Design of Digital Circuits (S4)

Chapter 2, Part 1

Synthesis and logic optimization

Section 2.1 Register transfer synthesis

to 2.3 Binary decision diagram

Prof. G. Kemnitz

Institute of Informatics, Technical University of Clausthal

May 14, 2012

RT synthesis
1.1 Register
1.2 Combinational circuits
1.3 Processing + sampling
1.4 Latches
1.5 Constraints
1.6 Entwurfsfehler
1.7 Zusammenfassungf
1.8 Aufgaben
Logikoptimierung
2.1 Umformungsregeln
2.2 Optimierungsziele
2.3 Konjunktionsmengen

2.4 KV-Diagramme
2.5 Quine und McCluskey
2.6 Aufgaben

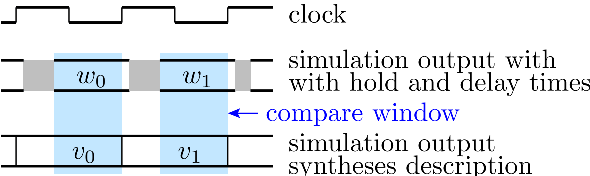
BDD

3.1 Vereinfachungsregeln
3.2 Operationen mit ROBDDs
3.3 ROBDD ⇒ minimierte Schaltung
3.4 Aufgaben

Synthesis

Search for circuit with the same function. Problem timing:

- can be solved only for run time tolerant circuits (optimization, technology mapping etc. change timing)
- no pre-specified delay ⇒ simulation model without delay
- same function ⇒ same output values when the output signals are valid



Synthesis descriptions are simplified simulation models:

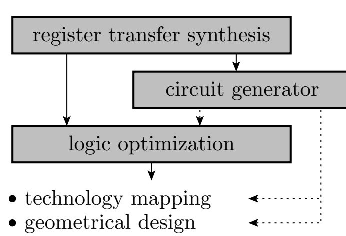
- without delay times (no after statements, no wait statements etc.)
- without check of validity and other plausibility tests (no output of text messages, no pseudo value for »invalid etc.).

After resolving hierarchy it consists of:

- pre-designed circuits, which synthesis transfers unchanged
- combinatorial processes with undelayed signal assignments and
- sampling processes with undelayed signal assignments and without check of setup and input hold conditions.

RT synthesis

RT synthesis as the first synthesis step



works similar as a compiler, but extraction of a signal instead of a control flow

- resolving hierarchy⇒ circuit structure out of predesigned subcircuit an processes
- mapping the calculation flow within the process by a signal flow of technology independent basic circuitry or parametrized functional blocks

---

- Circuit generators: produce optimized circuit descriptions from a parametrized functional block; local optimization, optional incl. technology mapping and geometrical design.
- Logic optimization, technology mapping etc. later

Mapping control flow ⇒ signal graph

is already without timing an ill posed problem:

- for most imperative functional descriptions no circuit exists with the same function
- multiple ways to describe the same circuit
- small changes in description allows new completely different interpretations

Twist the objective:

- How the description must look, so that the synthesis creates a correct circuit?

How registers, combinatorial circuits, etc. have to be described, so that the synthesis recognize them.

Register

Description and extraction of registers

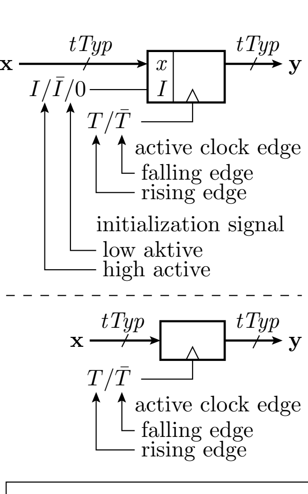
Simulation model (all ready simplified):

```
process(T)
 begin
  if RISING_EDGE(T) then
   if x'LAST_EVENT>ts then --- check setup condition
    y <= invalid after thr, x after tdr;
   else
    y <= invalid after thr;
   end if;
  end if;
end process;
```

further simplifications for synthesis:

- no means to describe timing, validity, ...
- no check of setup and input hold conditions
- no text output (warnings, error messages).

```
--- Register template with initialisation
process(I, T)
begin
 if I='1' then
--- or if I='0' then
  y <= initialization_value;
 elsif RISING_EDGE(T) then
--- or elsif FALLING_EDGE(T) then
  y <= x;
 end if;
end process;

--- without initialisation
process(T)
begin
 if RISING_EDGE(T) then
--- orr elsif FALLING_EDGE(T) then
  y <= x;
 end if;
end process;
```



tTyp	in binary signals convertable datatype (bit, bitvector, number, ...)

Sample process ⇔ register

- given are three sampling processes

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

no initialization; sampling with the rising edge of a

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

Initialization with $c = 1$; sampling with the falling edge of b

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

Initialization with $c = 1$; sampling with the raining edge of a

```
process(a, c)
begin
 if c='1' then
  M <= "010";
 elsif RISING_EDGE(a)
  then
  M <= K;
 end if;
end process;
```

- inputs, outputs and parameters of the described registers

data input signal	d	K(0)	K(1)	K	K
data output signal	K(0)	K(1)	K(2)	L	M
bit width	1	1	1	3	3
clock signal	a ↑	a ↑	a ↑	b ↓	a ↑
initialization signal	–	–	–	c (H)	c (L)
initialization value	–	–	–	000	010

↑ – rising edge; ↓ – falling edge; H – high active; L – low active

Register transfer synthesis

- extraction of the terminals and parameters of all described registers
- Forwarding the data to circuit generators; generation of the registers

The designer

- must stick to the description templates
- may summarize all registers withe the same initialization and sampling conditions in one sampling process

---

Advantage of less processes

- less calculation effort for simulation
- less time critical transitions between different clocked circuit parts

Combinational circuits

Synthesis description of a combinational circuits

imperative behavioral model of a combinatorial circuit:

- if one of the input signal switches, new calculation of the output; no memory

Description template:

- process with all input signals in the sensitivity list
- store interim results in the calculation in variables (not signals!)
- no further processing of interim results from previous wake-up dates
- in each path of control a value must be assigned to each output signal.

---

additional simplifications for synthesis: no delay, no check for validity

If synthesis finds a process which complies with all rules:

- extract of the control flow
- substitute operators by logic gates
- produce case distinctions by multiplexers.

Circuits of basic gates

signal a, b, s, y: *tBit*;		y <= **not** a
		a
			if s=1 **then**
			y <= a;
			else y <= b;
			end if;

y <= a **and** b a, b, &, y	y <= a **or** b a, b, ≥1, y	y <= a **xor** b a, b, =1, y	*a*, *b*, 1, 0, *y*, *s*
y <= a **nand** b a, b, &, y	y <= a **nor** b a, b, ≥1, y	y <= a **xnor** b a, b, =1, y	

(tBit: BIT, BOOLEAN or STD_LOGIC; BOOLEAN: TRUE ↦ 1, FALSE ↦ 0; using STD_LOGIC the pseudo values for ≫invalid≪ etc. remains unused)

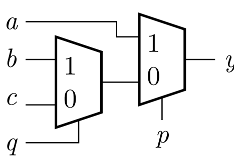
Description & extraction of gates

```
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;
```

From case distinctions to multiplexers

- if control flow branches by »If« or »Case« instructions output or interim values has to be assign in each branch
- If-Elsif becomes a multiplexer chain

```
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;
```



- instead of »Else« it is also possible to overwrite a default value (even for signals):

```
y<=c;
if p='1' then y<=a; end if;
```

Parametrized functional blocks

- operators with bit vector operands
  $\Rightarrow$ functional blocks with bit sizes as parameters
- arithmetic operation and compare ($>$, $\geq$ etc.)
  $\Rightarrow$ additional parameter: number representation
- select statements with more than two cases
  $\Rightarrow$ additional parameters: selection values for each data input

---

register transfer synthesis:

- extraction of functional blocks and its parameters

circuit generators:

- building the circuits out of basic gates; optional incl. technology mapping, ...

Select statement ⇒ parametrized 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;
```


- parameters: data bit size and select values
- last selection value must be »others«[1]

[1] On one hand selection values such as »0X«, »XU« are forbidden in synthesis descriptions, on the other hand a case-statement has to take into account the whole range of selection values.

Bit comparators

x_1 x_0	$x_0 = 0$	$x_0 \neq 0$	$x_0 = 1$	$x_0 \neq 1$	$x_1 = x_0$	$x_1 \neq x_0$
0 0	TRUE (1)	FALSE (0)	FALSE (0)	TRUE (1)	TRUE (1)	FALSE (0)
0 1	FALSE (0)	TRUE (1)	TRUE (1)	FALSE (0)	FALSE (0)	TRUE (1)
1 0					FALSE (0)	TRUE (1)
1 1					TRUE (1)	FALSE (0)
	x_0 –[ ]o–	x_0 —	x_0 —	x_0 –[ ]o–	x_0, x_1 –[==]–	x_0, x_1 –[=1]–

- branch values in if-statements are generally produced by comparisons
- in synthesis the Boolean values »false« and »true« are mapped to the value range $\{0, 1\}$

Arithmetical operations (+, −, *) and >, ≥ etc.

- only defined for number and bit vector types
- the circuit to be generated depends on the number representation (natural, whole numbers, floating point, ...)

types of bit vectors for number representation in this lecture:

```
--- for unsignd whole numbers
tUnsigned is array (NATURAL range <>) of STD_LOGIC;
--- for signd whole numbers
tSigned is array (NATURAL range <>) of STD_LOGIC;
```

defined together with the arithmetical, logical and compare operations in the package »Tuc.Numeric_Synth«

Data flow symbols for compare operators

signal a, b: t[Un]signed(n-1 **downto** 0);

parameters:

- bit size n
- data type (signed or unsigned whole numbers) [2]

[2]Other number types will not be used for synthesis in this lecture.

Addition, subtraction and multiplication

```
signal a: T[Un]signed(n-1 downto 0);
signal b: T[Un]signed(m-1 downto 0);
constant c: T[Un]signed(m-1 downto 0);
```

adder | subtractor | multiplier

adder: **a** (n), **b** (m) → + → $\max(n,m)$ → **a+b**; **a** (n) → +**c** → $\max(n,m)$ → **a+c**

subtractor: a (n), b (m) → **a**, **b**, **a−b** → $\max(n,m)$ → **a−b**; **a** (n) → −**c** → $\max(n,m)$ → **a−c**

multiplier: a (n), b (m) → * → $n+m$ → **a*b**; **a** (n) → * **c** → $n+m$ → **a*c**

- sum and difference have the bit size of the larges operand
- the bit size of a product is the sum of the bit sizes of the operands
- operations not listed here (division, power) are generally realized by an operation sequence

Special calculators with arithmetic operations

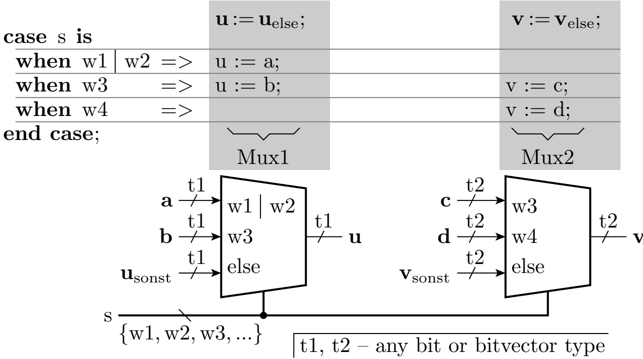
```
signal a, b: tUnsigned(2 downto 0);
signal y: tUnsigned(7 downto 0);
...
process(a, b)
 variable v: tUnsigned(6 downto 0);
begin
 A1: v:= a * "1010";
 A2: y<= ('0'&v) + b;
end process;
```


- size of the product: $3 \times 4 \Rightarrow 7$ bit
- concatenation of a leading one extends first summand to 8 bits
- size of the sum: $\max(3, 8) \Rightarrow 8$ bit

Circuit with adder, comparator, multiplexer and gates

```
signal a, b, y: tUnsigned(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;
```

One select statement, multiple multiplexers



- assignment of a default value before the select statement
- overwrite the default value for select values assigning a different value

Describing a truth table by a case statement

```
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
	else			0

Processing + sampling

Combinatorial circuit and sampling register

Combining combinatorial circuitry with the subsequent register in a sampling process reduces simulation effort; changes in description starting from a pure combinatorial process:

- clock instead all input signal in the sensitivity list
- aligning signal assignments to the active clock edge
- optional initialization (in addition initialization signal in the sensitivity list etc.)

```
process(I, T)
 if I=active then
  y <= initial_value;
 elsif active_clock_edge_of_T then
  output_calculation_of_the_processing_function;
  y <= assigning_the_processing_result;
 end if;
end process;
```

Combinatorial circuit with output register

```
signal a, b, y: STD_LOGIC_VECTOR(4 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;
```

Output register with conditional sampling

```
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;
```


registers may store values for multiple clocks; describing conditional sampling:

- multiplexer between input and actual value
- register extension by an enable input

Sampled signal may processed further

```
signal y: tUnsigned(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 combinatorial circuits a feedback of the form ≫y<=y+"1"≪ is not allowed

Up/down counter with enable input

```
variable tmp: tUnsigned(3 downto 0);
...
if V='1' then
  tmp:=y+"1";
else
  tmp:=y-"1";
 end if;
 if V='1' or R='1'  then
  y<=tmp;
 end if;
```


- different descriptions of the same function may synthesised to different circuits
- way for optimization

Variables as data storage in sampling processes

```
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;
```


- variables, which are read before assignment in the control flow, store data for one clock period; behavior of a register
- variables are only readable and writable within a process / can can not be used for terminal signals

Behavior of conditional variable assignments

```
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;
```


- A1: ≫v≪ is assigned at least one clock earlier (register output)
- A3: ≫v≪ may be assigne in the same clock (register input)

Variables that store data may represent in a synthesized circuit different circuit points.

The description of registers by variables

- is for that error-prone
- should be used deliberately.

Latches

Latches, stage triggered memory element

```
signal E: STD_LOGIC; signal x, y: tTyp;
...
process(x, E)
 variable tE:DELAY_LENGTH;
begin
 if E='1' then
  y <= invalid after th, x after td;
 end if;
 --- check setup condition
 if RISING_EDGE(E) then
  tE:=NOW;
 elsif Falling_EDGE(E) and (NOW-tE<ts
    or x'LAST_EVENT<ts) then
   y <= invalid;
 end if;
end process;
```


t_d	Verzögerungszeit
t_h	Haltezeit
t_s	Vorhaltezeit
E	Freigabeeingang
$tTyp$	Bit- oder Bitvektortyp

- einfachere Schaltung als Register; genutzt zur Aufwandsminimierung
- das Freigabesignal ist zeit- und glitch-empfindlich

Die Synthesebeschreibungsschablone ist das stark vereinfachte Simulationsmodell:

- ohne Zeitangaben
- ohne Kontrolle der Vor- und Nachhaltebedingungen
- ohne Berechnung der Gültigkeitsfenster

```
process(x, E)
begin
 if E='1' then
  y <= x;
 end if;
end process;
```

Auch wenn die Kontrollen fehlen, müssen die Vor- und Nachhaltebedingungen erfüllt sein.

Die Tücken einer Latch-Schaltung am Beispiel

```
signal x1, x2, y:
 STD_LOGIC_VECTOR(n-1 downto 0);
...
process(x1, x2)
begin
 if x1=x2 then
  y <= x1;
 end if;
end process;
```


F1: möglich Invalidierung des gespeicherten Wertes

F2: möglicher Übernahmefehler bei Übereinstimmung

Blockspeicher mit Latches

```
signal E: STD_LOGIC;
signal a: STD_LOGIC_VECTOR(1 downto 0);
signal x, s, q, y0, y1, y2, y3, s:
  STD_LOGIC_VECTOR(3 downto 0);
...
Dec:process(a)
begin
 case a is
   when "00"  => s <= "0001";
   when "01"  => s <= "0010";
   when "10"  => s <= "0100";
   when others => s <= "1000";
  end case;
end process;
```

```
Und:process(s, E)
begin
 if E='1' then q <= s;
 else q <= "0000";
end if;
end process;
Latch0:process(x, q(0))
begin
 if q(0)='1' then y0<=x; end if;
end process;
...
```

Latch-Schaltungen sind nur mit einer bestimmten Struktur und Ansteuerung laufzeitrobust

- UND-Verknüpfung mit E unmittelbar vor den Freigabeeingängen der Latches
- E muss inaktiv sein, wenn die Signale s_i invalid sind
- laufzeitkritische Teile als vorentworfene Schaltungen einbinden

Latch zur Variablennachbildung in Abtastprozessen

```
signal T, a, b, c, y: STD_LOGIC;
...
process(T)
 variable v: STD_LOGIC;
begin
 if RISING_EDGE(T) then
  if b='1' then
   v:= c;
  end if;
  y <= v xor a;
 end if;
end process;
```

- bei einem Latch ist der Übernahmewert sofort, nicht erst im Folgetakt am Ausgang verfügbar; erspart Multiplexer
- b muss mit Latch ein glitch-freies Abtastsignal sein

Constraints

Constraints

Zusatzinformationen für die Synthese;
Festlegungen/Empfehlungen für

- die Struktur und die Technologieabbildung (z.B. ⋙keep⋘, um die Wegoptimierung bestimmter Signale und Teilschaltungen zu verbieten)
- die Platzierung (z.B. Zuordnung zwischen Anschlusssignalen und Schaltkreispins)
- maximale Verzögerungszeiten (Taktfrequenz oder -periode, Eingabe-Register-Verzögerung etc.)

Für Praktikum (ise/Versuchsboard mit Xilinx-FPGA):
Beschreibung der Pin-Zuordnung und der Taktfrequenz in der ucf-Datei (user constraints file):

```
<loc> ..
```

Entwurfsfehler

Entwurfsfehler und Fehlervermeidung

Synthesebeschreibungen haben ihre typischen Entwurfsfehler, darunter auch einige die nur das Zeitverhalten und/oder die Zuverlässigkeit der synthetisierten Schaltung beeinträchtigen und dadurch schwer zu finden sind.

Speicherverhalten in kombinatorischen Prozessen

```
signal x1, x2, sz, y, yF1, yF2, yF3: STD_LOGIC;
```

korrekt	F1

`process (x1, x2);`
`  variable z: STD_LOGIC;`
`begin`
`  z := x1 and x2;`
`  y <= not z;`
`end process;`
	`process (x1);`
	`  variable z: STD_LOGIC;`
	`begin`
	`  z := x1 and x2;`
	`  yF1 <= not zb;`
	`end process;`

F2	**F3**

`process (x1, x2);`
`begin`
`  sz <= x1 and x2;`
`  yF2 <= not sz;`
`end process;`
	`process(x1, x2);`
	`  variable z: STD_LOGIC;`
	`begin`
	`  yF3 <= not z;`
	`  z := x1 and x2;`
	`end process;` warte

x_1, x_2 → & → z → ○ → y

Signale: x_1, x_2, y, y_{F1}, y_{F2}, y_{F3} (jeweils 1 / 0)

— Fehlverhalten
— Soll-Verhalten

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;
     --- Korrektur:  y <= x3 statt null
  end case;
 end process;
```


- auch wenn ein Auswahlwert nicht auftreten kann, ist in einer Multiplexerbeschreibung ein Wert zuzuweisen

Fehlersymptome:

- nicht synthetisierbar oder unerwarteter Latch-Einbau

Fehlervermeidung

- Verwendung der Kurzform »nebenläufige Signalzuweisung«, im Beispiel aus UND und Inverter:

```
y <= not (x1 and x2);
```

- später für beliebig komplexe kombinatorische Schaltungen:

```
signal x: tEingabe;
signal y: tAusgabe;
function f(x: tEingabe) return tAusgabe;
...
y <= invalid after th, f(x), after td;
```

- Synthesebeschreibung ohne grau unterlegte Teile
- nebenläufige Signalzuweisungen und Funktionen können kein Speicherverhalten beschreiben / keinen der skizzierten Fehler enthalten

Bösartige Timing-Probleme

Symptome:

- zulässige Taktfrequenz laut Synthsereport zu niedrig
- messbare Zeitprobleme an den Anschlusssignalen
- Änderung der Beschreibung zeigt keine Wirkung
- Fehlerwirkung nicht mit Simulation nachstellbar / nicht reproduzierbar

Ursache sind in der Regel fehlende oder falsche Zeit-Constraints

- Constraint-Doku lesen und Constraints richtig beschreiben

Gutartige Timing-Probleme

Symptome:

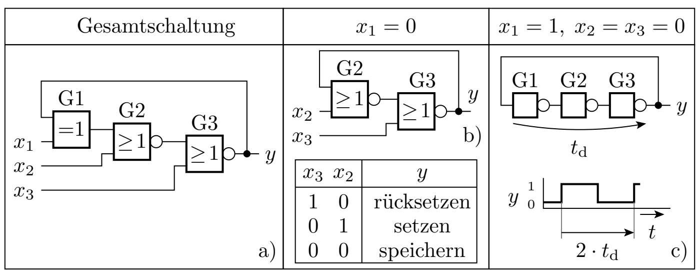
- Zusatzverzögerungen um ganze Taktperioden
- Fehlerwirkung mit Simulation nachstellbar

typische Ursachen

- Zwischenergebnisse in Abtastprozessen in Signalen statt Variablen weitergereicht
- Variablenwerte vor der Wertzuweisung ausgewertet
- Denkfehler im Algorithmus und in der Ablaufplanung

Nicht synthetisierbar

ungeeignete Funktion, z.B. kein gerichteter Berechnungsfluss



- Speicher- oder Schwingungsverhalten bei der Simulation

ungeeignete Beschreibungsstruktur

- Wenn das simulierte Verhalten ok. ist, ausweichen auf bewährte Beschreibungsschablonen (Doku des Syntheseprogramms lesen)

Zusammenfassungf

Zusammenfassung

Eine Synthesebeschreibung ist ein vereinfachtes Simulationsmodell ohne Verzögerungen und ohne Berechnung der Signalgültigkeit etc.. Zuordnungen:

- Signalzuweisung bei aktiver Taktflanke ⇒ Register
- logische, arithmetische und Vergleichsoperatoren ⇒ Gatter, Rechenwerke und Komparatoren
- Fallunterscheidungen und Auswahlanweisungen ⇒ Multiplexer
- zustandsgesteuerte bedingte Zuweisungen ⇒ Latches, laufzeitempfindlich

Zeit- und Strukturanforderungen werden durch Constraints beschrieben. Bei einer Synthesebeschreibung kommt es nicht nur auf die Funktion, sondern auch auf die Beschränkung auf bewährte Beschreibungsschablonen an.

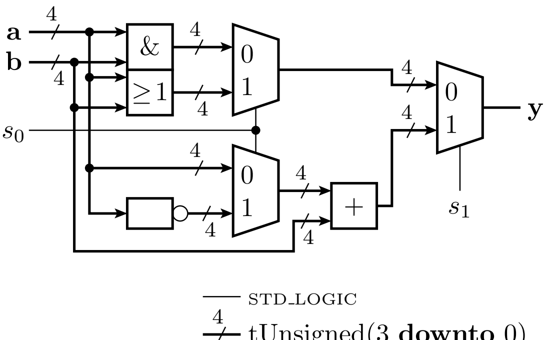
Aufgaben

Aufgabe 2.1: Registerextraktion

```
signal K,L,M,N,P,Q: STD_LOGIC_VECTOR(7 downto 0);
signal c: STD_LOGIC_VECTOR(1 downto 0);

process(c(0))
begin
 if RISING_EDGE(c(0)) then
  L <= K;
 end if;
end process;

process(c(1))
begin
 if RISING_EDGE(c(1)) then
  N<=M; M<=L;
 end if;
end process;

process(c(1))
begin
 if FALLING_EDGE(c(1)) then
  P<=L; Q<=P;
 end if;
end process;
```

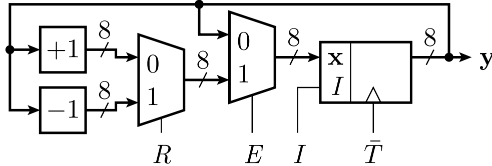
- Anschlusssignale, Übernahmebedingungen etc. aller beschriebenen Register suchen
- Signalflussplan zeichnen

Aufgabe 2.2: Beschreibung als synthesefähiger kombinatorischer Prozess



Aufgabe 2.3: Beschreibung als synthesefähiger Abtastprozess

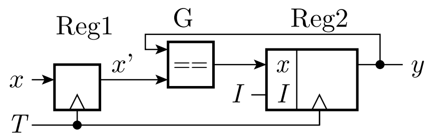
```
signal y: tSigned(7 downto 0);
signal R, E, I, T: STD_LOGIC;
```



- Initialisierungswert »alles null«

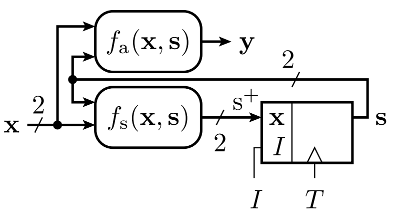
Aufgabe 2.4: Synthesefähige Beschreibung mit möglichst wenig Prozessen

```
signal T, I, x, x_del, y: STD_LOGIC;
```



- Initialwert »0«

Aufgabe 2.5: Synthesefähige Beschreibung des Automaten



d)	$\mathbf{s}^+ = f_s(\mathbf{x},\mathbf{s})$	$\mathbf{y} = f_a(\mathbf{x},\mathbf{s})$
s	x : 00 01 10	00 01 10
00	01 00 11	01 00 11
01	10 01 00	10 01 00
10	11 10 01	11 10 01
11	00 11 10	00 11 10

Aufgabe 2.6: Extraktion des Signalflussplans

```
signal x, tmp, acc, y: STD_LOGIC_VECTOR(3 downto 0);
signal op: STD_LOGIC_VECTOR(1 downto 0);
signal T: STD_LOGIC;
...
process(T)
begin
 if RISING_EDGE(T) then
  case op is
   when "00" => acc <= x;
   when "01" => acc <= acc + tmp;
   when "10" => acc <= acc - tmp;
   when others => null;
  end case;
 end if;
 tmp <= x;
end process;
y <= acc;
```

Logikoptimierung

Definition 1

Eine n-stellige logische Funktion ist eine Abbildung eines n-Bit-Vektors aus freien binären Variablen auf eine abhängige binäre Variable:

$$f: \mathcal{B}^n \to \mathcal{B} \qquad \text{mit } \mathcal{B} = \{0, 1\}$$

bzw.

$$y = f(x_{n-1}, x_{n-2}, \ldots, x_0) \quad \text{mit } y, x_i \in \{0, 1\}$$

(y – abhängige Variable; x_i freie Variablen).

- Jede logische Funktion kann durch praktisch unbegrenzt viele logische Ausdrücke und Schaltungen nachgebildet werden.
- Die Register-Transfer-Synthese extrahiert logische Funktionen in Form von kombinatorischen Teilschaltungsbeschreibungen.

- Anschließende Vereinfachung:
  - Vereinfachung logischer Ausdrücke mit Hilfe der Schaltalgebra (dieser Abschnitt)
  - Darstellung der logischen Ausdrücke zur Vereinfachung als binäre Entscheidungsdiagramme (nächster Abschnitt).

Bereits behandelte Vereinfachungstechniken:

- Konstantenelimination
  beseitigt alle Operationen, bei denen freie Variablen mit konstanten Werten belegt sind; reduziert die Stelligkeit und die Anzahl der Operationen
- Verschmelzung
  fasst gleiche Berechnungsschritte mit gleichen Operanden zusammen und reduziert so die Anzahl der Operationen

Umformungsregeln

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
$\bar{x}_1 \vee \bar{x}_2 = \overline{x_1 \wedge x_2}$ $\bar{x}_1 \wedge \bar{x}_2 = \overline{x_1 \vee x_2}$	de morgansche Regeln

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

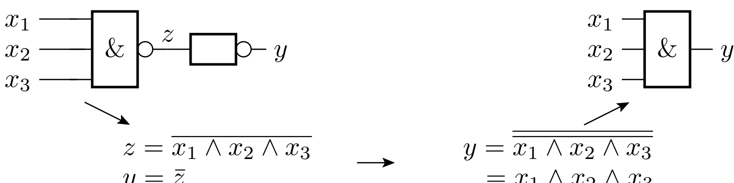
Beweis der Umformungsregeln

- Aufstellen und Vergleich der Wertetabellen.
- Für die de morganschen Regeln gilt z.B.:

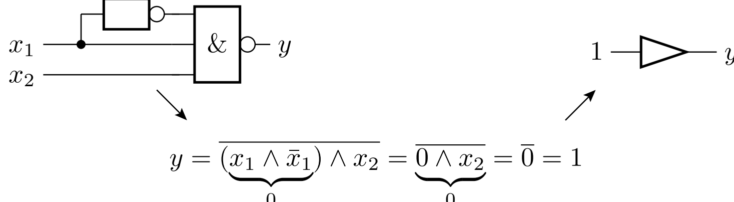
x_1	x_2	$\bar{x}_1 \vee \bar{x}_2$	$\overline{x_1 \wedge x_2}$	$\bar{x}_1 \wedge \bar{x}_2$	$\overline{x_1 \vee x_2}$
0	0	1	1	1	1
0	1	1	1	0	0
1	0	1	1	0	0
1	1	0	0	0	0

Anwendung der Umformungsregeln zur Schaltungsvereinfachung

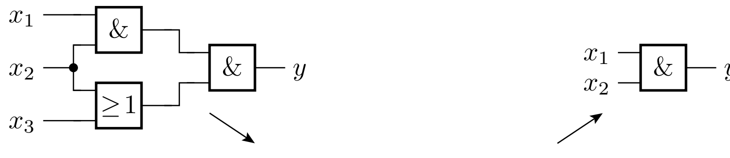
- mehrfache Negationen im Signalfluss heben sich paarweise auf



- doppelte Anwendung der Eliminationsgesetze



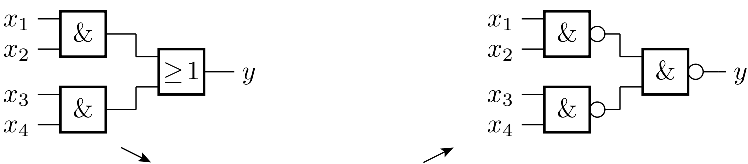
- Anwendung des Absorbtionsgesetzes



gegeben 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: $y = x_1 \wedge x_2$

Suchproblem: in gegebenen Ausdrücken
Anwendungsmöglichkeiten für die Gesetze finden

- Terme nutzen die Grundoperationen UND, ODER und Negation
- technische Gatter haben oft eine invertierte Ausgabe (siehe später »Konstruktion von Logikgattern aus Transistoren«)
- Umformung der UND-ODER-Form in die NAND-NAND-Form

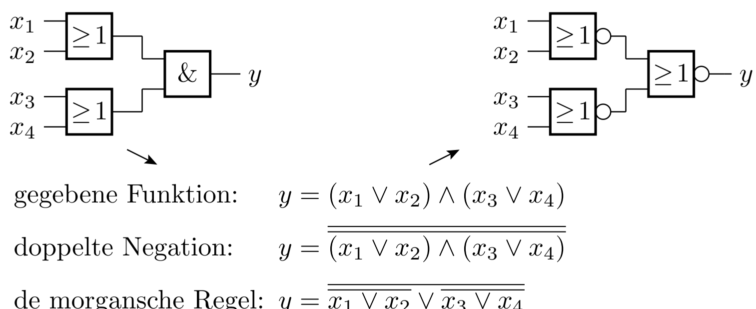


gegebene Funktion: $y = (x_1 \wedge x_2) \vee (x_3 \wedge x_4)$

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

de morgansche Regel: $y = \overline{\overline{x_1 \wedge x_2} \wedge \overline{x_3 \wedge x_4}}$

- Umformung der ODER-UND-Form in die NOR-NOR-Form

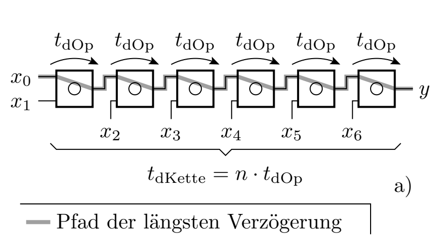


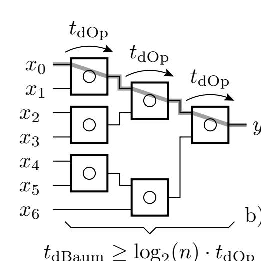
Optimierungsziele

Optimierungsziele

- minimale Gatteranzahl
- minimale Chipfläche
- minimaler Stromverbrauch
- minimale Verzögerung etc.

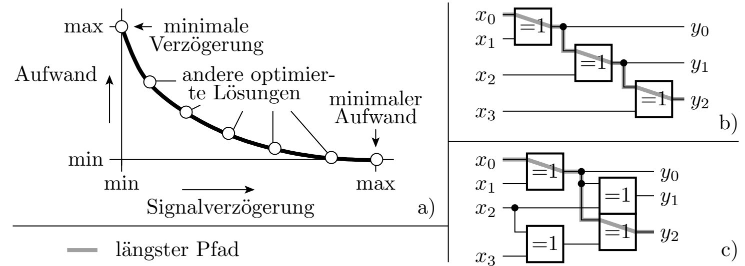
Regel zur Geschwindigkeitsoptimierung: Bäume statt Ketten





◦ assoziative Operation (Zusammenfassungsreihenfolge vertauschbar)

oft unterscheiden sich die Lösungen für minimalen Aufwand und max. Geschwindigkeit etc.



Konjunktionsmengen

Logikminimierung mit Konjunktionsmengen

klassische Verfahren

- KV-Diagramm: graphisches Verfahren
- Quine-McCluskey: gleicher Algorithmus als Suchproblem

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

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

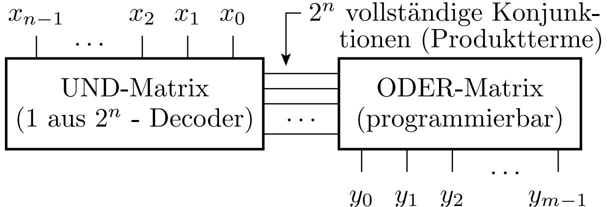
Minterm: Konjunktion, die alle Eingabevariablen entweder in direkter oder in negierter Form enthält.

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 ist.

Die Funktion ist entweder die

- ODER-Verknüpfung der Minterme, für die $y = 1$ ist



- 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	$\bar{x}_2\bar{x}_1 x_0$ (K_{001})	1	1	0	1	$x_2\bar{x}_1 x_0$ (K_{101})	1
0	1	0	$\bar{x}_2 x_1\bar{x}_0$ (K_{010})	0	1	1	0	$x_2 x_1\bar{x}_0$ (K_{110})	1
0	1	1	$\bar{x}_2 x_1 x_0$ (K_{011})	0	1	1	1	$x_2 x_1 x_0$ (K_{111})	0

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

$$y = \bar{x}_2\bar{x}_1 x_0 \vee x_2\bar{x}_1\bar{x}_0 \vee x_2\bar{x}_1 x_0 \vee x_2 x_1\bar{x}_0$$

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

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

Vereinfachungsgrundlage

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

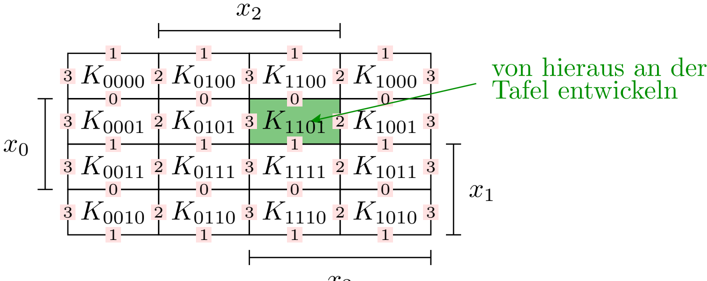
Schritt	ODER-Verknüpfung	Konjunktionsmenge
1	$\ldots \vee x_2\bar{x}_1\bar{x}_0 \vee x_2\bar{x}_1 x_0 \vee \ldots$	$\{\ldots, K_{100}, K_{101}, \ldots\}$
2	$\ldots \vee x_2\bar{x}_1 (\bar{x}_0 \vee x_0) \vee \ldots$	
3	$\ldots \vee x_2\bar{x}_1 \vee \ldots$	$\{\ldots, K_{10*}, \ldots\}$

- nur Abweichung in Stelle Eins
- Wert von x_1 »don't care«, in VHDL '-'
- eine Konjunktion weniger

KV-Diagramme

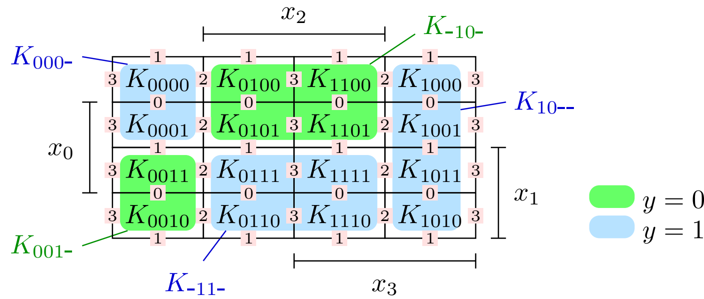
Aufbau eines KV-Diagramms

- KV Karnaugh und Veitch



- Tabellarische Anordnung der Funktionswerte so, dass sich die Konjunktionen der benachbarten Stellen genau in einer Negation unterscheiden.

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



- Minimierte Konjunktionsmenge der Einsen

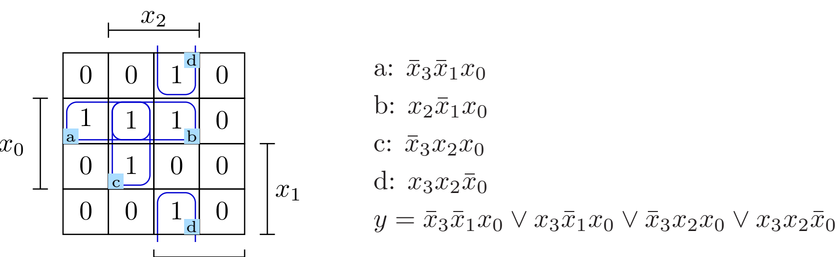
$$\{K_{000\text{-}},\ K_{\text{-11-}},\ K_{10\text{--}}\} \Rightarrow y = \bar{x}_3\bar{x}_2\bar{x}_1 \vee x_2x_1 \vee x_3\bar{x}_2$$

- Minimierte Konjunktionsmenge der Nullen:

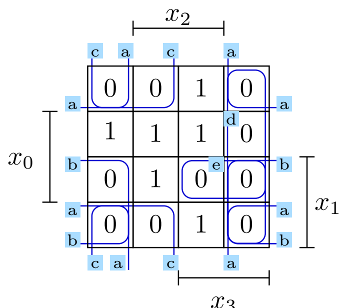
$$\{K_{001\text{-}},\ K_{\text{-10-}}\} \Rightarrow y = \overline{\bar{x}_3\bar{x}_2x_1 \vee x_2\bar{x}_1}$$

Praktische Arbeit mit KV-Diagrammen

- Geordnete Tabelle mit Funktionswerten
- Optimierungsziel: möglichst große und möglichst wenige Blöcke
- Blöcke dürfen sich überlagern



- zirkulare Blockbildung über den Rand hinaus und die Entwicklung nach den Nullen sind auch zulässig



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

b: $\bar{x}_2 x_1$

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

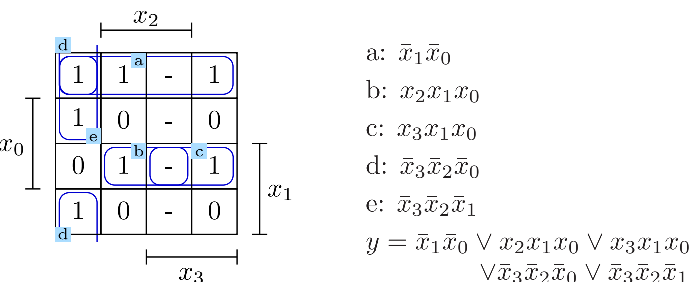
d: $x_3\bar{x}_2$

e: $x_3 x_1 x_0$

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

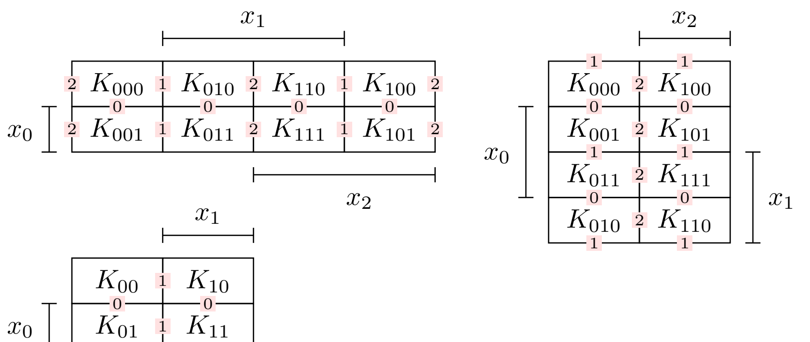
KV-Diagramme mit don't-care-Feldern

- don't-care-Felder kennzeichnen Eingabemöglichkeiten, die im normalen Betrieb nicht auftreten
- ihr Wert wird so festgelegt, dass sich möglichst wenige und möglichst große Blöcke bilden lassen

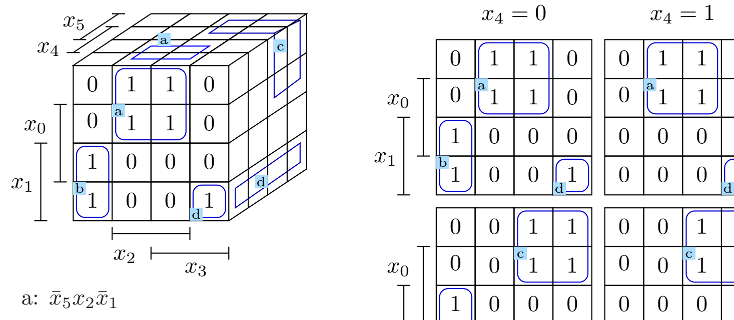


KV-Diagramme für zwei- und dreistellige Funktionen

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



KV-Diagramme für sechsstellige Funktionen



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

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

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	beliebig

y_0, y_1, y_2, y_3, y_4, y_5, y_6

y_0:

1	0	- (b)	1
0 (a)	1	-	1
1	1	-	-
1	1	-	-

y_1:

1	1	-	1
1	0	- (c)	1
1	1	-	-
1	0	- (d)	-

y_2:

1	1	-	1
1	1	-	1
1	1	-	-
0 (e)	1	-	-

y_3:

1	0	- (b)	1
0 (a)	1	-	1
1	0	- (f)	-
1	1	-	-

y_4:

1	0	- (b)	1
0	0	-	0
0	0	-	- (g)
1	1	-	-

y_5:

1	1	-	1
0 (a)	1	-	1
0	0	-	- (i)
0 (h)	1	-	-

y_6:

0	1	-	1
0 (j)	1	-	1
1	0	- (f)	-
1	1	-	-

(Spalten: x_2, x_3; Zeilen: x_0, x_1)

$a := \bar{x}_3\bar{x}_2\bar{x}_1x_0$

$b := x_2\bar{x}_1\bar{x}_0$

$c := x_2\bar{x}_1x_0$

$d := x_2x_1\bar{x}_0$

$e := \bar{x}_2x_1\bar{x}_0$

...

```
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;
```

y_0

1	0	-	1
0	1	-	1
1	1	-	-
1	1	-	-

y_1

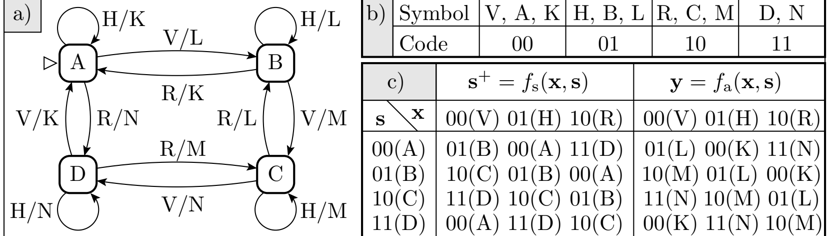
1	1	-	1
1	0	-	1
1	1	-	-
1	0	-	-

y_2

1	1	-	1
1	1	-	1
1	1	-	-
0	1	-	-

x_2 (columns 2–3), x_3 (columns 3–4), x_0 (rows 2–3), x_1 (rows 3–4); groups: a, b, c, d, e

Beispiel für einen Automatenentwurf



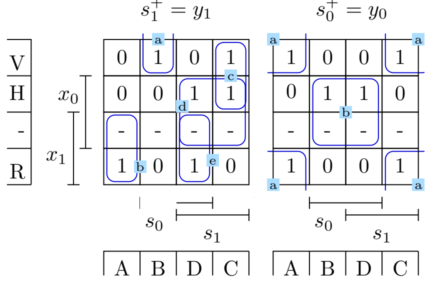
b) Symbol	V, A, K	H, B, L	R, C, M	D, N
Code	00	01	10	11

c)	$\mathbf{s}^+ = f_\mathrm{s}(\mathbf{x}, \mathbf{s})$			$\mathbf{y} = f_\mathrm{a}(\mathbf{x}, \mathbf{s})$		
s \ **x**	00(V)	01(H)	10(R)	00(V)	01(H)	10(R)
00(A)	01(B)	00(A)	11(D)	01(L)	00(K)	11(N)
01(B)	10(C)	01(B)	00(A)	10(M)	01(L)	00(K)
10(C)	11(D)	10(C)	01(B)	11(N)	10(M)	01(L)
11(D)	00(A)	11(D)	10(C)	00(K)	11(N)	10(M)

V, H, R	symbolische Eingabewerte	$\mathbf{x}$	Eingabesignal (2 Bit)
A, B, C, D	symbolische Zustände	$\mathbf{s}$, $\mathbf{s}^+$	Zustand, Folgezustand (je 2 Bit)
K, L, M, N	symbolische Ausgabewerte	$\mathbf{y}$	Ausgabesignal

- Zustandscodierung ist so gewählt, dass $\mathbf{y} = \mathbf{s}^+$ gilt

Aufstellen der KV-Diagramme



$$y_0 = s_0^+ \quad = \quad \bar{s}_0\bar{x}_0 \vee s_0x_0$$
$$y_1 = s_1^+ \quad = \quad \bar{s}_1 s_0 \bar{x}_1 \bar{x}_0 \vee \bar{s}_1 \bar{s}_0 x_1 \vee s_1 s_0 x_1 \vee \bar{s}_1 x_1 \bar{x}_0 \vee s_1 \bar{s}_0 \bar{x}_1 \vee s_1 x_0$$

Vom KV-Diagramm zur Schaltung

- Auswahl der zu verwendenden Gatter und Speicherzellen (Technologieabbildung)

- Beispielbausteinsystem: Inverter + NAND-Gatter mit unterschiedlicher Eingangsanzahl + 1-Bit-Register mit Rücksetzeingang
- Gleichungsumformung mit de morganscher Regel

$$y_0 = s_0^+ = \bar{s}_0\bar{x}_0 \vee s_0 x_0 = \overline{\left(\overline{\bar{s}_0\bar{x}_0}\right)\left(\overline{s_0 x_0}\right)}$$

$$\begin{aligned} y_1 = s_1^+ &= \bar{s}_1 s_0 \bar{x}_1 \bar{x}_0 \vee \bar{s}_1 \bar{s}_0 x_1 \vee s_1 s_0 x_1 \vee \bar{s}_1 x_1 \bar{x}_0 \vee s_1 \bar{s}_0 \bar{x}_1 \vee s_1 x_0 \\ &= \overline{\left(\overline{\bar{s}_1 s_0 \bar{x}_1 \bar{x}_0}\right)\left(\overline{\bar{s}_1 \bar{s}_0 x_1}\right)\left(\overline{s_1 s_0 x_1}\right)\left(\overline{s_1 \bar{s}_0 \bar{x}_1}\right)\left(\overline{s_1 x_0}\right)} \end{aligned}$$

$$y_0 = s_0^+ = \overline{\left(\overline{\bar{s}_0\bar{x}_0}\right)\left(\overline{s_0x_0}\right)}$$

$$y_1 = s_1^+ = \overline{\left(\overline{\bar{s}_1 s_0\bar{x}_1\bar{x}_0}\right)\left(\overline{\bar{s}_1\bar{s}_0 x_1}\right)\left(\overline{s_1 s_0 x_1}\right)\left(\overline{s_1\bar{s}_0\bar{x}_1}\right)\left(\overline{s_1 x_0}\right)}$$

⇒Web-Projekt:
P2.2/Test_VRZ.vhdl

Quine und McCluskey

Optimierung nach Quine & McCluskey – Prinzip

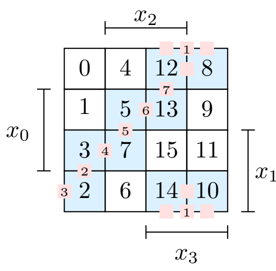
Alternatives tabellenbasiertes Minimierungsverfahren, das gleichfalls auf der Zusammenfassung von Konjunktionen, die sich nur in einer Negation unterscheiden, basiert:

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

- Zusammenstellen der Menge aller Minterme, für die der Funktionswert Eins (bzw. Null) ist ⇒ 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 ⇒ quinesche Tabelle erster Ordnung
- etc.
- Ausdrucksminimierung mit Hilfe der Abdeckungstabelle der Primterme

Aufstellen der quineschen Tabellen

Visualisierung der Blockbildungsmöglichkeiten mit einem KV-Diagramm



- mögliche Zweierblöcke
- möglicher Viererblock

$\surd$ abgedeckte Konjunktionen

P_i Primterm

quinesche Tabelle nullte Ordnung

	$x_3\,x_2\,x_1\,x_0$	
2	0 0 1 0	$\surd$
8	1 0 0 0	$\surd$
3	0 0 1 1	$\surd$
5	0 1 0 1	$\surd$
10	1 0 1 0	$\surd$
12	1 1 0 0	$\surd$
7	0 1 1 1	$\surd$
13	1 1 0 1	$\surd$
14	1 1 1 0	$\surd$

quinesche Tabellen erste und zweite Ordnung

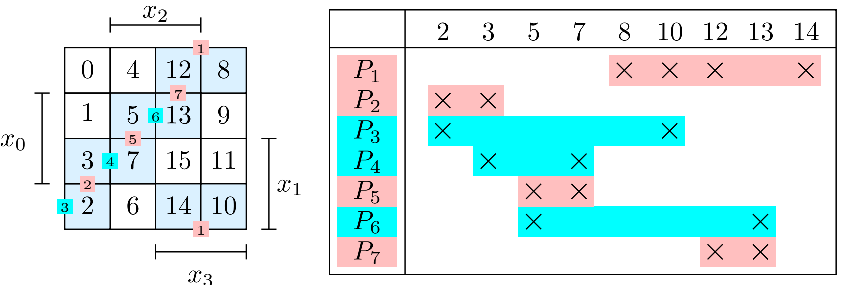
	$x_3\,x_2\,x_1\,x_0$	
2, 3	0 0 1 -	P_2
2, 10	- 0 1 0	P_3
8, 10	1 0 - 0	$\surd$
8, 12	1 - 0 0	$\surd$
3, 7	0 - 1 1	P_4
5, 7	0 1 - 1	P_5
5, 13	- 1 0 1	P_6
10, 14	1 - 1 0	$\surd$
12, 13	1 1 0 -	P_7
12, 14	1 1 - 0	$\surd$

	$x_3\,x_2\,x_1\,x_0$	
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: $\{P_1, P_2, P_5, P_7\}$:

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

- 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.

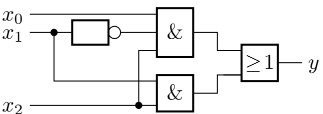
Zusammenfassung des Gesamtabschnitts:

- 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.

Aufgaben

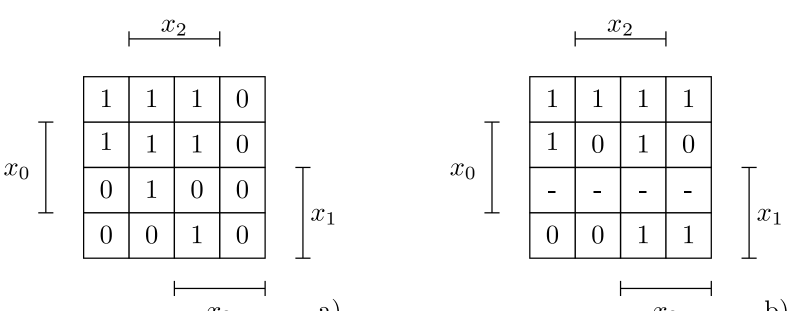
Aufgabe 2.7: Schaltungsumformung

Wandeln Sie die Schaltung in eine Schaltung aus NAND-Gattern und Invertern um.



Aufgabe 2.8: Schaltungsminimierung mit KV-Diagrammen

Lesen Sie aus den KV-Diagrammen minimierte logische Ausdrücke ab.



Aufgabe 2.9: Fragen zu KV-Diagrammen

1. Warum dürfen sich die Rechtecke, mit denen in KV-Diagrammen die Einsen[3] abgedeckt werden, gegenseitig überlagern?
2. In welchen Mintermen überlagern sich die Rechtecke im KV-Diagramm für den Ausdruck:

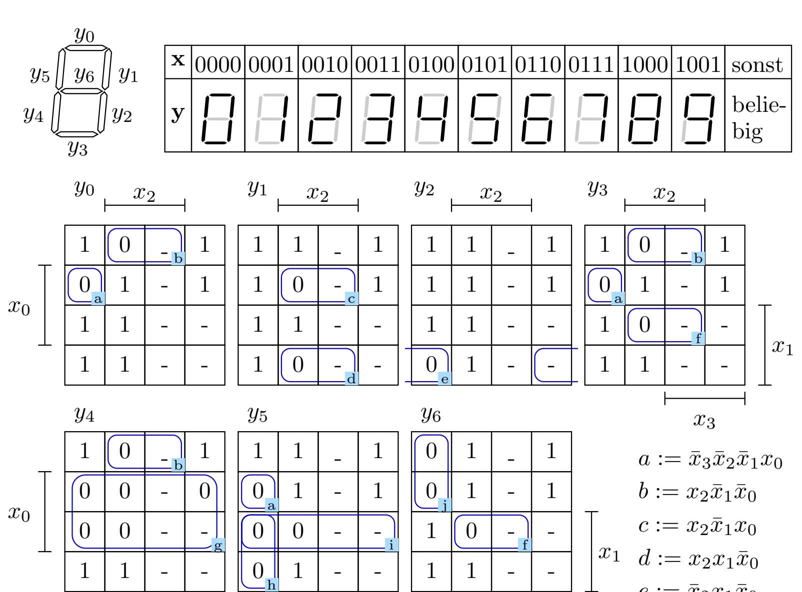
$$\bar{x}_0 \vee x_2 x_1 \vee x_3 x_2$$

[3] Bei einer Entwicklung nach »0« die Nullen.

Aufgabe 2.10: 7-Segment-Decoder

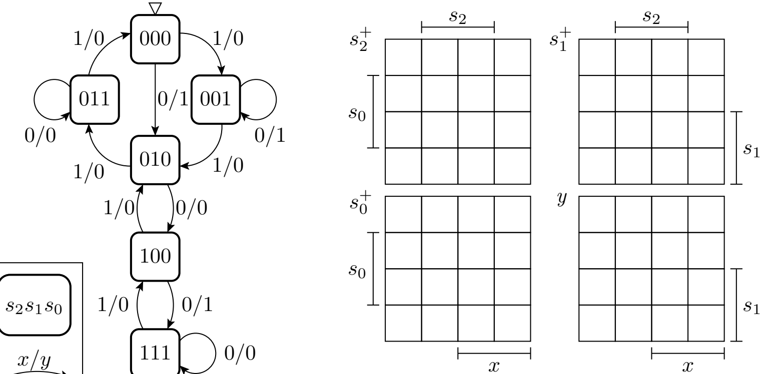
Ergänzen Sie die fehlenden Zuweisungen an die Variablen f bis j und die Signale y_3 bis y_6 in der VHDL-Beschreibung des 7-Segment-Decoders Seite 94.

```
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
--- auf den Folien ab Seite 94
 a := ... e:= ...; y(0)<= ...  y(2)<= ...;
--- Aufgabe: Ergänzen der zuzuweisenden Ausdrücke
an
 f:= ... j:= ...;
 y(3)<= ...  y(6)<=...;
end process;
```



x
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
sonst
y
belie-
big
y_0 y_1 y_2 y_3 y_4 y_5 y_6
x_0 x_1 x_2 x_3
$a := \bar{x}_3\bar{x}_2\bar{x}_1 x_0$
$b := x_2\bar{x}_1\bar{x}_0$
$c := x_2\bar{x}_1 x_0$
$d := x_2 x_1\bar{x}_0$
$e := \bar{x}_2 x_1\bar{x}_0$
...

Aufgabe 2.11: Automatenentwurf



Kompletter Handentwurf einer funktionsgleichen Schaltung

- KV-Diagramme ausfüllen (die Folgezustände und die Ausgabe der redundanten Zustände seien beliebig)
- minimierte Ausdrücke bestimmen (legt die Werte der Don't-Care-Stellen fest)
- redundante Zustände und deren abgehende Kanten in den Ablaufgraphen einzeichnen (hängt von den Wertzuordnung an die Don't-Care-Stellen ab)
- Schaltung zeichnen; zu verwendende Bausteine: Inverter, UND- und ODER-Gatter mit variabler Eingangsanzahl, 1-Bit-Register mit Rücksetzeingang

Dateneingang — x — Datenausgang
Rücksetzeingang — R — invertierter Datenausgang
Takteingang

Aufgabe 2.12: Schaltungsminimierung nach Quine und McCluskey

- Gegeben ist die Menge Minterme, für die der Funktionswert ≫1≪ ist:

$$K \in \{100000, 100100, 101010, 101110, 111110, 110000, 011000, 101011, 101111, 101000, 101001\}$$

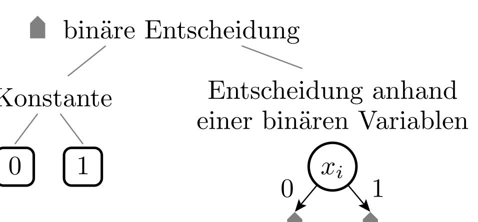
- Aufstellen der quineschen Tabellen
- Aufstellen der Tabelle der Primterme
- Suche der minimalen Abdeckungsmenge

BDD

Binäres Entscheidungsdiagramm

(BDD – binary decision diagram)

- Zusammensetzung einer Funktion aus binären Entscheidungen
- Rekursive Definition: Eine binäre Entscheidung ist eine Konstante oder eine Entscheidung zwischen binären Entscheidungen



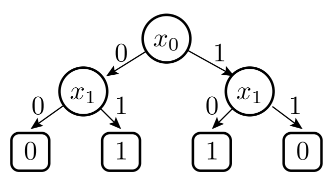
Wertetabelle als binäres Entscheidungsdiagramm

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

$\Longrightarrow$

Binären Entscheidungsdiagramm als Programm

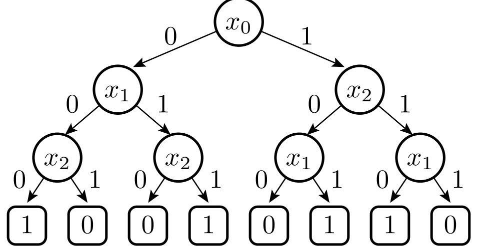
```
if x0='0' then
 if x1='0' then
  y<='0';
 else
  y<='1';
 end if;
else
 if x1='0' then
  y<='1';
 else
  y<='0';
 end if;
end if;
```



Vereinfachungsregeln

Unreduziertes binäres Entscheidungsdiagramm

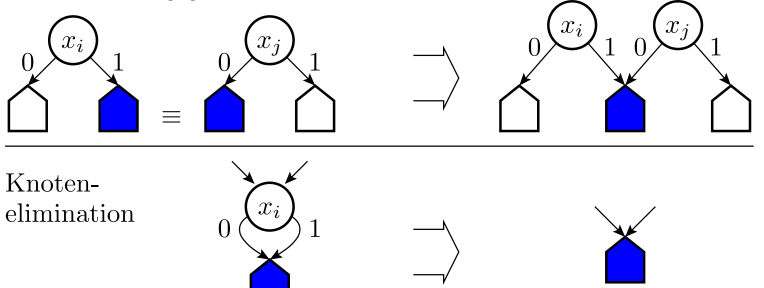
- Abkürzung: BDD (binary decision diagram)



- Baum
- direkt ablesbar aus der Wertetabelle

Vereinfachungsregeln

- Verschmelzung gleicher Teilgraphen
- Löschen von Knoten mit zwei gleichen Nachfolgern

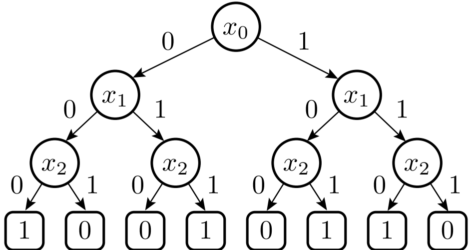


Offensichtliche Optimierungsvoraussetzung: gleiche Abfragereihenfolge auf allen Entscheidungswegen

Geordnetes binäres Entscheidungsdiagramm

Gleiche Abfragereihenfolge auf allen Entscheidungswegen.
Abkürzung:

- OBDD (ordered binary decision diagram)



Bezeichnung für ein vereinfachtes binäres Entscheidungsdiagr.:

- ROBDD (reduced ordered binary decision diagram)

Vereinfachung am Beispiel

unterste Ebene

x_0 — 0: x_1 (0: x_2 (0: Υ_1, 1: Υ_2), 1: x_2 (0: Υ_2, 1: Υ_1)); 1: x_1 (0: x_2 (0: Υ_2, 1: Υ_1), 1: x_2 (0: Υ_1, 1: Υ_2))

unterschiedliche Teilgraphen:

Υ_1 :	Υ_2 :	Υ_3 :	Υ_4 :
1	0	x_2: 0 → Υ_1, 1 → Υ_2	x_2: 0 → Υ_2, 1 → Υ_1

nächste Ebene

x_0 — 0: x_1 (0: Υ_3, 1: Υ_4); 1: x_1 (0: Υ_4, 1: Υ_3)

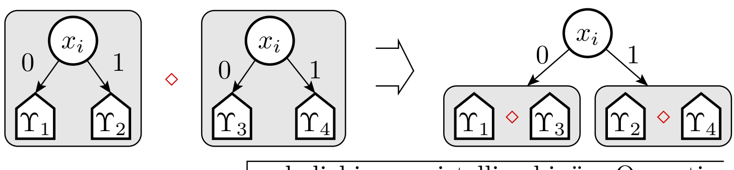
Ergebnis

x_0 — 0: x_1 (0: x_2, 1: x_2); 1: x_1 (0: x_2, 1: x_2); x_2 (0: 1, 1: 0); x_2 (0: 0, 1: 1)

Operationen mit ROBDDs

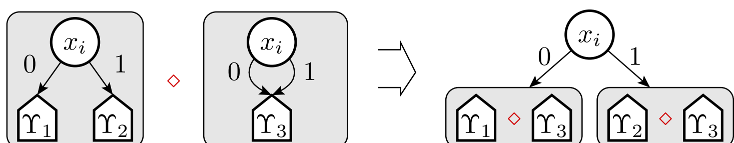
Zweistellige Operationen mit ROBDD

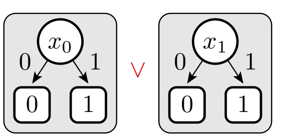
- für zwei beliebige Teilbäume, bei denen im obersten Knoten dieselbe Variable ausgewertet wird, verschiebt sich die Operation eine Entscheidungsebene tiefer



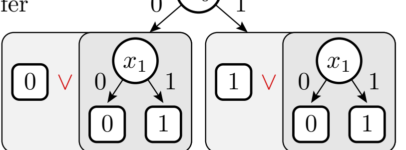
- in einem ROBDD erfolgen auf allen Wegen die Entscheidungen in derselben Reihenfolge

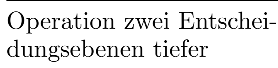
- fehlende Entscheidungsknoten sind wie Entscheidungsknoten mit gleichen Nachfolgern zu behandeln

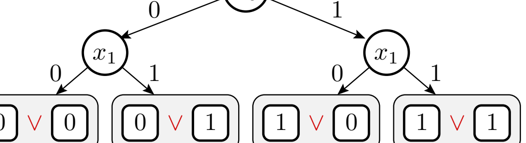




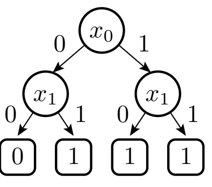
Operation ein Entscheidungsebene tiefer



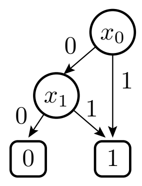




Ausführung der Operation auf der untersten Ebene



Vereinfachung



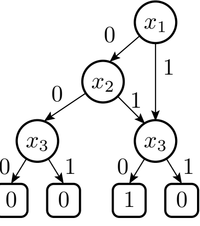
Verknüpfung
$(x_1 \vee x_2) \wedge \bar{x}_3$

Operation zwei Entscheidungsebenen tiefer

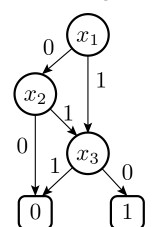
Operation drei Entscheidungsebenen tiefer

$0 \wedge 1$ | $0 \wedge 0$ | $1 \wedge 1$ | $1 \wedge 0$

Ausführung der Operation auf der untersten Ebene



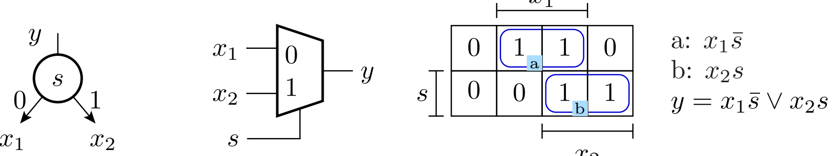
Vereinfachung



ROBDD ⇒ minimierte Schaltung

Entscheidungsknoten als Signalflussumschalter

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



- minimiert mit einem KV-Diagramm:

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

- eine konstante Eingabe ⇒ 2-stellige logische Funktion

$$x_1 = 0: \quad y = (0 \wedge \bar{s}) \vee x_2 s \quad = x_2 s$$

$$x_1 = 1: \quad y = (1 \wedge \bar{s}) \vee x_2 s \quad = \bar{s} \vee x_2 s = \bar{s} \vee x_2$$

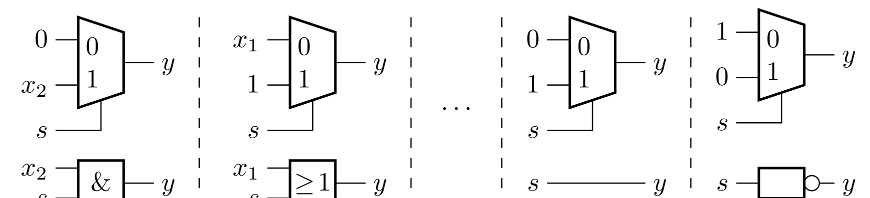
$$x_2 = 0: \quad y = x_1 \bar{s} \vee (0 \wedge s) \quad = x_1 \bar{s}$$

$$x_2 = 1: \quad y = x_1 \bar{s} \vee (1 \wedge s) \quad = x_1 \bar{s} \vee s = x_1 \vee s$$

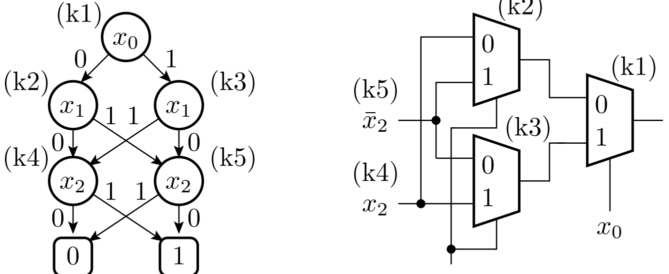
- zwei konstante Eingaben ⇒ 1-stellige logische Funktion

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

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



Entscheidungsdiagramm als Signalflussplan

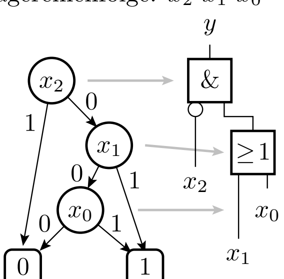


Abfragereihenfolge und Schaltungsaufwand

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 (mit Ausdrücken ist die Anzahl der Darstellungsmöglichkeiten unbegrenzt)
- 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

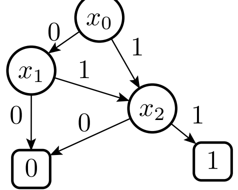
Aufgaben

Aufgabe 2.13: VHDL-Beschreibung ⇒ Wertetabelle ⇒ OBDD ⇒ ROBDD ⇒ Schaltung

```
signal  a, b, c, y: STD_LOGIC;
...
 y <= a xor b xor c;
```

- Stellen Sie die Wertetabelle für die Schaltung auf.
- Entwickeln Sie aus der Wertetabelle das unreduzierte OBDD für die Abfragereihenfolge a-b-c.
- Entwickeln Sie mit Hilfe der Vereinfachungsregeln – Verschmelzung und Knotenelimination – das zugehörige ROBDD.
- Entwickeln Sie aus dem ROBDD eine optimierte Schaltung.

Aufgabe 2.14: ROBDD ⇒ Wertetabelle, Schaltung



- Stellen Sie die Wertetabelle auf.
- Bilden Sie das Entscheidungsdiagramm durch einen Datenflussplan nach.