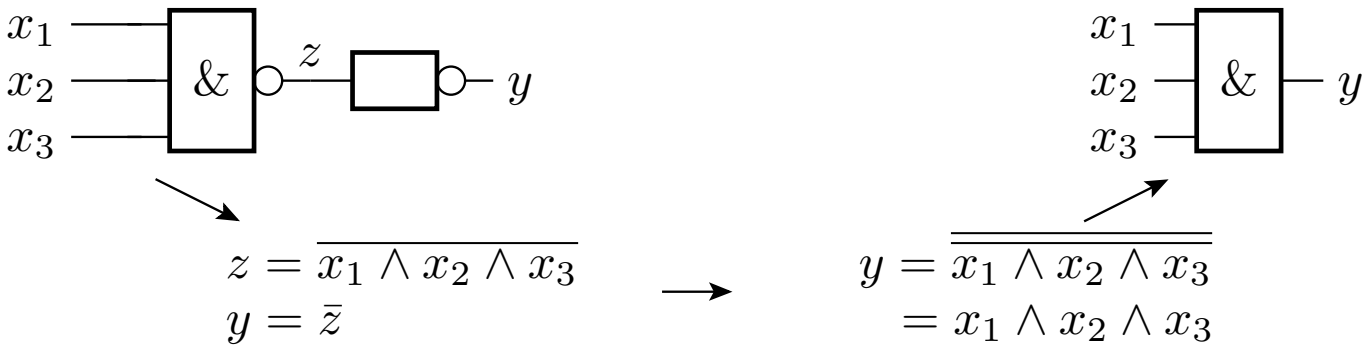
Beweis der Umformungsregeln

- Aufstellen und Vergleich der Wertetabellen.
- Für die De Morganschen Regeln gilt z.B.:

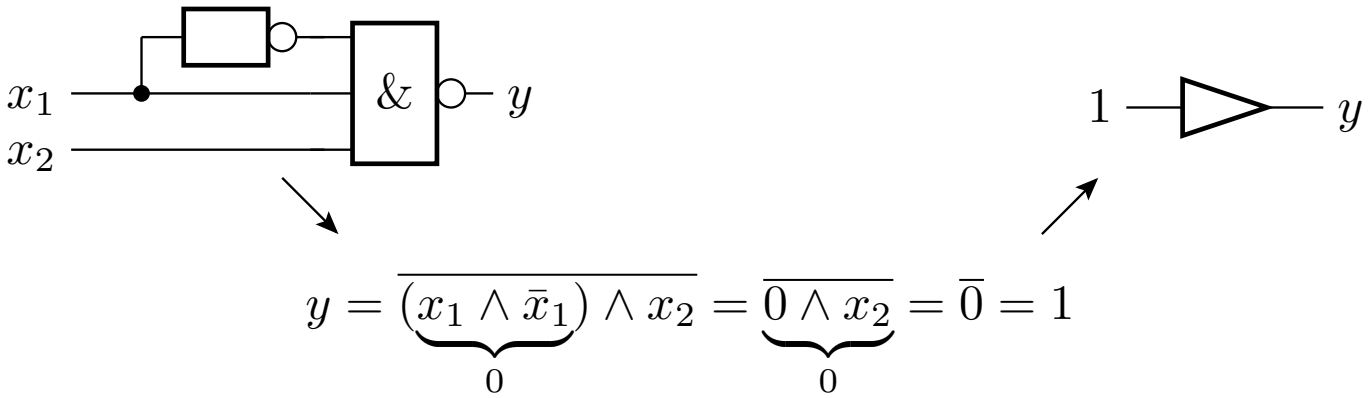
x_1	x_2	$\bar{x}_1 \vee \bar{x}_2$	$\overline{x_1 \wedge x_2}$	$\bar{x}_1 \wedge \bar{x}_2$	$\overline{x_1 \vee x_2}$
0	0	1	1	1	1
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	0	0

Umformungen zur Schaltungsvereinfachung

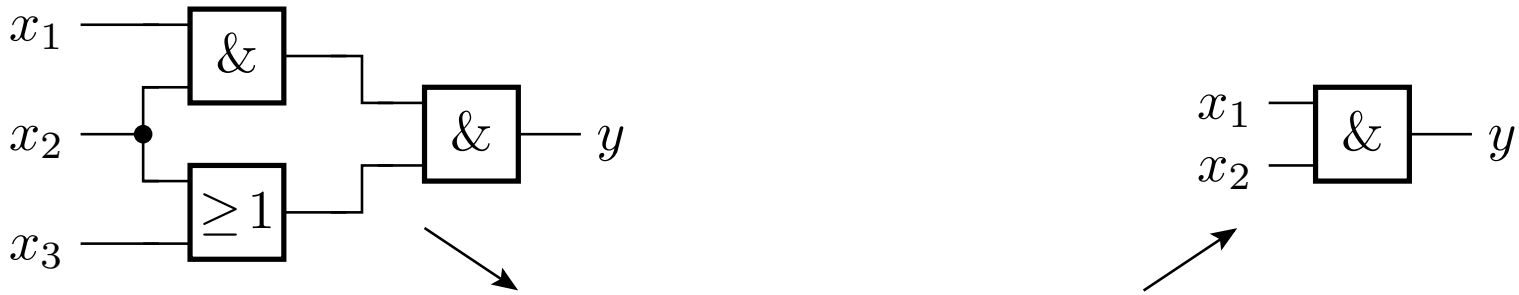
Mehrfache Negationen im Signalfluss heben sich paarweise auf.



Doppelte Anwendung der Eliminationsgesetze:



Anwendung des Absorbtionsgesetzes:



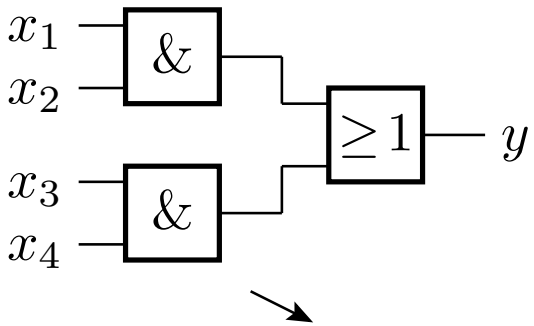
gegebene Funktion: $y = (x_1 \wedge x_2) \wedge (x_2 \vee x_3)$
Assoziativgesetz: $y = x_1 \wedge (x_2 \wedge (x_2 \vee x_3))$
Absorbtionsgesetz: $y = x_1 \wedge x_2$

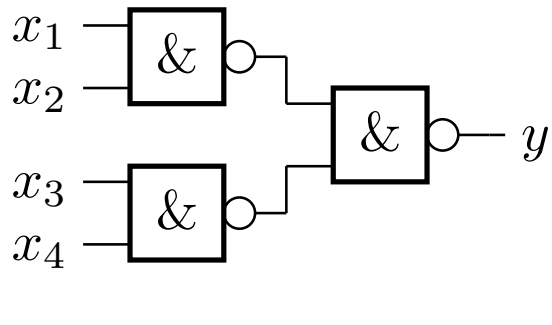
Die Vereinfachungsregeln selbst sind unkompliziert. Problematisch ist die Strukturierung und Größe der Suchräume nach Anwendungsmöglichkeiten der Vereinfachungsregeln. In den Folgeabschnitten werden folgende Beispiele dafür vorgestellt:

- KV-Diagramm,
- Verfahren von Quine und McCluskey und
- die Vereinfachung mit binären Entscheidungsdiagrammen.

AND-OR nach NAND-NAND

Eine AND-OR-Schaltung lässt sich in eine fast strukturgleiche NAND-NAND-Schaltung umwandeln. Das war früher beim Entwurf mit Standardschaltkreisen eine ganz wichtige Vereinfachung, weil das die erforderliche Anzahl unterschiedlicher Schaltkreistypen verringert hat.



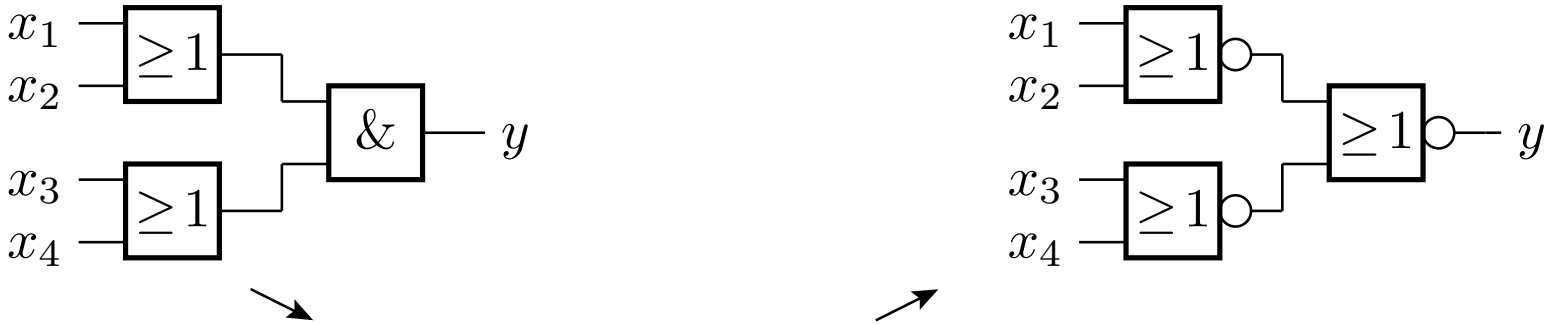


gegebene Funktion: $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

doppelte Negation: $y = \overline{\overline{(x_1 \wedge x_2) \vee (x_3 \wedge x_4)}}$

De Morgansche Regel: $y = \overline{\overline{x_1 \wedge x_2} \wedge \overline{x_3 \wedge x_4}}$

Eine ähnliche Vereinfachung ist auch mit OR-AND-Schaltungen möglich, die durch NOR-NOR-Schaltungen ersetzt werden können.



gegebene Funktion: $y = (x_1 \vee x_2) \wedge (x_3 \vee x_4)$

doppelte Negation: $y = \overline{\overline{(x_1 \vee x_2) \wedge (x_3 \vee x_4)}}$

De Morgansche Regel: $y = \overline{\overline{x_1 \vee x_2} \vee \overline{x_3 \vee x_4}}$

KV-Diagramm

Logikminimierung mit Konjunktionsmengen

Bleistift-und-Papier-Verfahren für die Optimierung von Gatterschaltungen mit $\leq$4 Eingabebits. Es basiert auf der Minimierung von Konjunktions- bzw. Disjunktionsmengen.

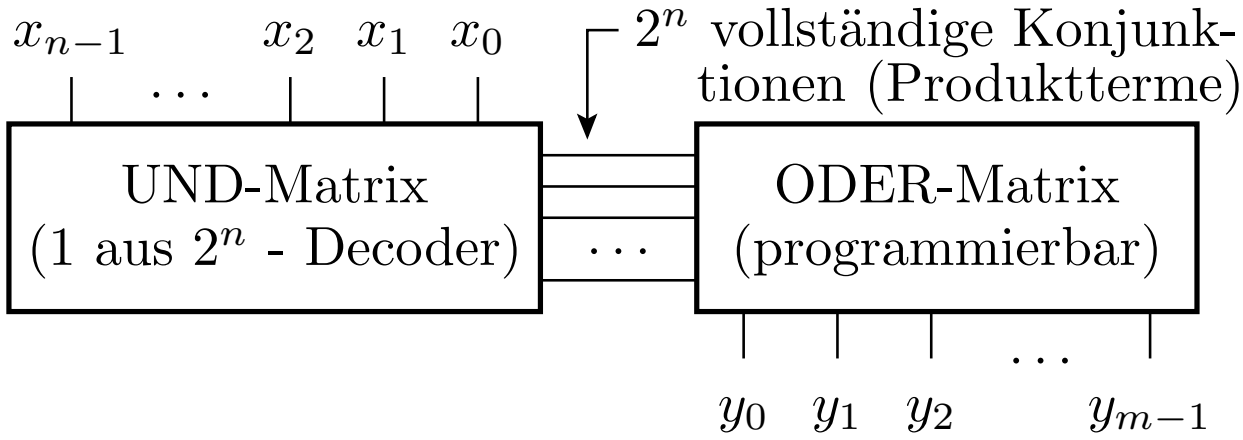
Konjunktion: Term, der direkte oder invertierte Eingabevariablen UND-verknüpft, z.B.:

$$x_3 \wedge \bar{x}_2 \wedge x_1 \wedge \bar{x}_0 = \underbrace{x_3\bar{x}_2x_1\bar{x}_0 \ (K_{1010})}_{\text{verkürzte Schreibweisen}}$$

Minterm: Konjunktion, die alle Eingabevariablen entweder in direkter oder in negierter Form enthält.

Satz: Jede logische Funktion lässt sich durch eine Menge von Konjunktionen darstellen, die ODER-verknüpft werden.

Beweis: Jede logische Funktion lässt sich durch eine Wertetabelle darstellen. Jeder Zeile einer Wertetabelle ist ein Minterm zugeordnet, der genau dann Eins ist, wenn die Zeile ausgewählt wird.



Die Funktion ist entweder die

- ODER-Verknüpfung der Minterme, für die $y = 1$ ist, oder
- negierte ODER-Verknüpfung der Minterme, für die $y = 0$ ist.

x_2	x_1	x_0	Konjunktion	y	x_2	x_1	x_0	Konjunktion	y
0	0	0	$\bar{x}_2\bar{x}_1\bar{x}_0$ (K_{000})	0	1	0	0	$x_2\bar{x}_1\bar{x}_0$ (K_{100})	1
0	0	1	$\bar{x}_2\bar{x}_1x_0$ (K_{001})	1	1	0	1	$x_2\bar{x}_1x_0$ (K_{101})	1
0	1	0	$\bar{x}_2x_1\bar{x}_0$ (K_{010})	0	1	1	0	$x_2x_1\bar{x}_0$ (K_{110})	1
0	1	1	$\bar{x}_2x_1x_0$ (K_{011})	0	1	1	1	$x_2x_1x_0$ (K_{111})	0

- Entwicklung nach den Einsen: $\{K_{001}, K_{100}, K_{101}, K_{110}\} \Rightarrow$

$$y = \bar{x}_2\bar{x}_1x_0 \vee x_2\bar{x}_1\bar{x}_0 \vee x_2\bar{x}_1x_0 \vee x_2x_1\bar{x}_0$$

- Entwicklung nach den Nullen: $\{K_{000}, K_{010}, K_{011}, K_{111}\} \Rightarrow$

$$\begin{aligned} y &= \overline{\bar{x}_2\bar{x}_1\bar{x}_0 \vee \bar{x}_2x_1\bar{x}_0 \vee \bar{x}_2x_1x_0 \vee x_2x_1x_0} \\ &= \overline{\bar{x}_2\bar{x}_1\bar{x}_0} \wedge \overline{\bar{x}_2x_1\bar{x}_0} \wedge \overline{\bar{x}_2x_1x_0} \wedge \overline{x_2x_1x_0} \\ &= (x_2 \vee x_1 \vee x_0)\,(x_2 \vee \bar{x}_1 \vee x_0)\,(x_2 \vee \bar{x}_1 \vee \bar{x}_0)\,(\bar{x}_2 \vee \bar{x}_1 \vee \bar{x}_0) \end{aligned}$$

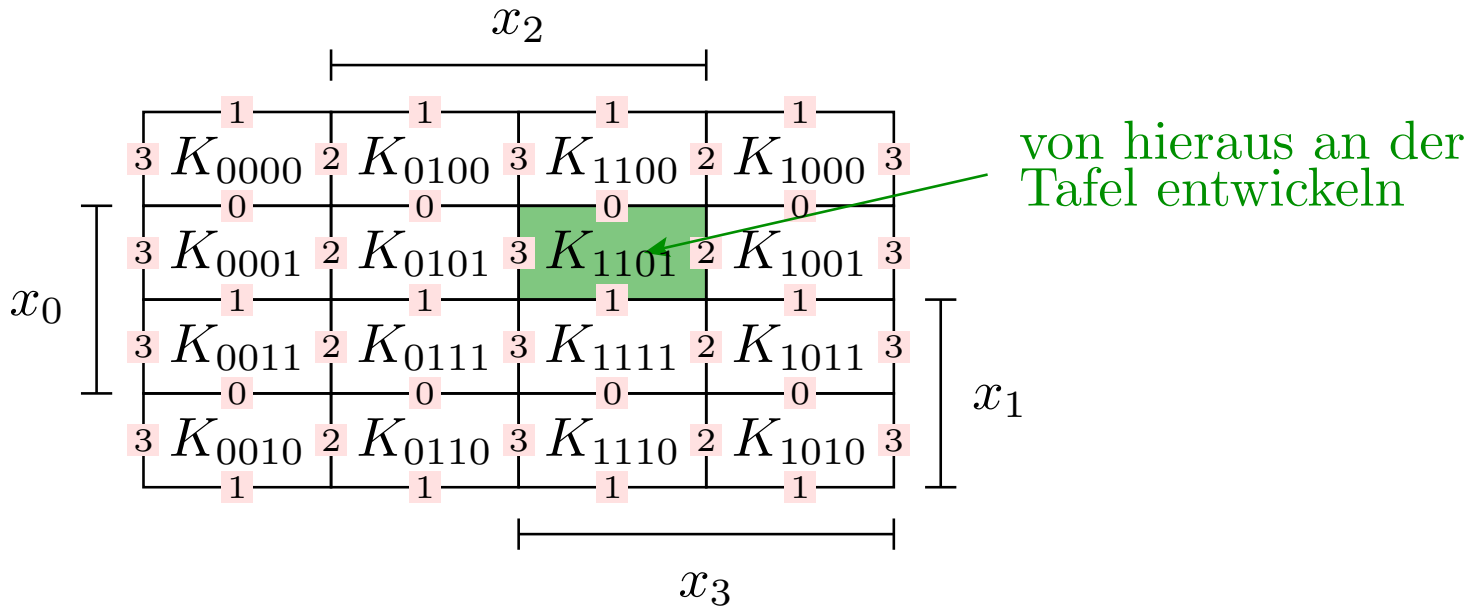
Vereinfachungsgrundlage

Satz: Zwei Konjunktionen, die sich nur in der Negation einer Variablen unterscheiden, können zu einer Konjunktion mit einer Variablen weniger zusammengefasst werden.

Schritt	ODER-Verknüpfung	Konjunktionsmenge
1	$\ldots \vee x_2\bar{x}_1\bar{x}_0 \vee x_2\bar{x}_1 x_0 \vee \ldots$	$\{\ldots, K_{100}, K_{101}, \ldots\}$
2	$\ldots \vee x_2\bar{x}_1 (\bar{x}_0 \vee x_0) \vee \ldots$	
3	$\ldots \vee x_2\bar{x}_1 \vee \ldots$	$\{\ldots, K_{10*}, \ldots\}$

1. Zwei Konjunktionen, die sich nur in der Negation von x_0 unterscheiden.
2. Ausklammern der UND-Verknüpfung der übrigen Variablen.
3. Konjunktion mit einer Variablen weniger.

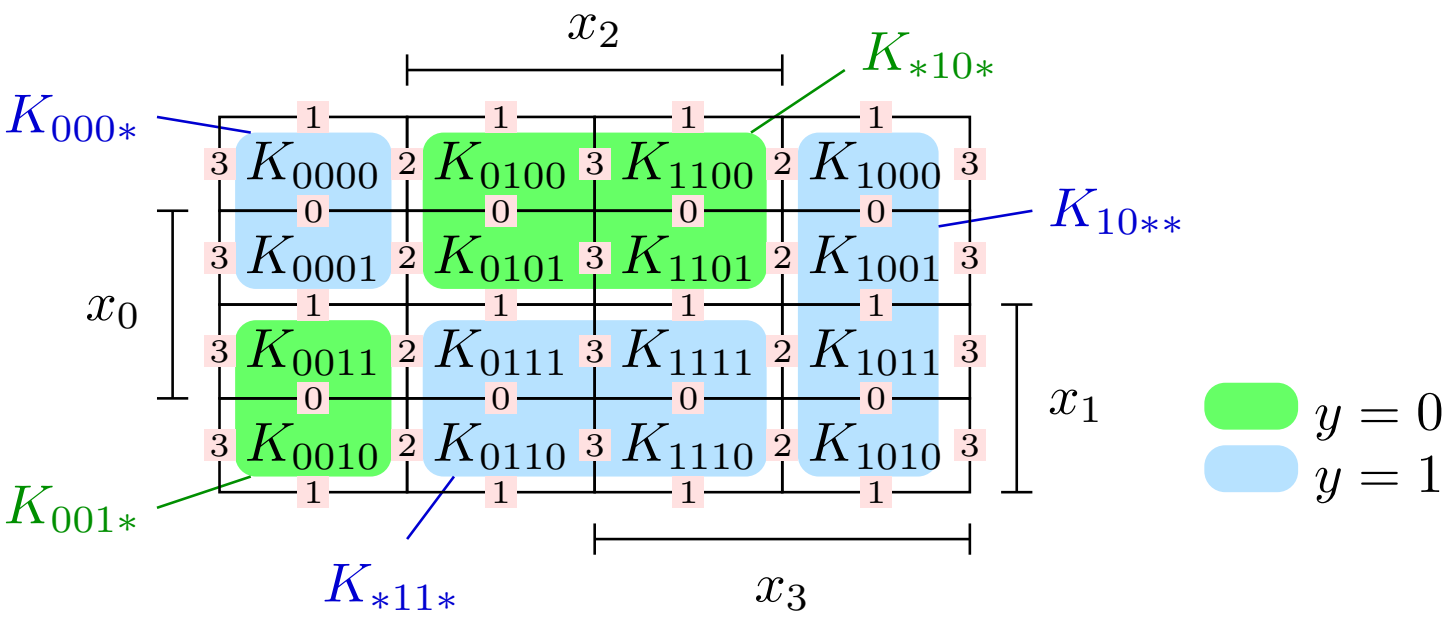
Aufbau eines KV-Diagramms[7]



Anordnung der Eingabewerte so, dass sich die Konjunktionen benachbarter Felder genau in einer Negation unterscheiden, so dass benachbarte Nullen oder Einsen zu einer Konjunktion mit einer Variablen weniger zusammengefasst werden können.

[7] Entwickelt von Maurice Karnaugh und Edward W. Veitch.

Zusammenfassen der ausgewählten Konjunktionen zu Blöcken der Kantenlänge eins, zwei oder vier:



- Minimierte Konjunktionsmenge der Einsen

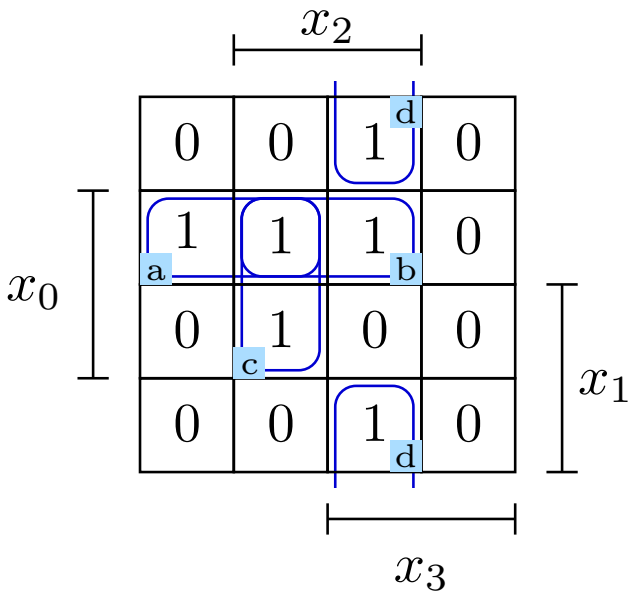
$$\{K_{000*},\ K_{*11*},\ K_{10**}\} \Rightarrow y = \bar{x}_3\bar{x}_2\bar{x}_1 \vee x_2x_1 \vee x_3\bar{x}_2$$

- Minimierte Konjunktionsmenge der Nullen:

$$\{K_{001*},\ K_{*10*}\} \Rightarrow y = \overline{\bar{x}_3\bar{x}_2x_1 \vee x_2\bar{x}_1}$$

Praktische Arbeit mit KV-Diagrammen

- Eintragen der Funktionswerte in das KV-Diagramm.
- Abdeckung der Einsen (der Nullen) mit Blöcken der Kantenlänge 1, 2 oder 4. Optimierungsziel: möglichst große und möglichst wenige Blöcke. Blöcke dürfen sich überlagern.
- Ablesen der Block-Konjunktionen und Zusammenfassen.



a: $\bar{x}_3\bar{x}_1x_0$

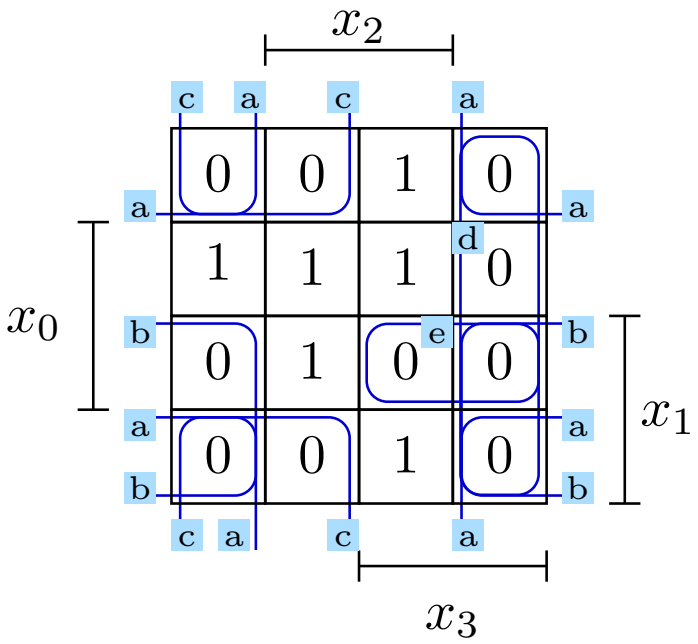
b: $x_2\bar{x}_1x_0$

c: $\bar{x}_3x_2x_0$

d: $x_3x_2\bar{x}_0$

$y = \bar{x}_3\bar{x}_1x_0 \vee x_3\bar{x}_1x_0 \vee \bar{x}_3x_2x_0 \vee x_3x_2\bar{x}_0$

Die Blockbildung kann auch zirkular über den Rand hinaus erfolgen. Bei einer Entwicklung nach den Nullen ist das Ergebnis die invertierte ODER-Verknüpfung der Konjunktionen.



a: $\bar{x}_2\bar{x}_0$

b: $\bar{x}_2 x_1$

c: $\bar{x}_3\bar{x}_0$

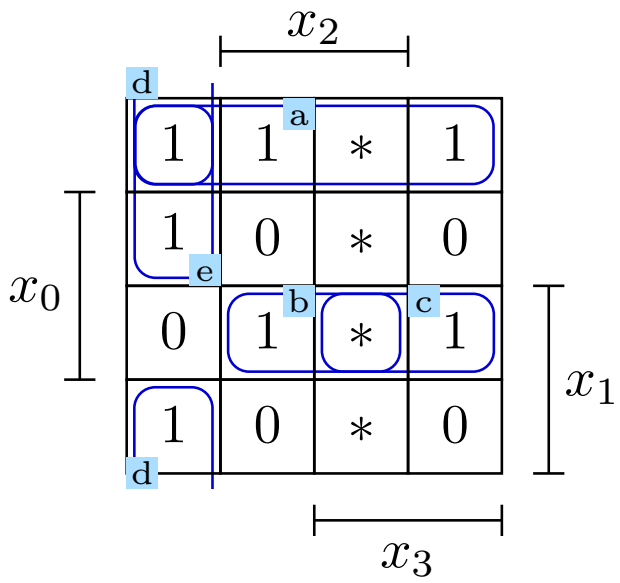
d: $x_3\bar{x}_2$

e: $x_3 x_1 x_0$

$$y = \overline{\bar{x}_2\bar{x}_0 \vee \bar{x}_2 x_1 \vee \bar{x}_3\bar{x}_0 \vee x_3\bar{x}_2 \vee x_3 x_1 x_0}$$

KV-Diagramme mit don't-care-Feldern

Don't-care-Felder kennzeichnen Eingabemöglichkeiten, die im normalen Betrieb nicht auftreten und für die folglich die Ausgabe ohne Bedeutung ist. Ihr Wert wird so festgelegt, dass sich möglichst wenige und möglichst große Blöcke bilden lassen.



a: $\bar{x}_1\bar{x}_0$

b: $x_2x_1x_0$

c: $x_3x_1x_0$

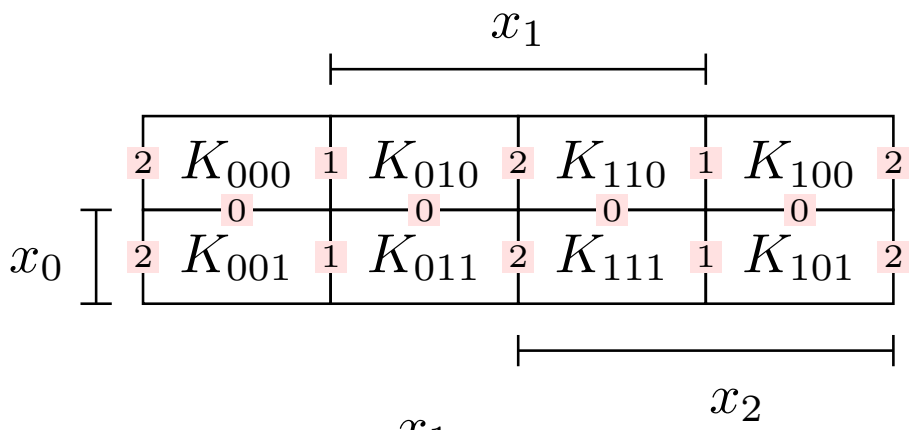
d: $\bar{x}_3\bar{x}_2\bar{x}_0$

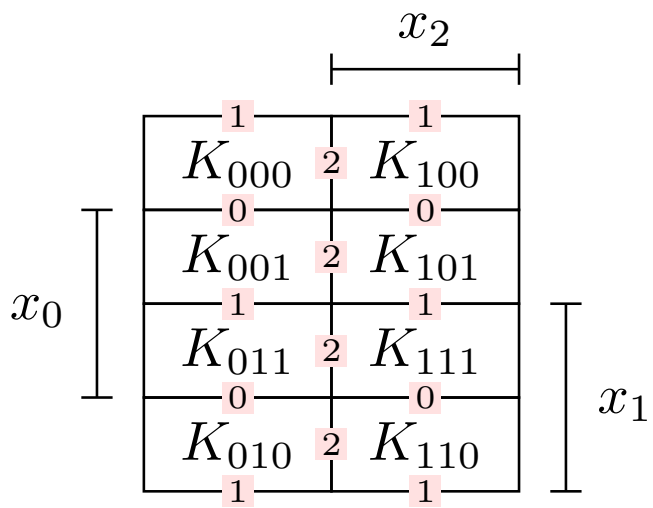
e: $\bar{x}_3\bar{x}_2\bar{x}_1$

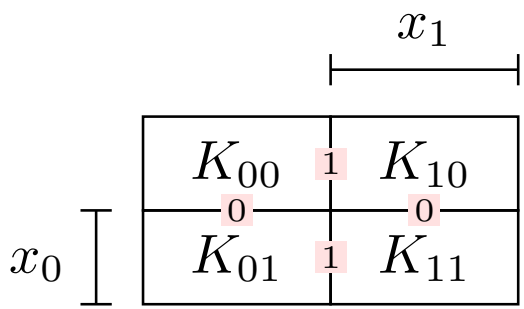
$$y = \bar{x}_1\bar{x}_0 \vee x_2x_1x_0 \vee x_3x_1x_0 \vee \bar{x}_3\bar{x}_2\bar{x}_0 \vee \bar{x}_3\bar{x}_2\bar{x}_1$$

Zwei- und dreistellige Funktionen

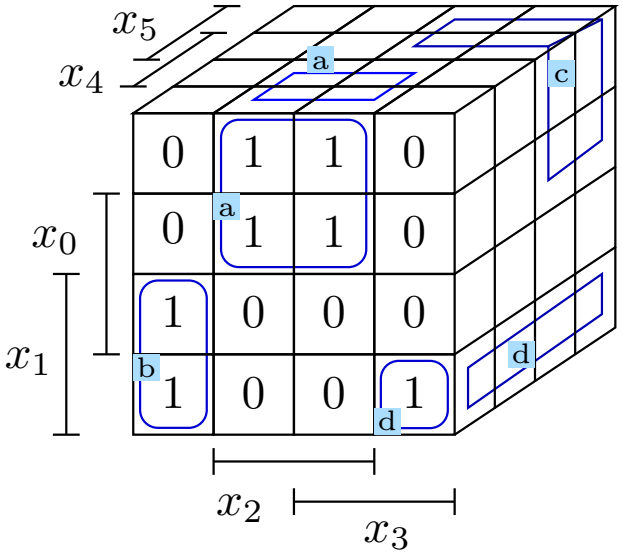
- Verringerung der Anzahl der Nachbarfelder, die sich in einer Negation unterscheiden, auf drei bzw. zwei.
- Halbierung der Höhe und/oder der Breite.

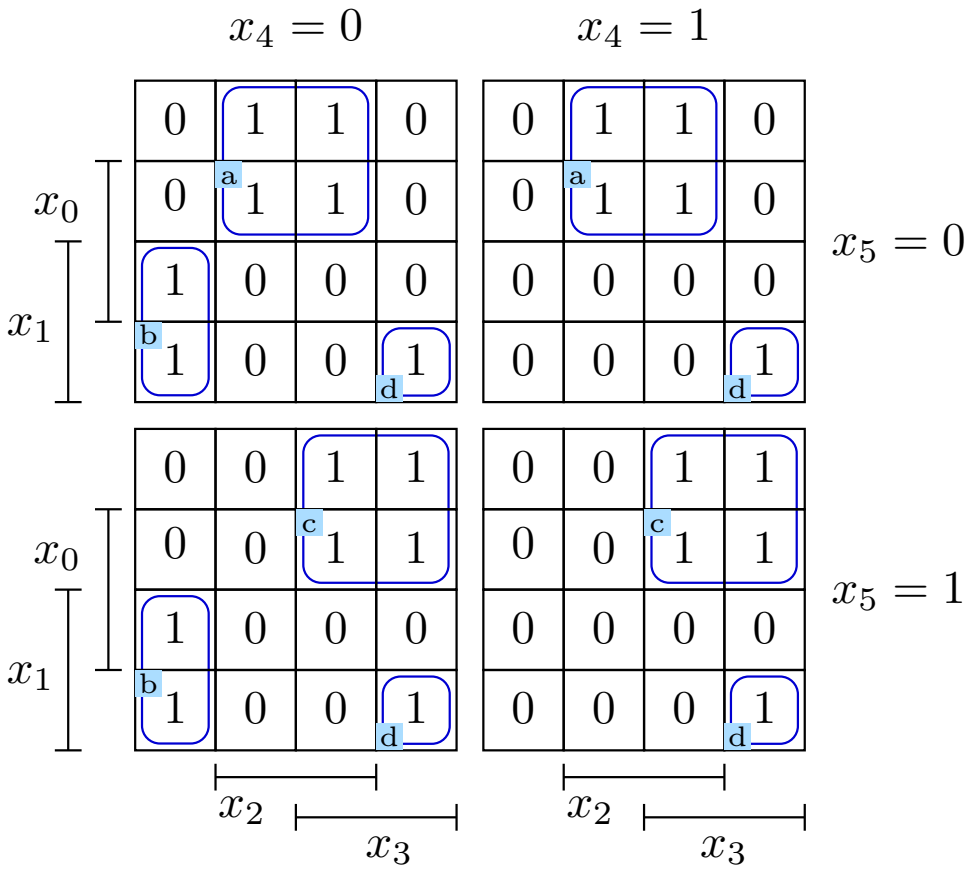






Erweiterung auf bis zu 6 Eingabebits





a: $\bar{x}_5 x_2 \bar{x}_1$

b: $\bar{x}_4 \bar{x}_3 \bar{x}_2 x_1$

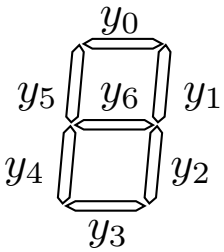
c: $x_5 x_3 \bar{x}_1$

d: $x_3 \bar{x}_2 x_1 \bar{x}_0$

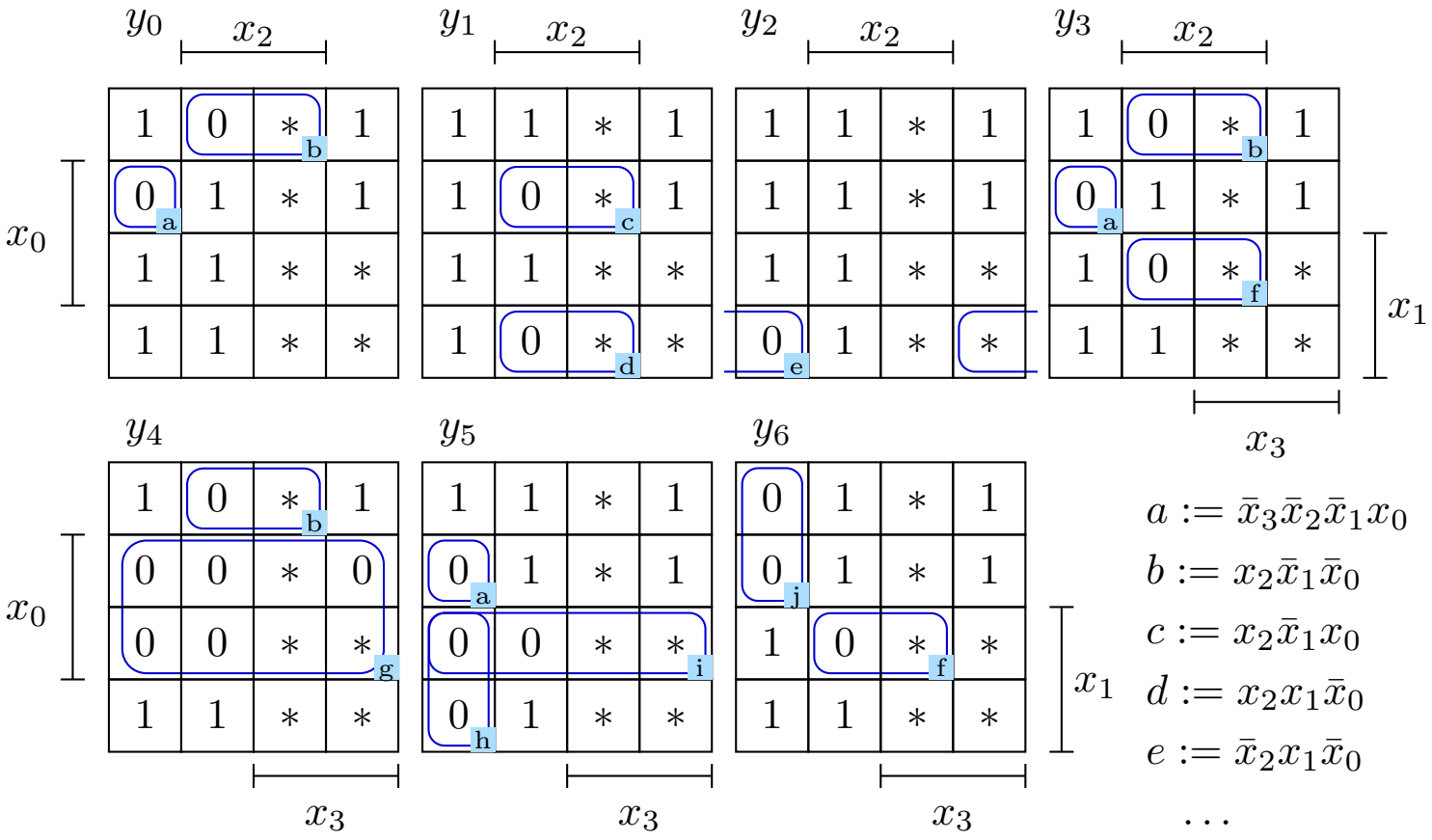
$y = \bar{x}_5 x_2 \bar{x}_1 \vee \bar{x}_4 \bar{x}_3 \bar{x}_2 x_1 \vee x_5 x_3 \bar{x}_1 \vee x_3 \bar{x}_2 x_1 \bar{x}_0$

Für eine 6-stellige Funktion muss jedes Element der Tabelle sechs Nachbarfelder haben. Verlangt eine 3D-Tabelle.

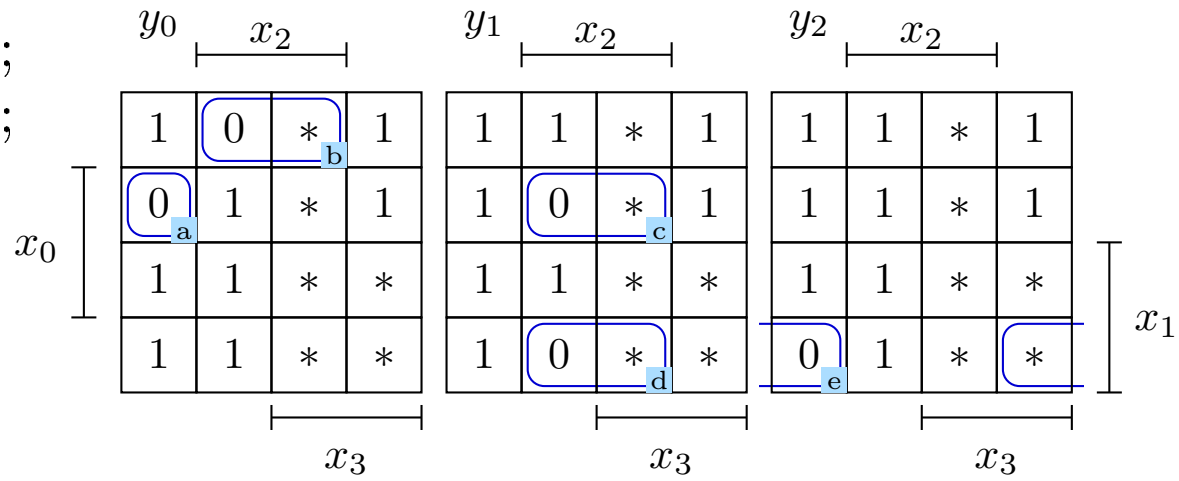
Schaltungen mit mehreren Ausgängen



x	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	sonst
y	0	1	2	3	4	5	6	7	8	9	belie-big



```
signal x: std_logic_vector(3 downto 0);
signal y: std_logic_vector(6 downto 0);
...
process(x)
 variable a,b,c,d,e,f,g,h,i,j: std_logic;
begin
 a := not x(3) and not x(2) and not x(1) and x(0);
 b := x(2) and not x(1) and not x(0);
 c := x(2) and not x(1) and x(0);
 d := x(2) and x(1) and not x(0);
 e := not x(2) and x(1) and not x(0);
 ...
 y(0)<=not(a or b);
 y(1)<=not(c or d);
 y(2)<=not e;
 ...
 end process;
```



Quine & McCluskey

Optimierung nach Quine & McCluskey – Prinzip

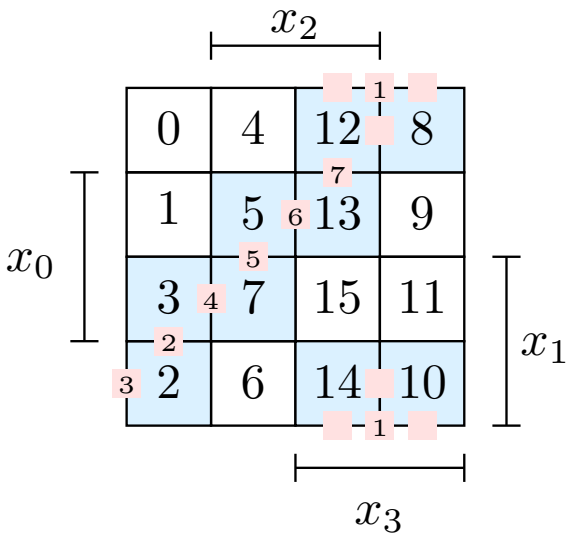
Tabellenbasiertes Minimierungsverfahren, das gleichfalls auf der Zusammenfassung von Konjunktionen, die sich nur in einer Negation unterscheiden, basiert:

$$\{\ldots, K_{100}, K_{101}, \ldots\} \Rightarrow \{\ldots, K_{10*}, \ldots\}$$

- Zusammenstellen der Menge aller Minterme, für die der Funktionswert eins (bzw. null) ist $\Rightarrow$ quinesche Tabelle nullter Ordnung
- Suche in der quineschen Tabelle nullter Ordnung alle Möglichkeiten zur Bildung einer Konjunktion mit einer don't-care-Stelle und abhaken der erfassten Konjunktionen $\Rightarrow$ quinesche Tabelle erster Ordnung.
- Dasselbe mit den quineschen Tabellen 2. bis n-ter Ordnung.
- Ausdrucksminimierung mit Hilfe der Abdeckungstabelle der Primterme.

Aufstellen der quineschen Tabellen

Visualisierung der Blockbildungsmöglichkeiten mit einem KV-Diagramm



- mögliche Zweierblöcke
- möglicher Viererblock
- √ abgedeckte Konjunktionen
- P_i Primterm

quinesche Tabelle nullte Ordnung

	x_3	x_2	x_1	x_0	
2	0	0	1	0	√
8	1	0	0	0	√
3	0	0	1	1	√
5	0	1	0	1	√
10	1	0	1	0	√
12	1	1	0	0	√
7	0	1	1	1	√
13	1	1	0	1	√
14	1	1	1	0	√

quinesche Tabellen erste und zweite Ordnung

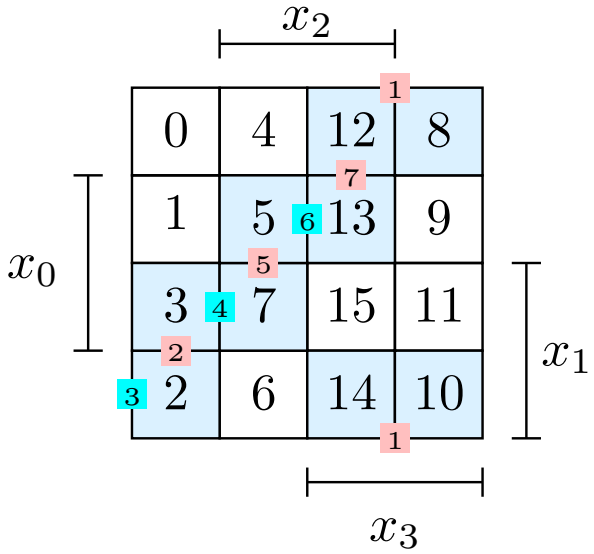
	x_3	x_2	x_1	x_0	
2, 3	0	0	1	*	P_2
2, 10	*	0	1	0	P_3
8, 10	1	0	*	0	√
8, 12	1	*	0	0	√
3, 7	0	*	1	1	P_4
5, 7	0	1	*	1	P_5
5, 13	*	1	0	1	P_6
10, 14	1	*	1	0	√
12, 13	1	1	0	*	P_7
12, 14	1	1	*	0	√

	x_3	x_2	x_1	x_0	
8, 10, 12, 14	1	*	*	0	P_1

- Die Konjunktionen sind in den quineschen Tabellen nach der Anzahl der Einsen im zugehörigen Bitvektor geordnet.
- Zusammenfassung nur möglich, wenn sich die Anzahl der Einsen genau um Eins unterscheidet.
- Abhaken der Konjunktionen mit Zusammenfassungsmöglichkeit.

Auswahl der Primterme

- Aufstellen der Abdeckungstabelle.
- Suche einer minimalen Abdeckungsmenge.



	2	3	5	7	8	10	12	13	14
P_1					×	×	×		×
P_2	×	×							
P_3	×					×			
P_4		×		×					
P_5			×	×					
P_6			×					×	
P_7							×	×	

genutzte Primterme ungenutzte Primterme

- Lösungsmenge: $\{P_1, P_2, P_5, P_7\}$:

$$y = \underbrace{x_3\bar{x}_0}_{P_1} \vee \underbrace{\bar{x}_3\bar{x}_2x_1}_{P_2} \vee \underbrace{\overline{x}_3x_2x_0}_{P_5} \vee \underbrace{x_3x_2\bar{x}_1}_{P_7}$$

- Die quineschen Tabellen wachsen im ungünstigen Fall exponentiell mit der Stelligkeit der Funktion.
- Praktische Programme arbeiten bei großen Funktionen mit Heuristiken und mit unvollständigen quineschen Tabellen.

Zusammenfassung des Gesamtabschnitts:

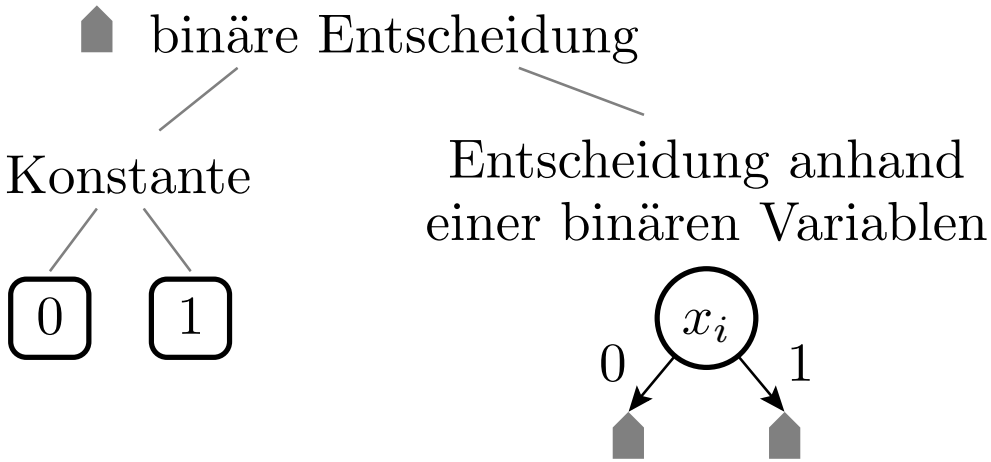
- Die Vereinfachung logischer Ausdrücke erfolgt mit Hilfe von Umformungsregeln.
- Die Optimierungsziele – geringer Schaltungsaufwand, geringe Verzögerung etc. – widersprechen sich zum Teil.
- Die beiden klassischen systematischen Optimierungsverfahren – die Optimierung mit KV-Diagrammen und das Verfahren von Quine und McCluskey – basieren auf der Vereinfachung von Konjunktionsmengen.

ROBDD

Binäre Entscheidungsdiagramme

Logische Funktionen wurden bisher als Tabellen, Ausdrücke, Schaltungen oder Programme dargestellt. Eine alternative Darstellungform sind binäre Entscheidungsdiagramme (BDD, **B**inary **D**ecision **D**iagram).

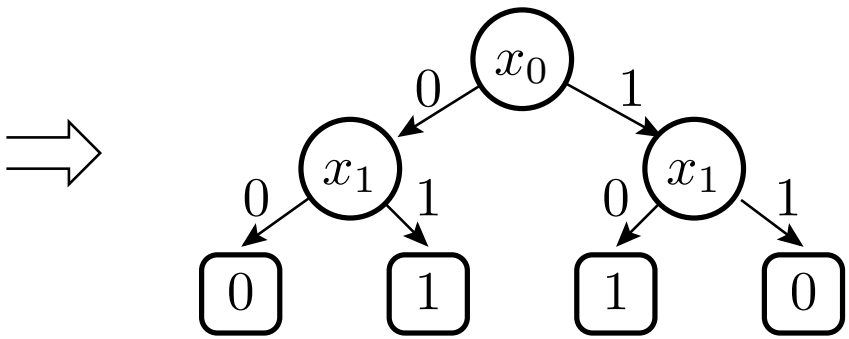
Eine binäre Entscheidung ist eine Konstante oder eine Entscheidung zwischen binären Entscheidungen (Rekursive Definition):



Wertetabelle als binäres Entscheidungsdiagramm

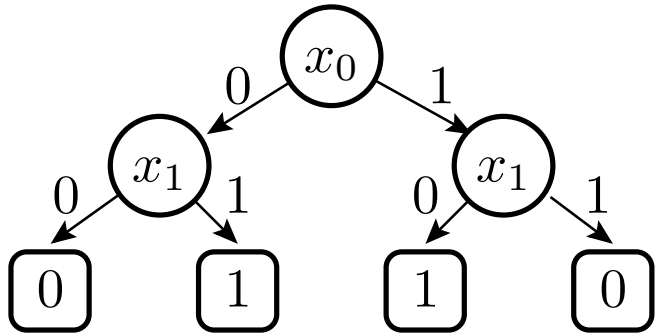
Konstruktion eines Entscheidungsdiagramms mit Blättern für alle Konjunktionen und Zuordnung der Tabellenwerte zu den Blättern.

x_1	x_0	y
0	0	0
0	1	1
1	0	1
1	1	0



Binäres Entscheidungsdiagramm als Programm

```
if x0='0' then
 if x1='0' then
  y <= '0';
 else
  y <= '1';
 end if;
else
 if x1='0' then
  y <= '1';
 else
  y <= '0';
 end if;
end if;
```



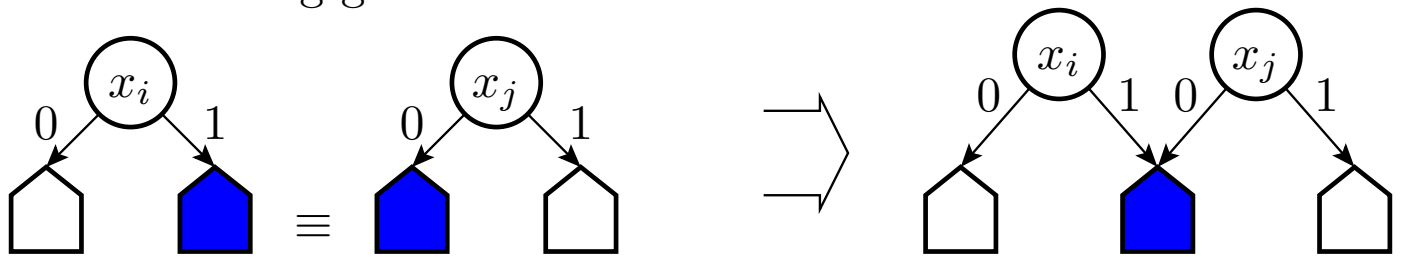
Als Programm, z.B. für die Simulation, besteht ein binäres Entscheidungsdiagramm aus verschachtelten If-Else-Anweisungen.

Vereinfachungsregeln

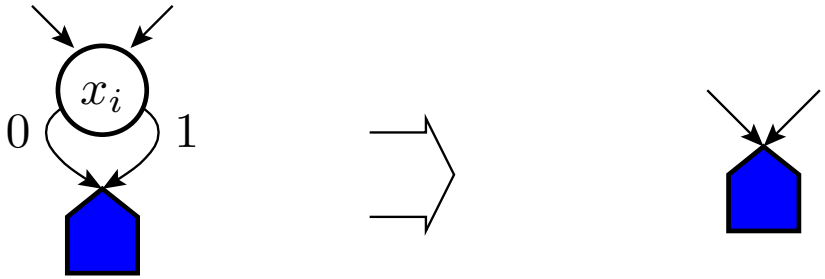
Im Gegensatz zu Ausdrücken gibt es nur zwei Vereinfachungsregeln:

- Verschmelzung gleicher Teilgraphen und
- Eliminierung von Knoten mit zwei gleichen Nachfolgern.

Verschmelzung gleicher Teilbäume



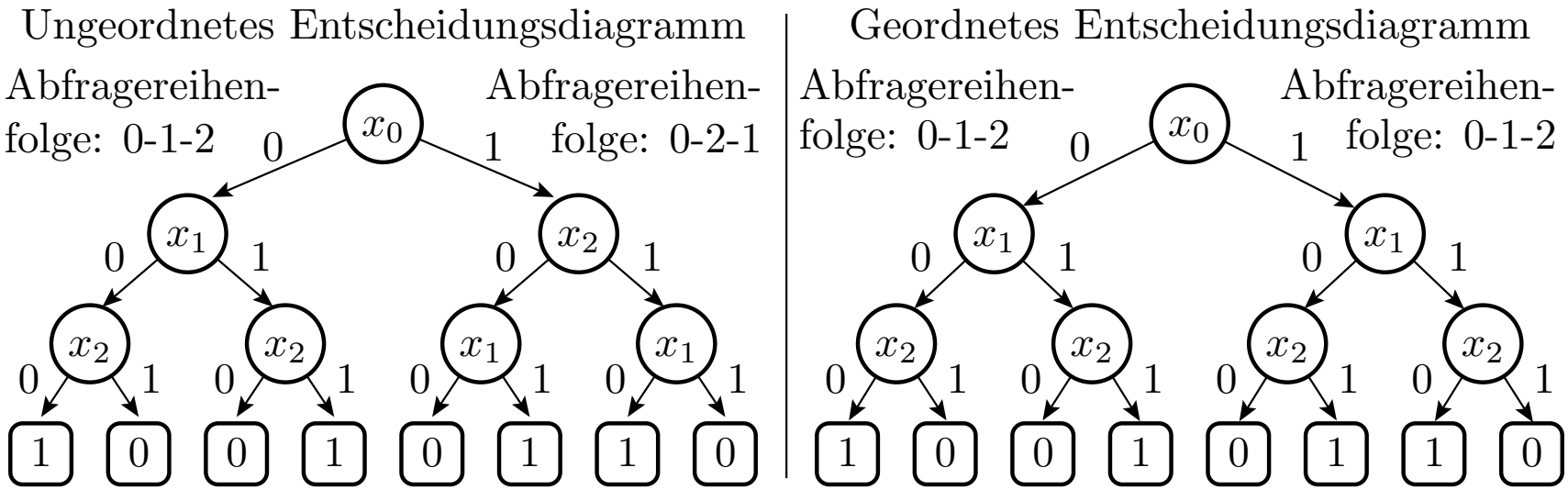
Knoten-
elimination



Das vereinfacht die Automatisierung der Minimierung.

Geordnetes binäres Entscheidungsdiagramm

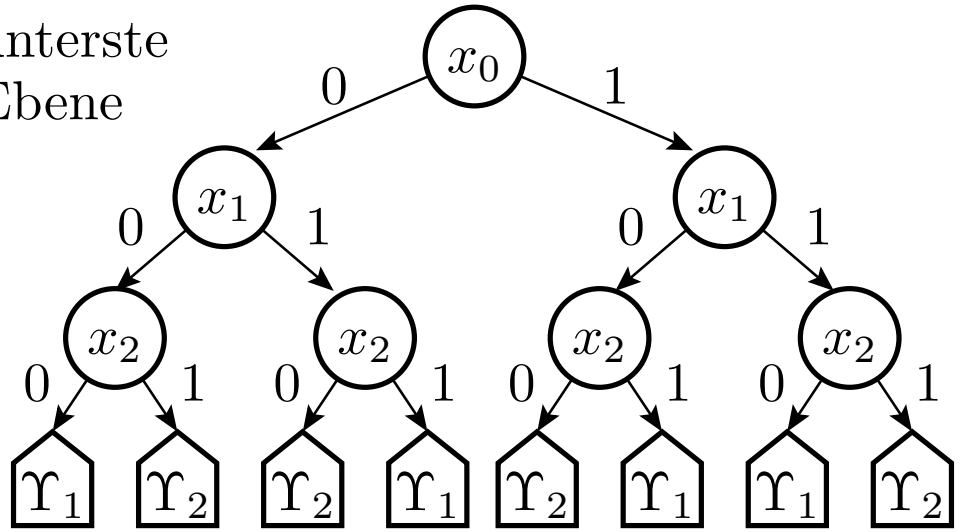
Offensichtliche Voraussetzung für das Verschmelzen ist dieselbe Abfragereihenfolge auf allen Entscheidungswegen. Ein BDD mit geordneter Entscheidungsreihenfolge wird als OBDD (**O**rdered **B**inary **D**ecision **D**iagram) bezeichnet.



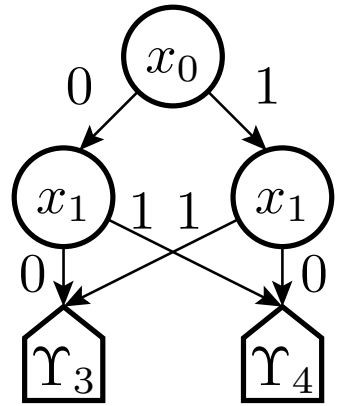
Bezeichnung für ein vereinfachtes binäres Entscheidungsdiagramm ist ROBDD (**R**educed **O**rdered **B**inary **D**ecision **D**iagram).

Vereinfachung am Beispiel

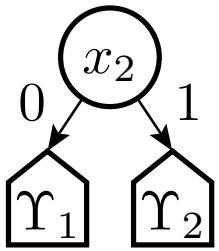
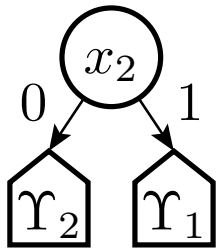
unterste Ebene



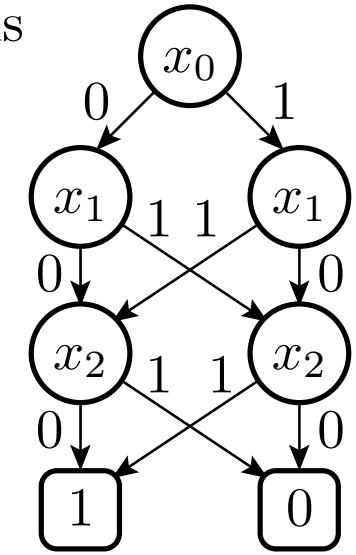
nächste Ebene



unterschiedliche Teilgraphen:

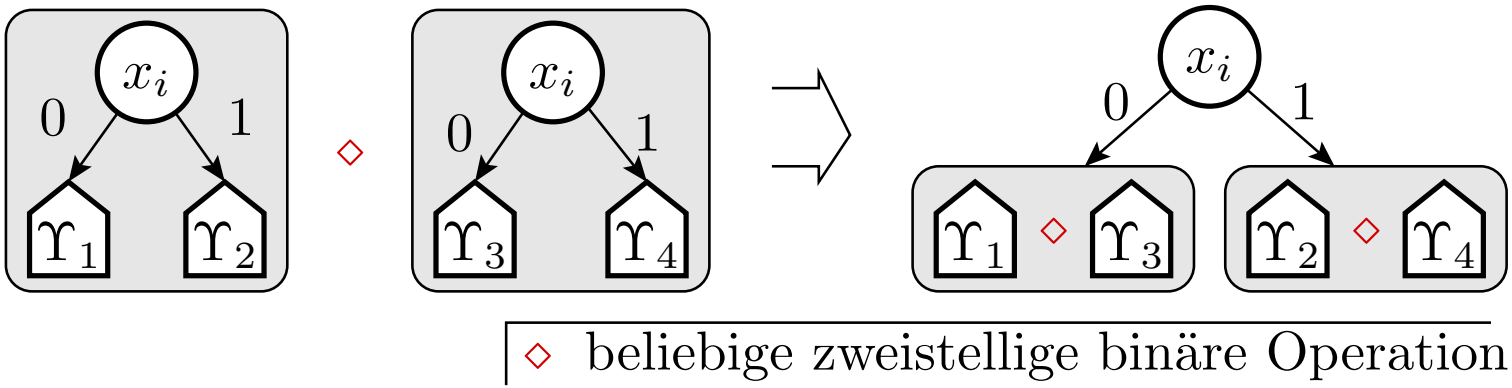
Υ_1 :	Υ_2 :	Υ_3 :	Υ_4 :
1	0		

Ergebnis



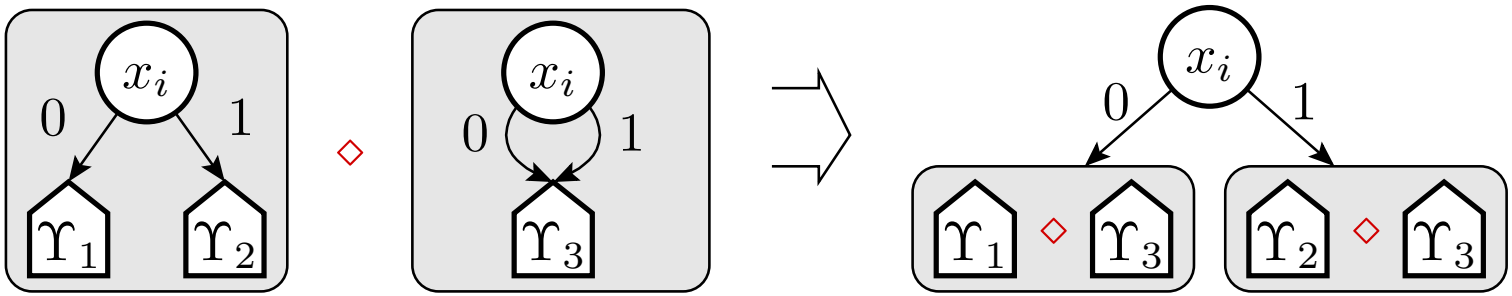
Zweistellige Operationen mit ROBDD

Für zwei beliebige Teilbäume, bei denen im obersten Knoten dieselbe Variable ausgewertet wird, verschiebt sich die Operation eine Entscheidungsebene tiefer.

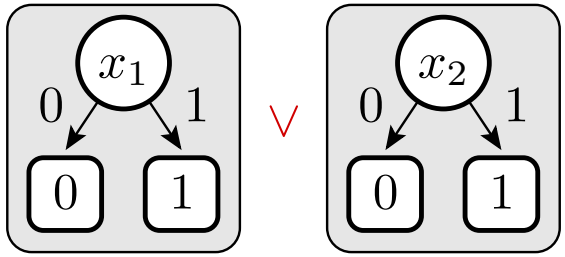


In einem ROBDD erfolgen auf allen Wegen die Entscheidungen in derselben Reihenfolge.

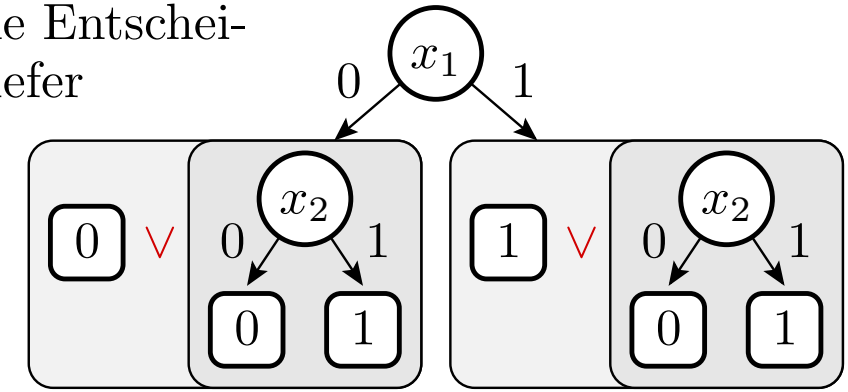
Fehlende Entscheidungsknoten sind wie Entscheidungsknoten mit gleichen Nachfolgern zu behandeln:



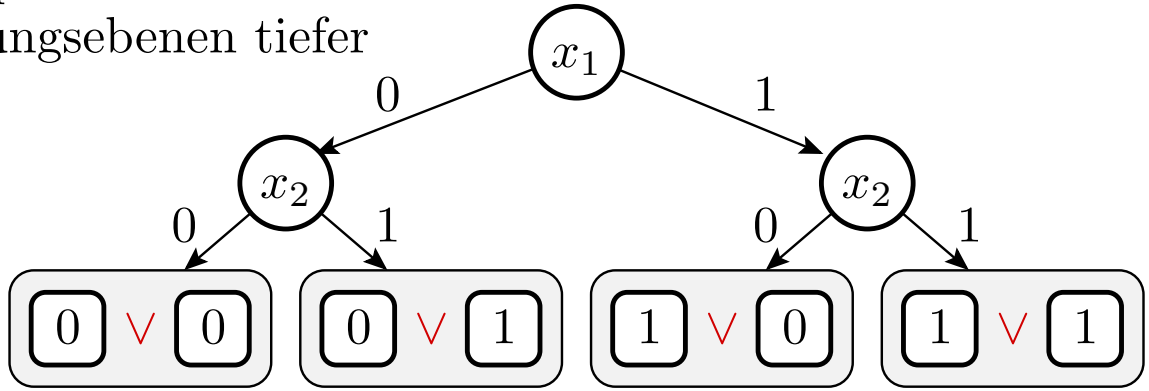
ODER-Verknüpfung
$x_1 \vee x_2$



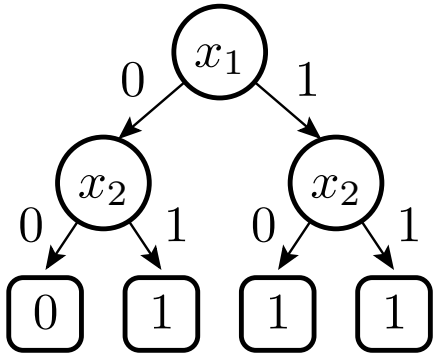
Operation eine Entscheidungsebene tiefer



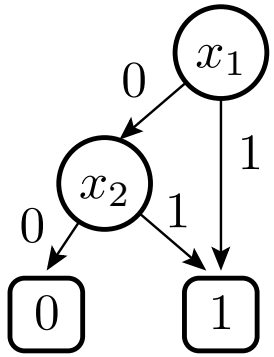
Operation zwei Entscheidungsebenen tiefer



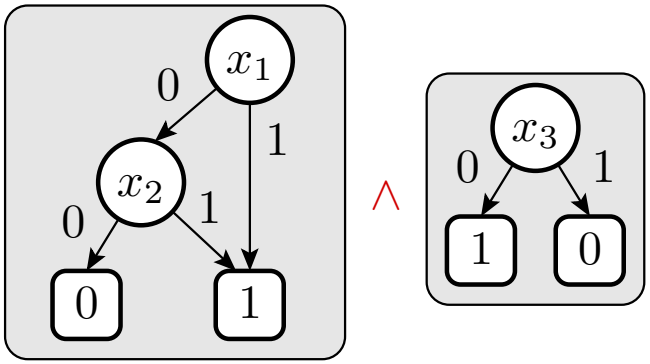
Ausführung der Operation auf der untersten Ebene



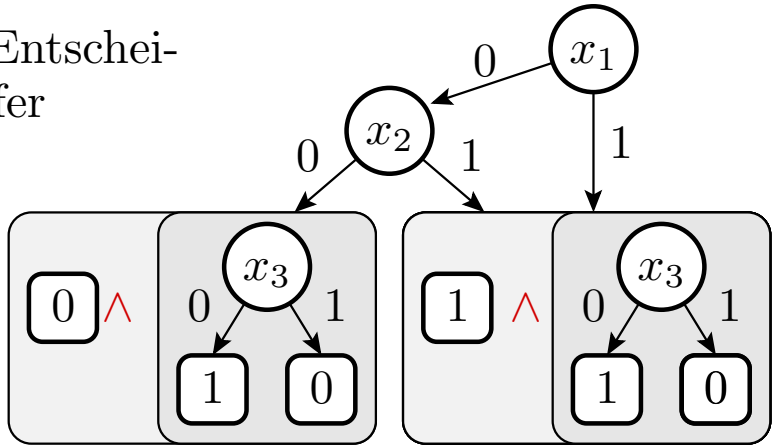
Vereinfachung



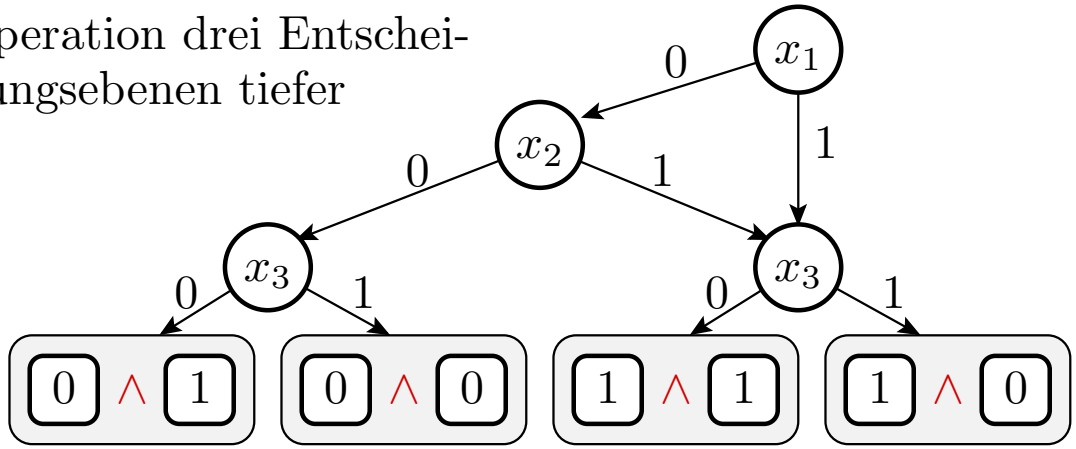
Verknüpfung
$(x_1 \vee x_2) \wedge \bar{x}_3$



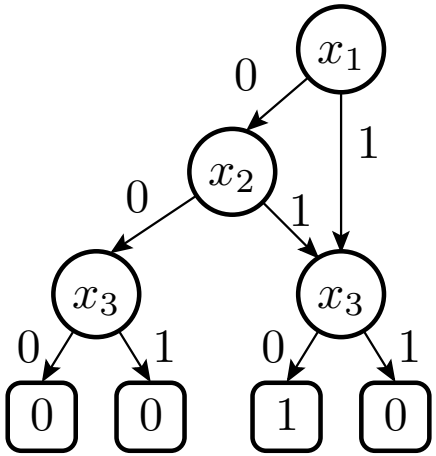
Operation zwei Entscheidungsebenen tiefer



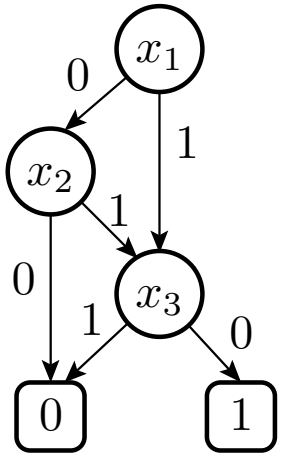
Operation drei Entscheidungsebenen tiefer



Ausführung der Operation auf der untersten Ebene

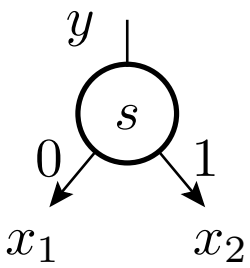


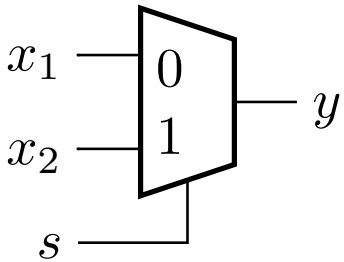
Vereinfachung

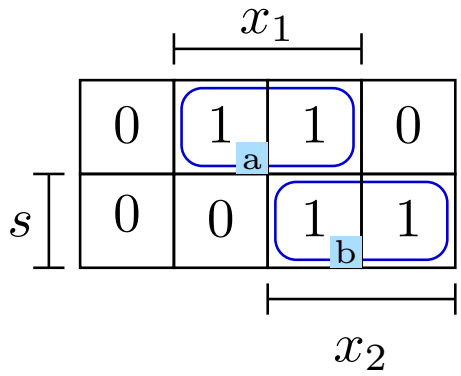


Entscheidungsknoten als Signalflussumschalter

- Entscheidungsknoten ⇒ binärer Umschalter im Datenfluss (Multiplexer); 3-stellige logische Funktion







a: $x_1\bar{s}$
b: $x_2 s$
$y = x_1\bar{s} \vee x_2 s$

- minimiert mit einem KV-Diagramm:

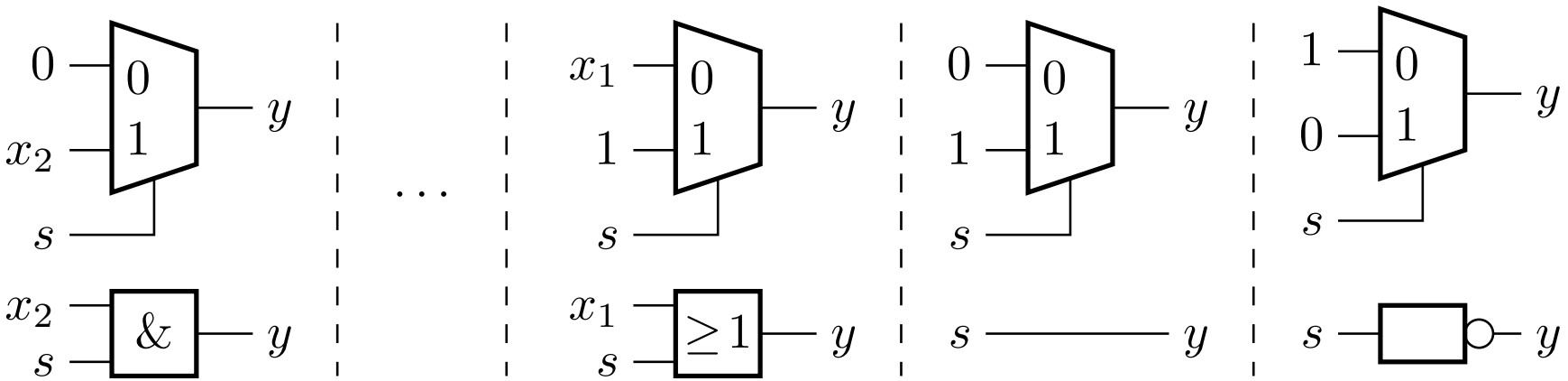
$$y = x_1\bar{s} \vee x_2 s$$

- eine konstante Eingabe $\Rightarrow$ 2-stellige logische Funktion

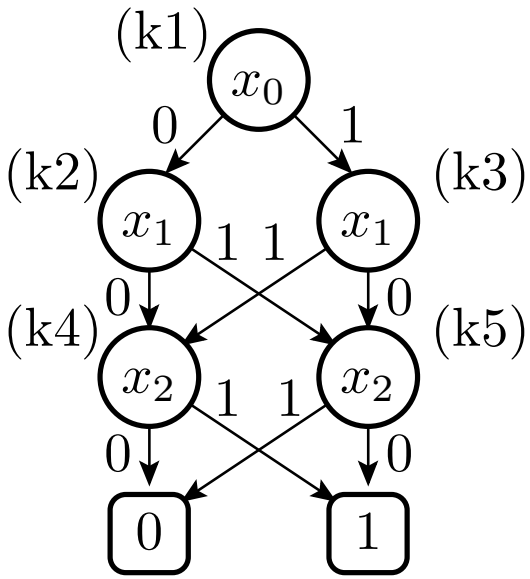
$$
\begin{aligned}
x_1 = 0: &\quad y = (0 \wedge \bar{s}) \vee x_2 s &&= x_2 s \\
x_1 = 1: &\quad y = (1 \wedge \bar{s}) \vee x_2 s &&= \bar{s} \vee x_2 s = \bar{s} \vee x_2 \\
x_2 = 0: &\quad y = x_1 \bar{s} \vee (0 \wedge s) &&= x_1 \bar{s} \\
x_2 = 1: &\quad y = x_1 \bar{s} \vee (1 \wedge s) &&= x_1 \bar{s} \vee s = x_1 \vee s
\end{aligned}
$$

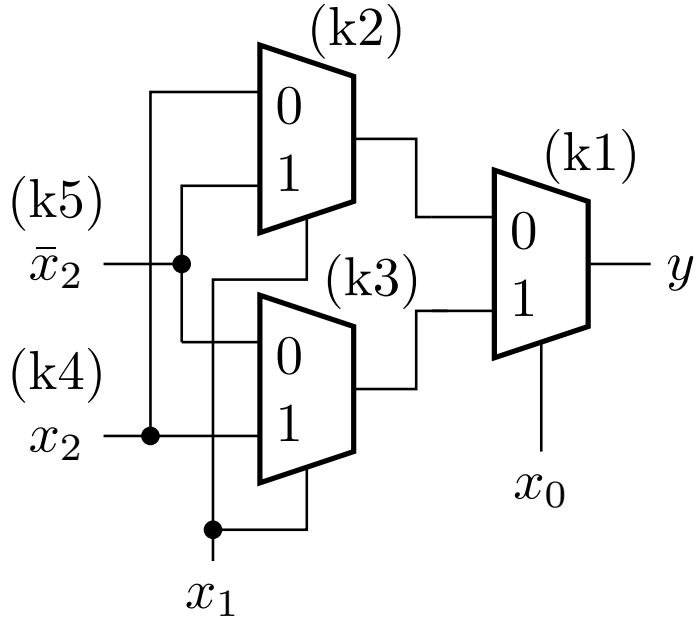
- zwei konstante Eingaben $\Rightarrow$ 1-stellige logische Funktion

$$
\begin{aligned}
x_1 = 0;\ x_2 = 1: &\quad y = (0 \wedge \bar{s}) \vee (1 \wedge s) &&= s \\
x_1 = 1;\ x_2 = 0: &\quad y = (1 \wedge \bar{s}) \vee (0 \wedge s) &&= \bar{s}
\end{aligned}
$$



Entscheidungsdiagramm als Signalflussplan

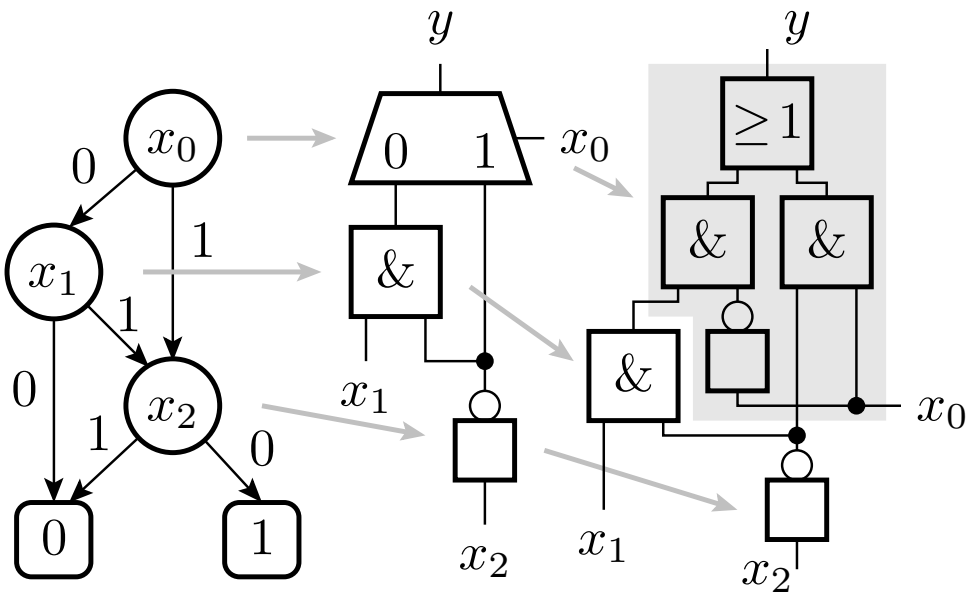




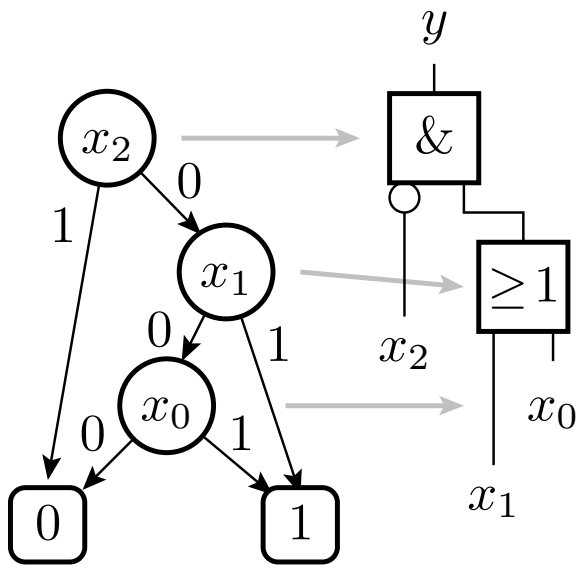
Abfragereihenfolge und Schaltungsaufwand

Zielfunktion: $y = (x_1 \vee x_0) \wedge \bar{x}_2$

Abfragereihenfolge: x_0-x_1-x_2



Abfragereihenfolge: x_2-x_1-x_0



- Abfragereihenfolge $x_0 - x_1 - x_2$: 1 Multiplexer, 1 UND, 1 Inverter
- Abfragereihenfolge $x_2 - x_1 - x_0$: 1 ODER, 1 UND, 1 Inverter

- Ein ROBDD mit einer vorgegebenen Abfragereihenfolge ist eine eindeutige Darstellung für eine logische Funktion.
- Eine n-stellige logische Funktion lässt sich (nur) durch $(n-1)!$ verschiedene ROBDDs darstellen[8].
- Lösungsraum nicht auf UND-ODER-/ODER-UND-Struktur beschränkt.
- Optimierung durch Variation der Abfragereihenfolge. Oft gute Optimierungsergebnisse.
- Ausnutzung der Eindeutigkeit der Darstellung für den Test auf Gleichheit von logischen Funktionen, z.B. einer Ist- und einer Soll-Funktion, ohne die Schaltungen mit allen Eingabevariationen simulieren zu müssen.

[8] Mit Ausdrücken ist die Anzahl der Darstellungsmöglichkeiten unbegrenzt.