



Kapitel 12: Schaltelemente und Hardwaresynthese

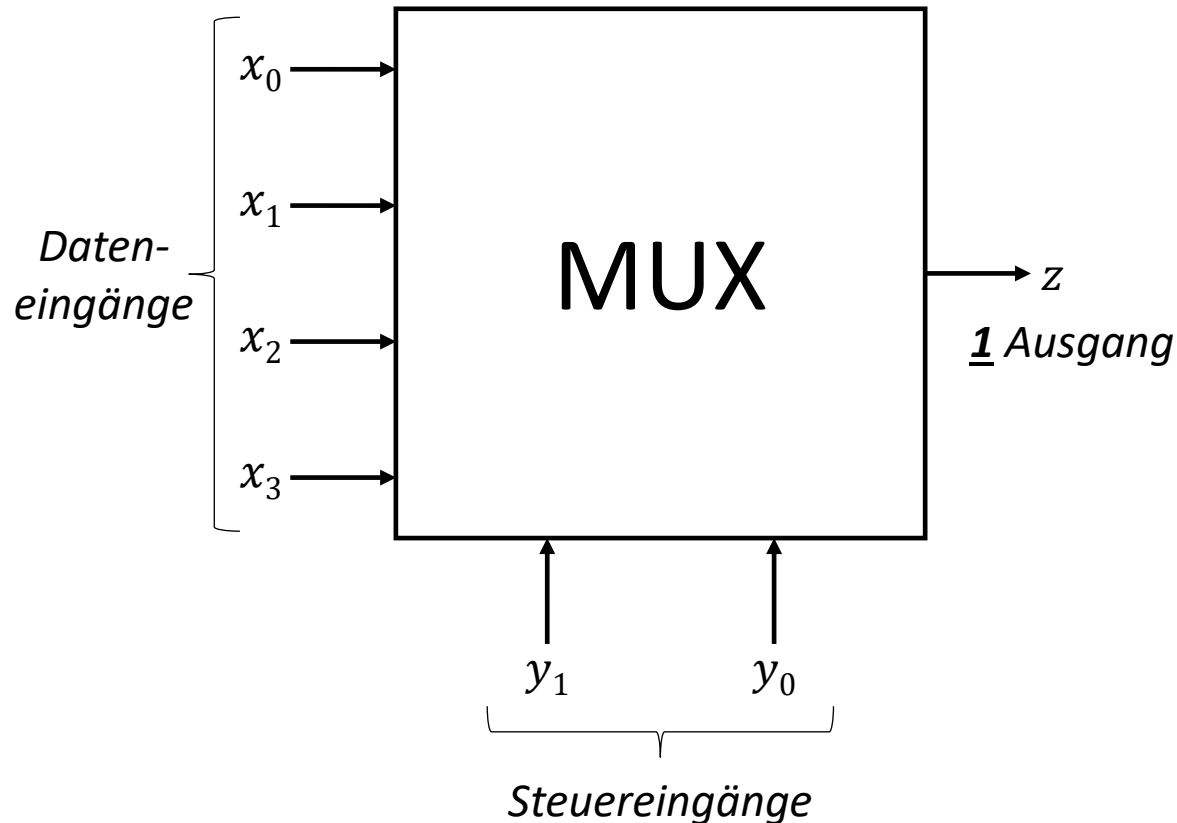
Abschnitt 12.1

MUX/DeMUX

- ▶ Allgemeiner MUX/DeMUX-Aufbau
- ▶ Top-Down-Multiplexer-Entwurf
- ▶ MUX zur Realisierung Boolescher Funktionen

2-MUX (Prinzip)

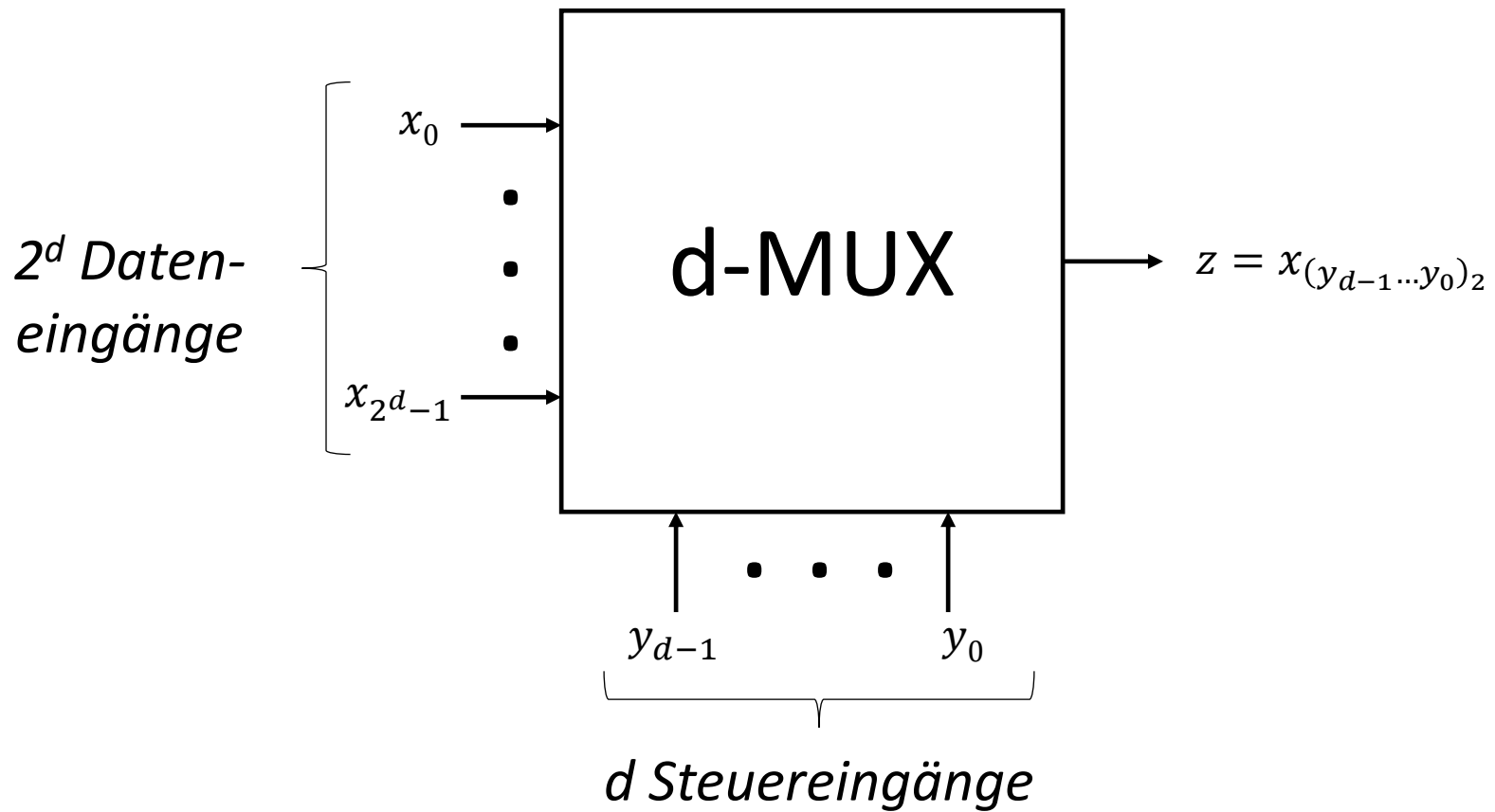
Funktion: Auswahl des Dateneingangs, der auf den Ausgang geht



y_1	y_0	z
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

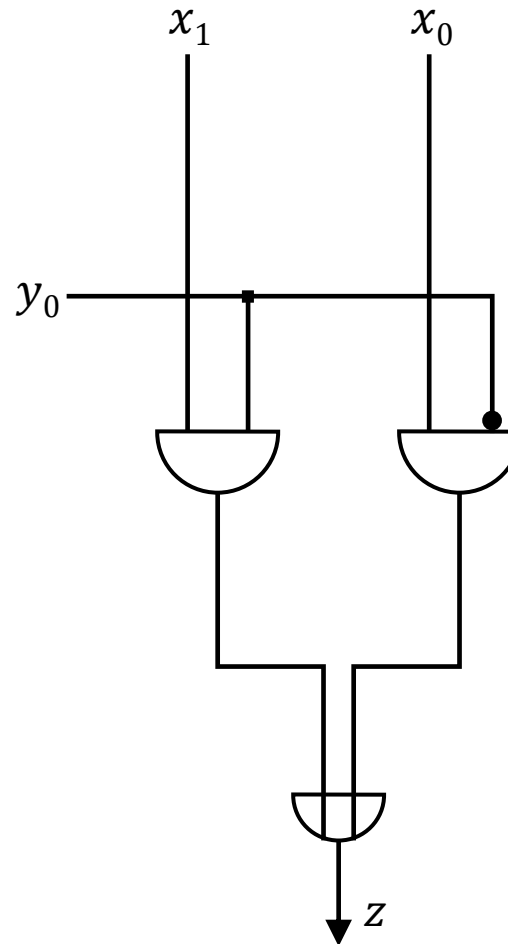
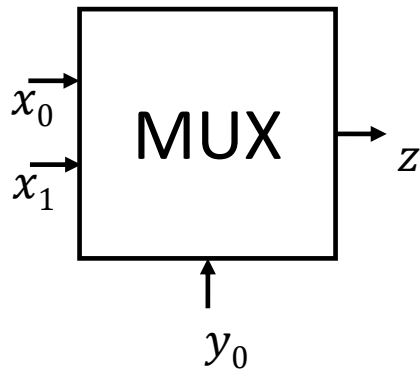
© G. Lakemeyer, W. Oberschelp, G. Vossen

Allgemeiner MUX-Aufbau



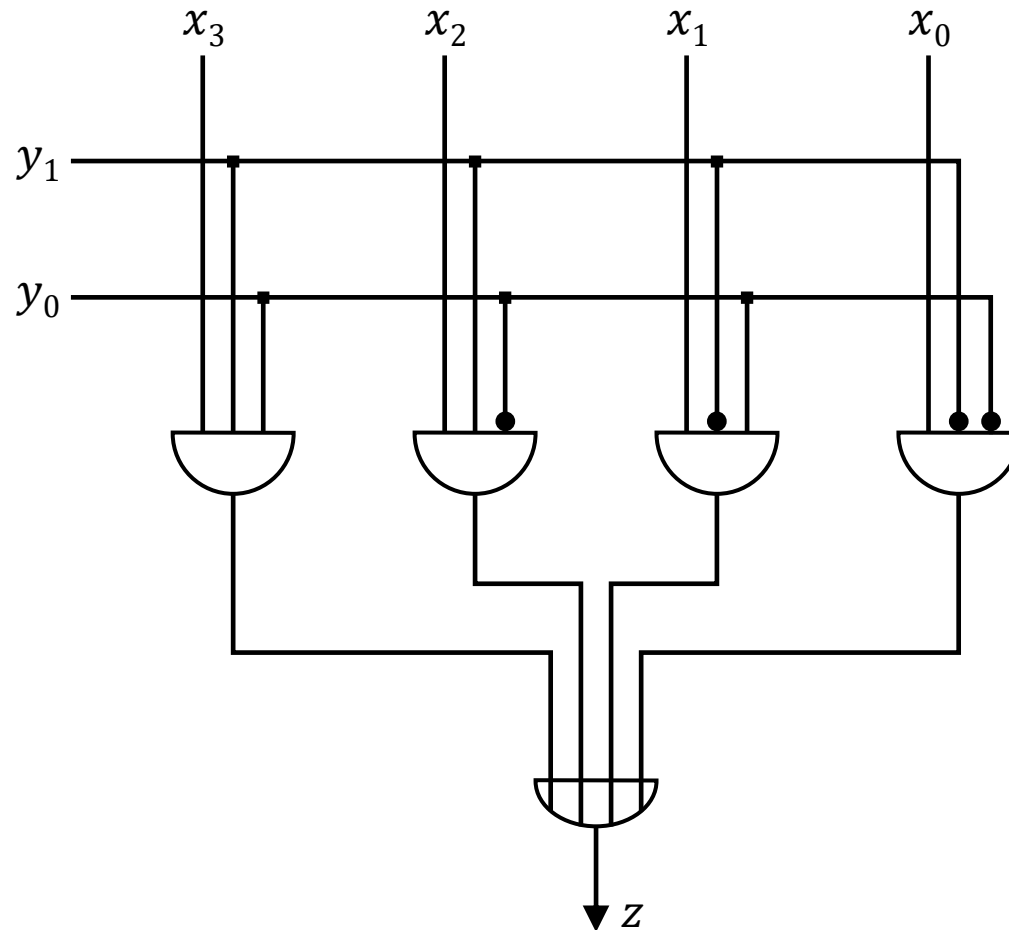
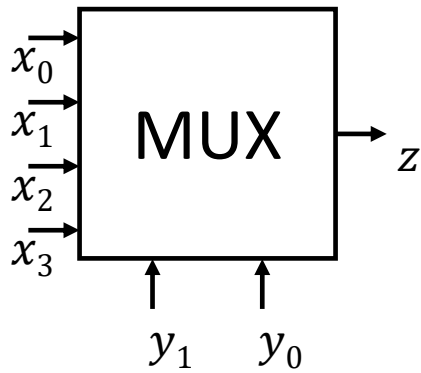
© G. Lakemeyer, W. Oberschelp, G. Vossen

Realisierung eines 1-MUX



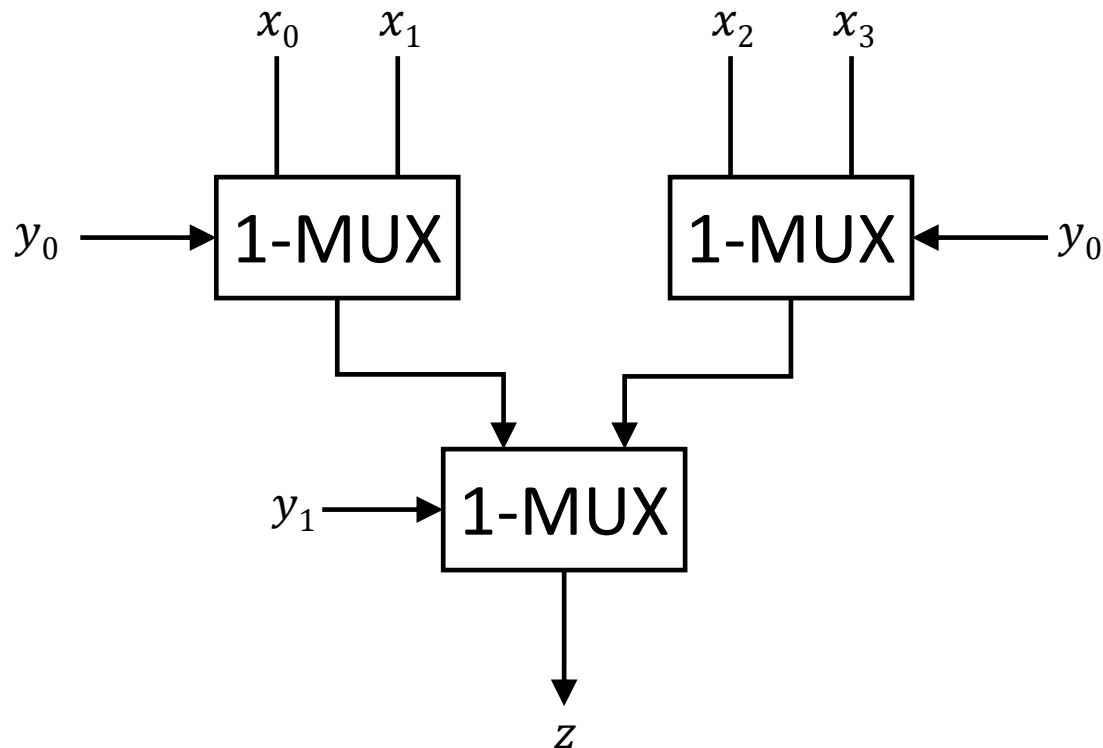
© G. Lakemeyer, W. Oberschelp, G. Vossen

Realisierung eines 2-MUX



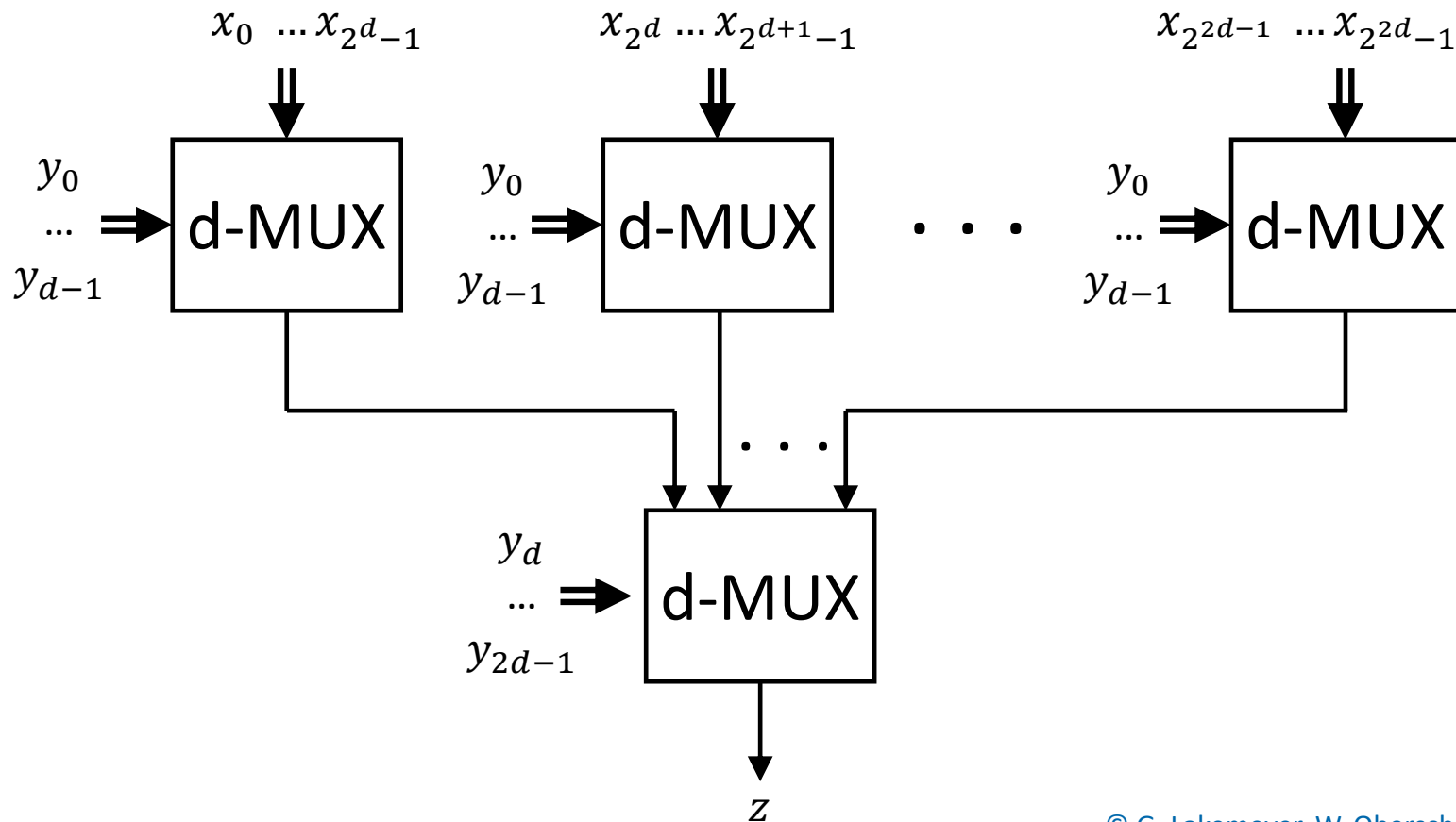
© G. Lakemeyer, W. Oberschelp, G. Vossen

Systematische Konstruktion des 2-MUX



© G. Lakemeyer, W. Oberschelp, G. Vossen

Top-Down-Multiplexer-Entwurf (Rekursion)



© G. Lakemeyer, W. Oberschelp, G. Vossen

MUX zur Realisierung Boolescher Funktionen

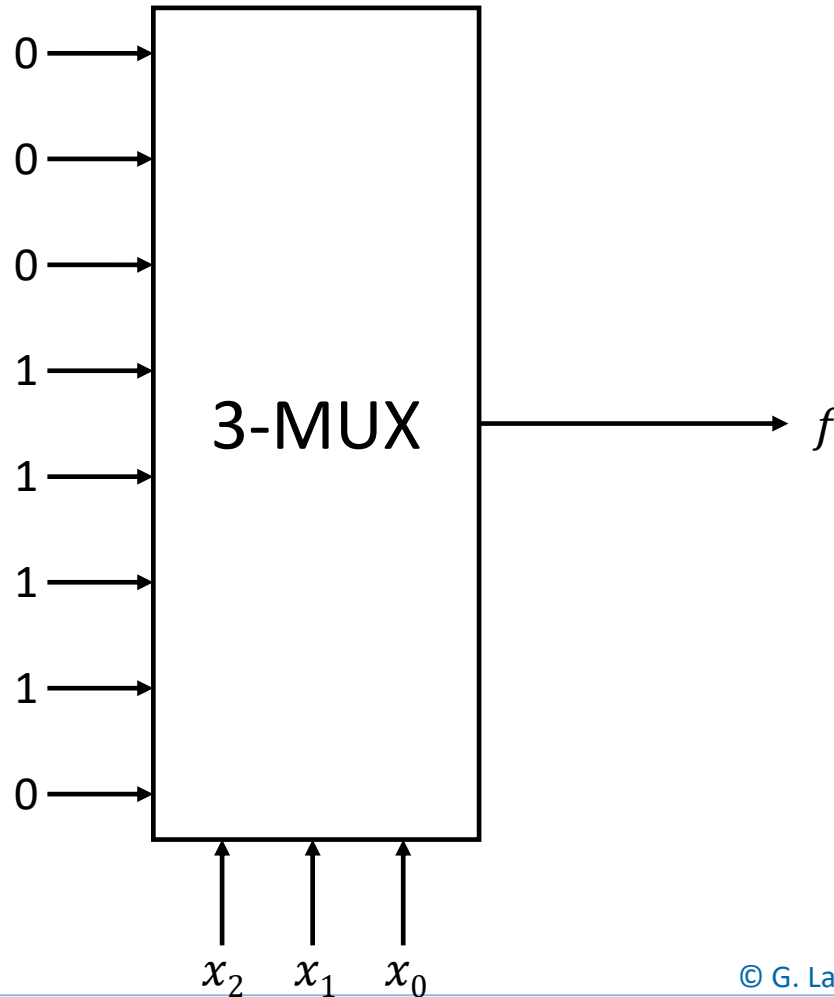
1. Möglichkeit: Verwendung von Mintermen

Betrachte:

x_2	x_1	x_0	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

© G. Lakemeyer, W. Oberschelp, G. Vossen

$$f(x_2, x_1, x_0) = m_3 + m_4 + m_5 + m_6$$



© G. Lakemeyer, W. Oberschelp, G. Vossen

MUX zur Realisierung Boolescher Funktionen

2. Möglichkeit: Verwendung von x_0 und \bar{x}_0 als Eingang

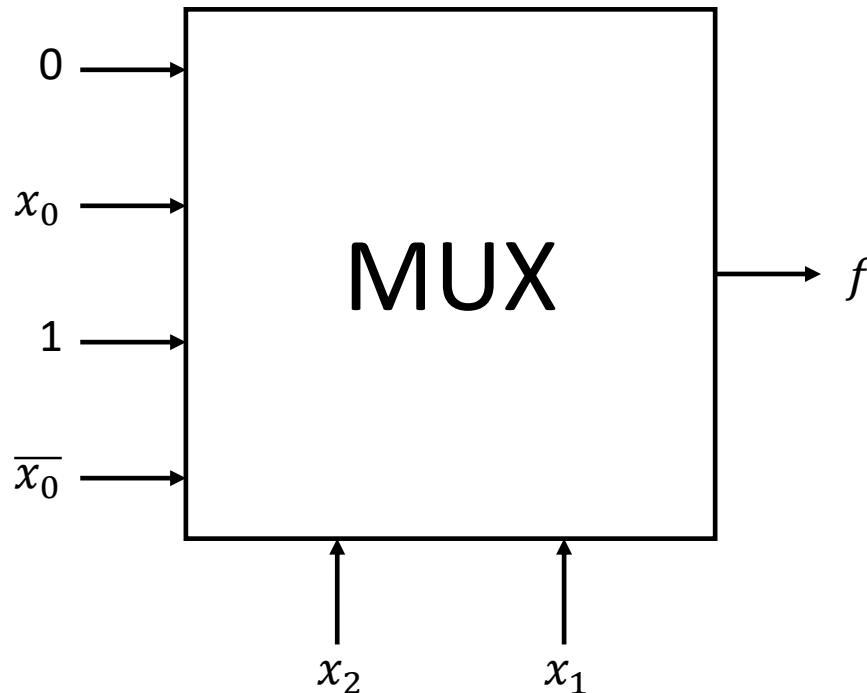
Betrachte:

x_2	x_1	x_0	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

© G. Lakemeyer, W. Oberschelp, G. Vossen

Realisierung einer Funktion

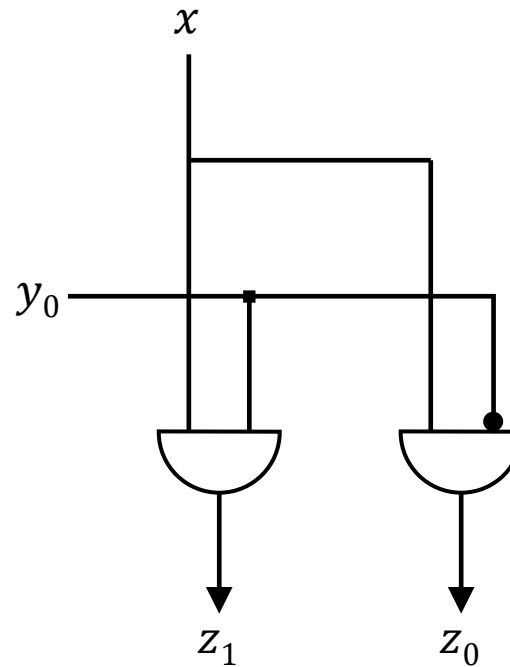
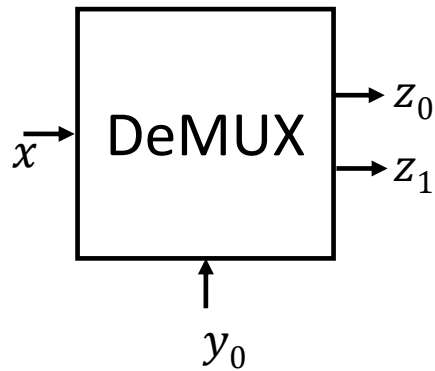
f dargestellt in alleiniger Abhängigkeit von x_2 und x_1 :



x_2	x_1	f
0	0	0
0	1	x_0
1	0	1
1	1	$\overline{x_0}$

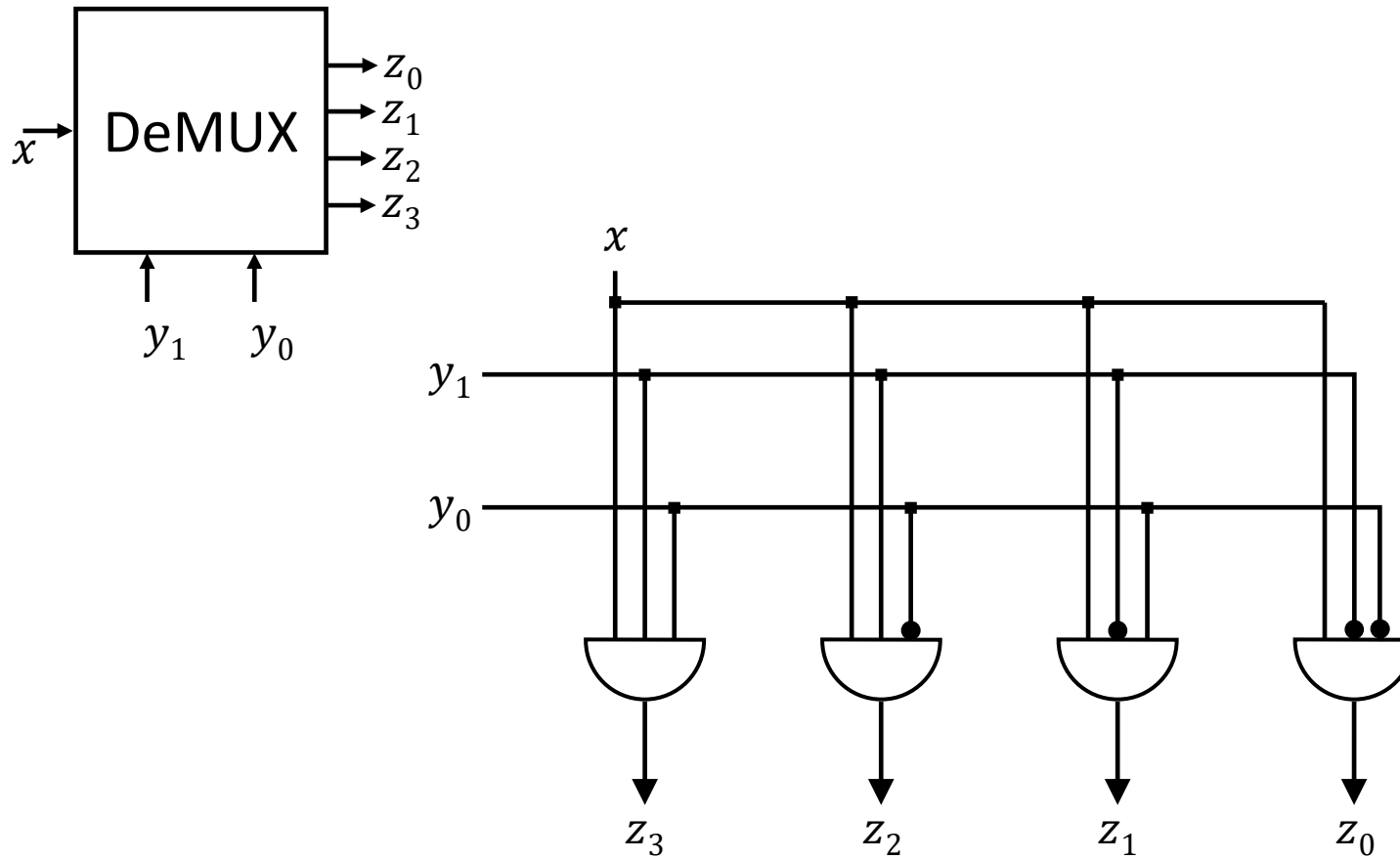
© G. Lakemeyer, W. Oberschelp, G. Vossen

1-DeMUX (Demultiplexer)



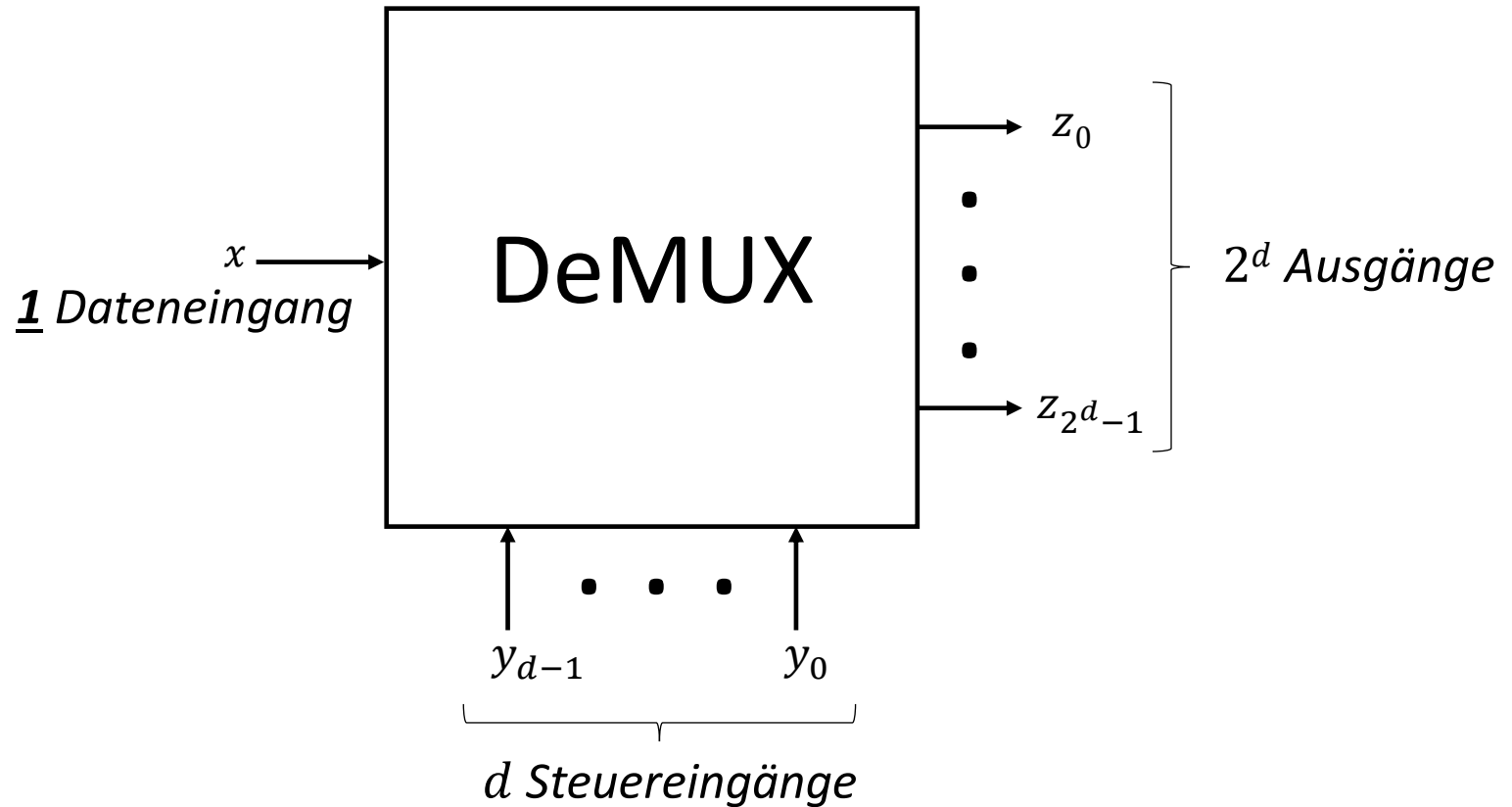
© G. Lakemeyer, W. Oberschelp, G. Vossen

2-DeMUX



© G. Lakemeyer, W. Oberschelp, G. Vossen

Allgemeiner Aufbau eines DeMUX



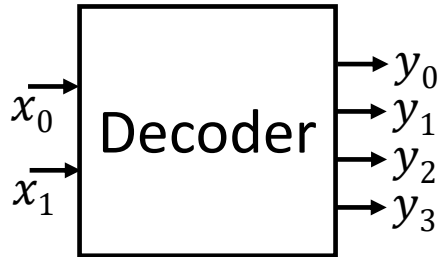
© G. Lakemeyer, W. Oberschelp, G. Vossen

Abschnitt 12.2

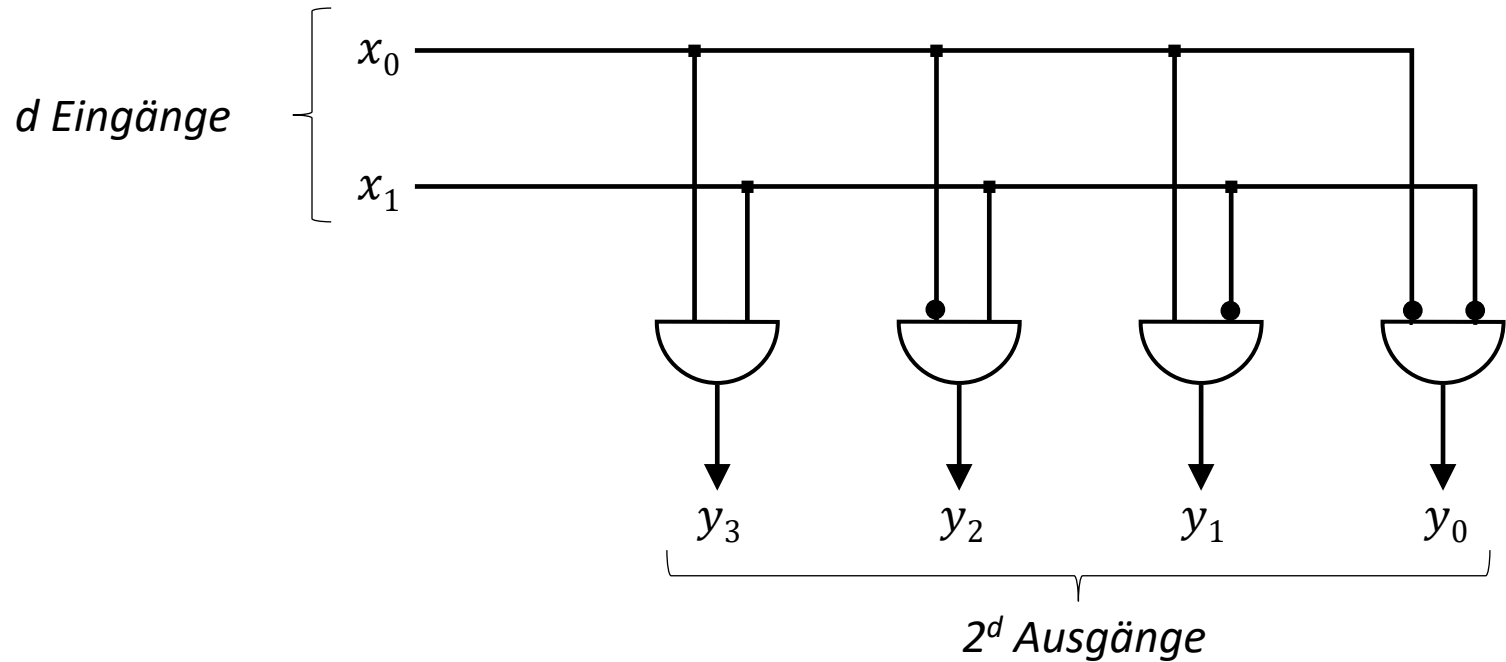
Encoder/Decoder

- ▶ Encoder/Decoder-Aufbau
- ▶ Decoder zur Realisierung Boolescher Funktionen

2x4-Decoder

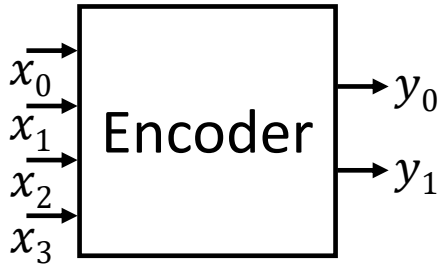


Funktion: Umwandeln des binären Eingangswerts in einen Dezimalwert

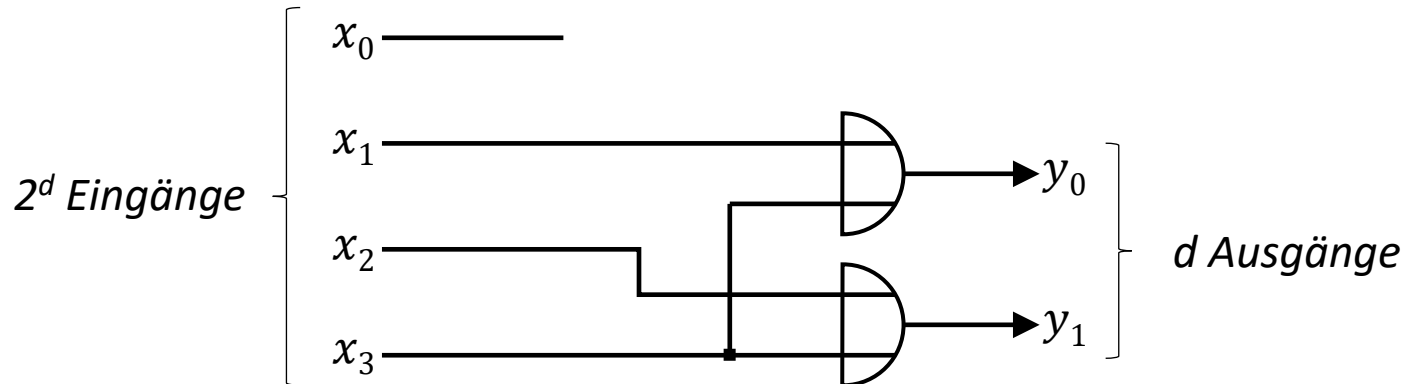


© G. Lakemeyer, W. Oberschelp, G. Vossen

4x2-Encoder



Funktion: Umwandeln des dezimalen Eingangswerts in einen Binärwert



x_3	x_2	x_1	x_0	y_1	y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

© G. Lakemeyer, W. Oberschelp, G. Vossen

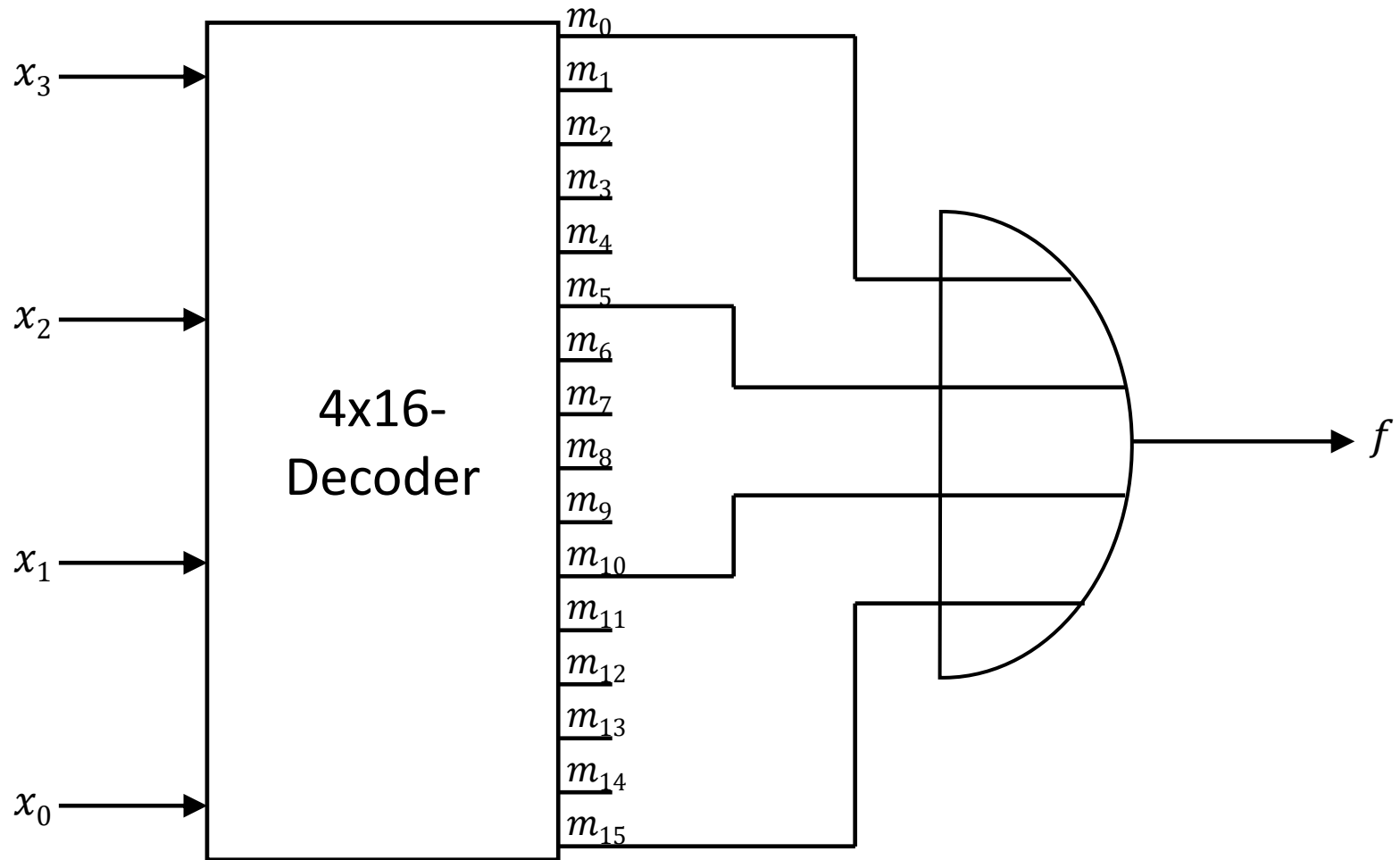
Beispiel:

$$f(x_3, x_2, x_1, x_0) = \overline{x_3} \overline{x_2} \overline{x_1} \overline{x_0} + \overline{x_3} x_2 \overline{x_1} x_0 + x_3 \overline{x_2} x_1 \overline{x_0} + x_3 x_2 x_1 x_0$$

1. mittels MUX: siehe oben

Realisierung Boolescher Funktionen

2. mittels Decoder:



© G. Lakemeyer, W. Oberschelp, G. Vossen

3. mittels Kombination von Decoder und MUX:

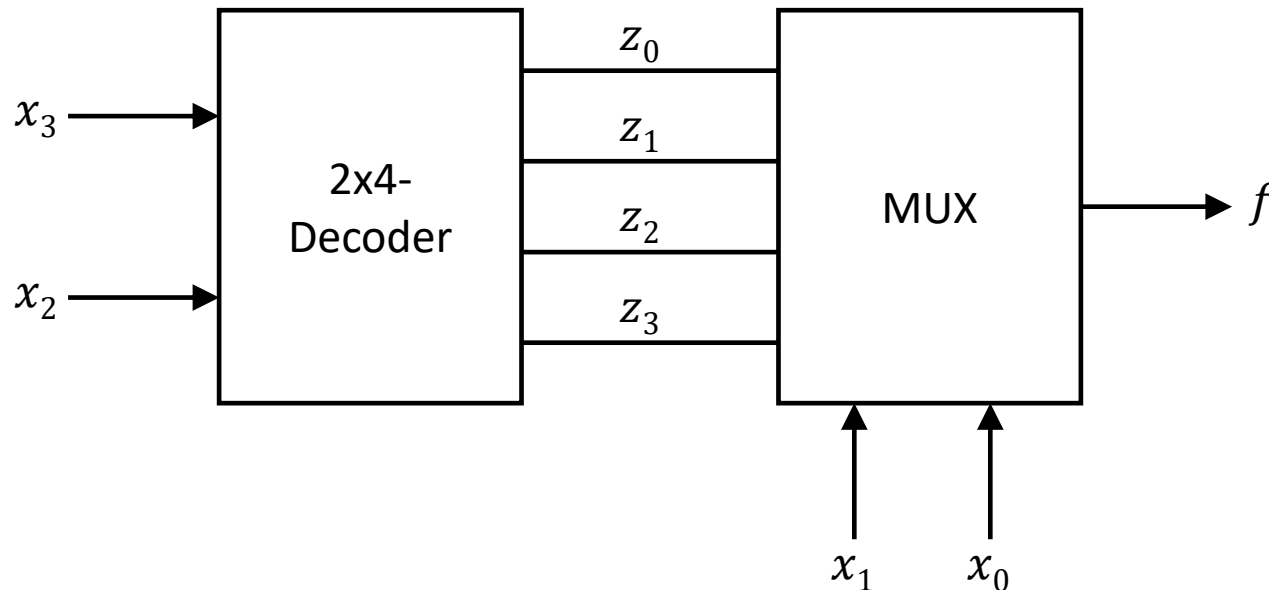
Es gibt 4 Input-Kombinationen, für welche $f = 1$ gilt:

$$x_3x_2 = 00 \text{ und } x_1x_0 = 00$$

$$x_3x_2 = 01 \text{ und } x_1x_0 = 01$$

$$x_3x_2 = 11 \text{ und } x_1x_0 = 11$$

$$x_3x_2 = 10 \text{ und } x_1x_0 = 10$$



Abschnitt 12.3

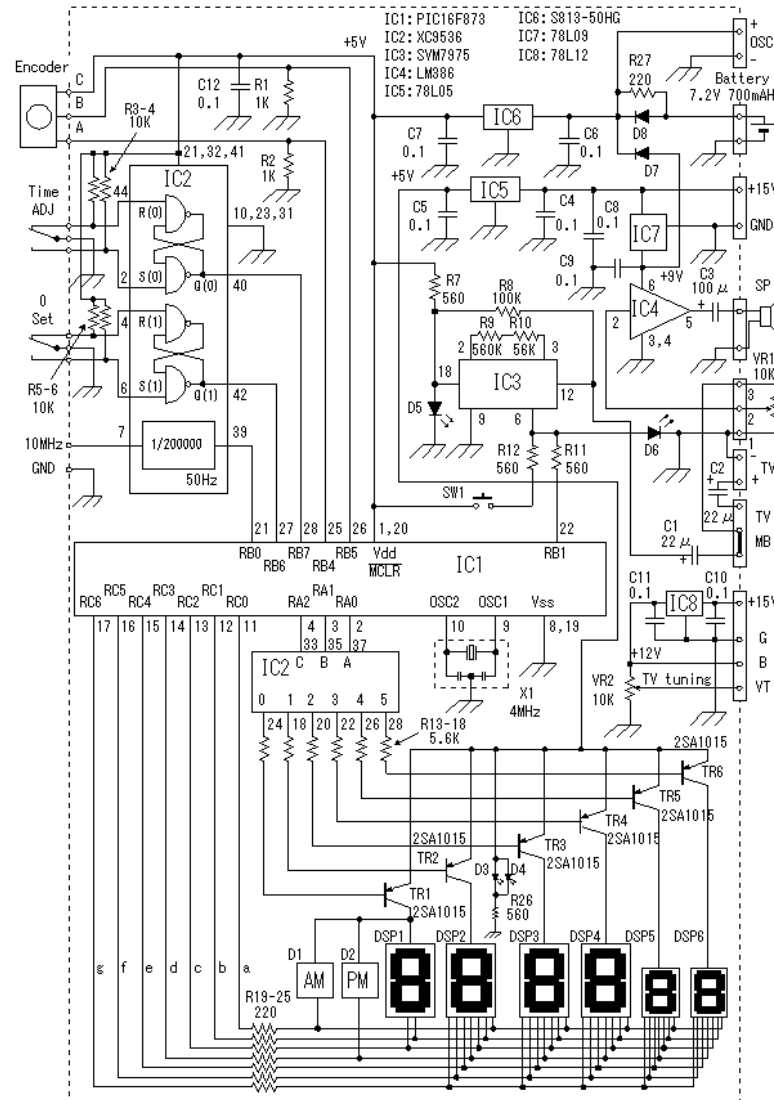
Historie der Hardwaresynthese



- Bis in die 70er Jahre:
 - Elektronische Schaltungen werden aus diskreten Bausteinen zusammengesetzt
 - Einzige „formale“ Beschreibung: „Schematics“

24

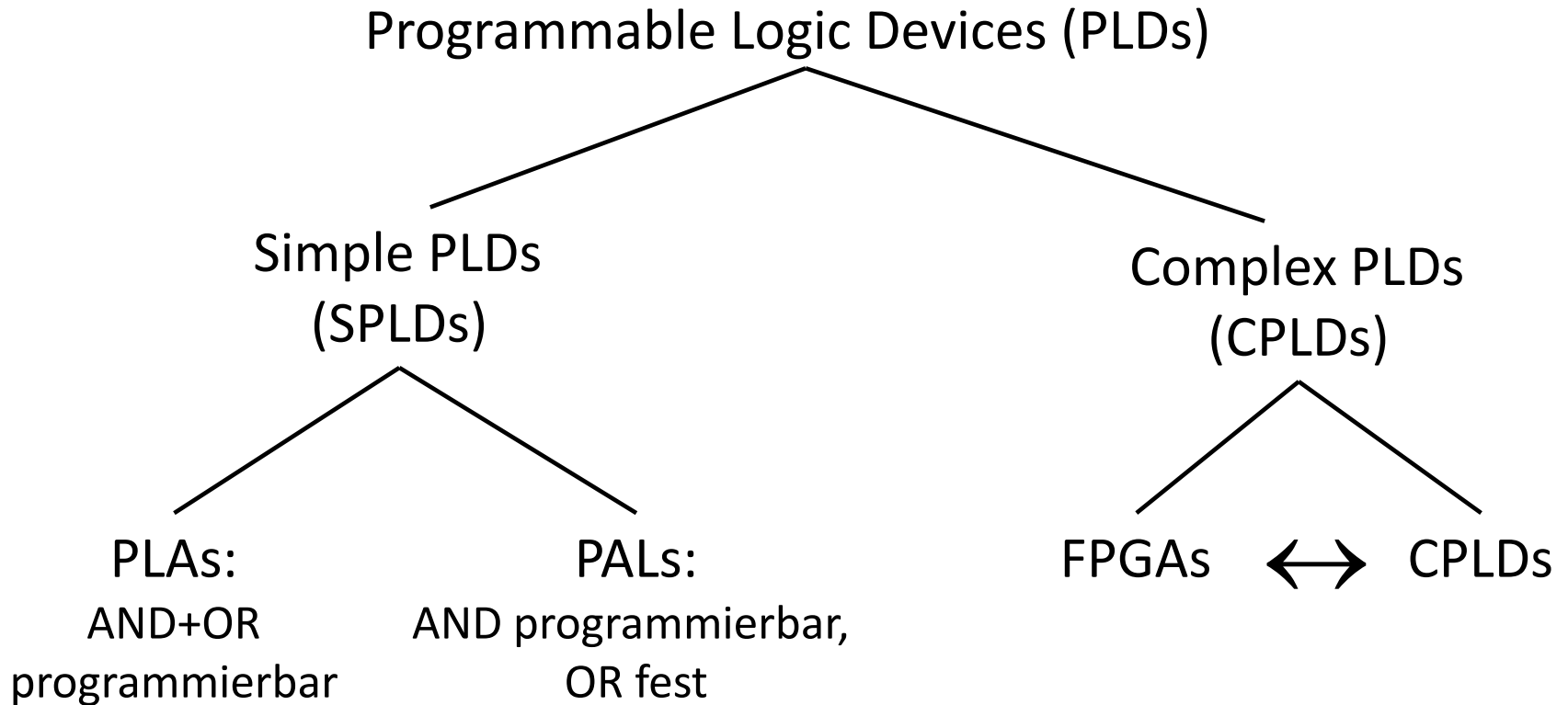




Quelle: static-resources.imageservice.cloud

Historie der Hardwaresynthese

- In den 80er Jahren kamen Standardbausteine zur Realisierung von logischen Schaltungen auf den Markt
- Idee:
 - Eine einheitliche Standardstruktur
 - Realisierung der eigentlichen Schaltung durch „Programmierung“
 - \approx Festlegen der Verbindungen und Verbindungsarten
 - Möglichkeit, Programmierung zu ändern
- Oberbegriff: **Programmable Logic Devices (PLDs)**
- Zunächst sehr einfache Strukturen
- Heute **Complex Programmable Logic Devices (CPLDs)**

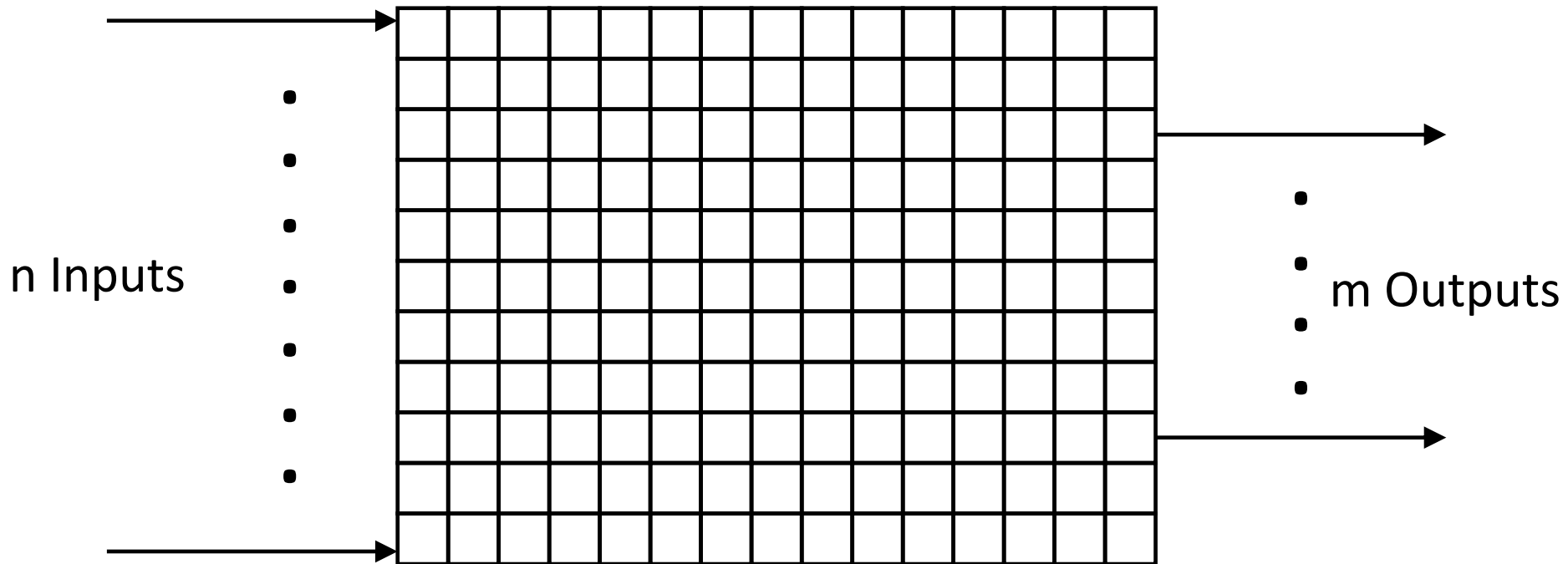


Abschnitt 12.4

Simple Programmable Logic Devices

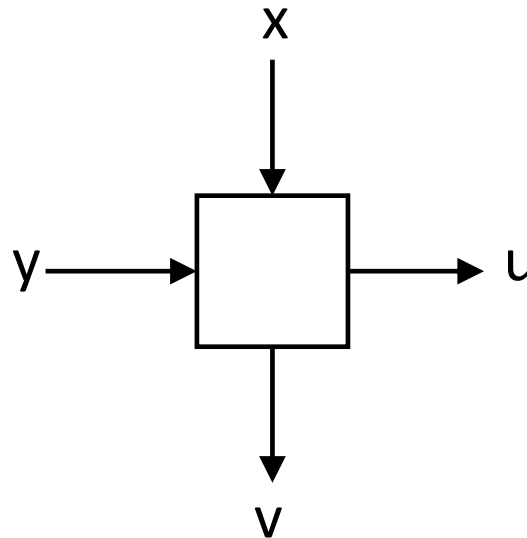
- ▶ Aufbau eines PLAs
- ▶ Bausteintypen
- ▶ Realisierung von Schaltfunktionen durch PLAs
- ▶ Programmierung von PLAs
- ▶ Punkt-orientierte Darstellung von PLAs
- ▶ Faltung von PLAs

Aufbau eines Programmable Logic Array (PLA)



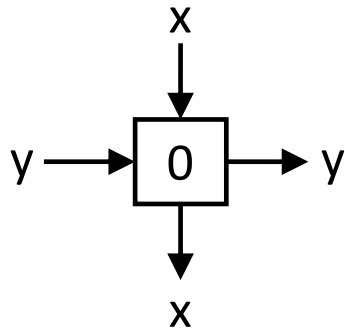
© G. Lakemeyer, W. Oberschelp, G. Vossen

Ein Gitterpunkt

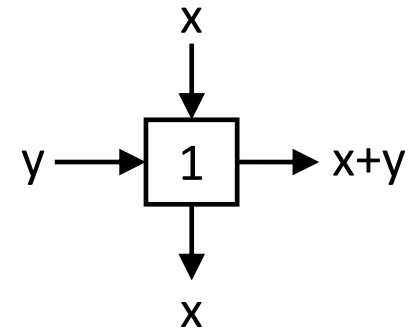


© G. Lakemeyer, W. Oberschelp, G. Vossen

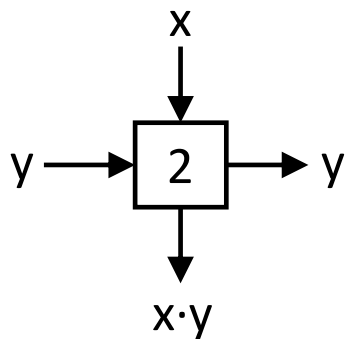
Identer



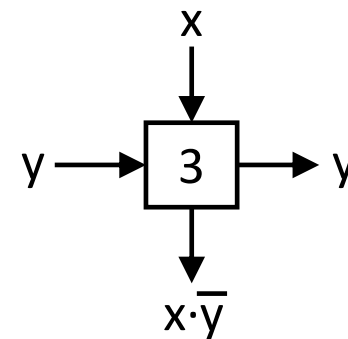
Addierer



Multiplizierer



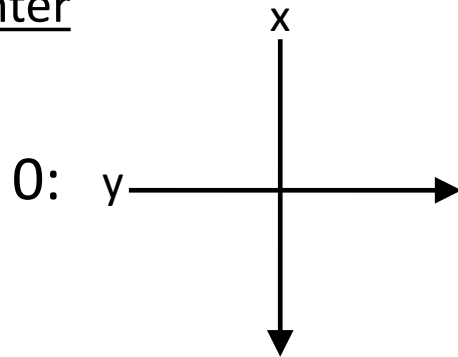
Negat-Multiplizierer



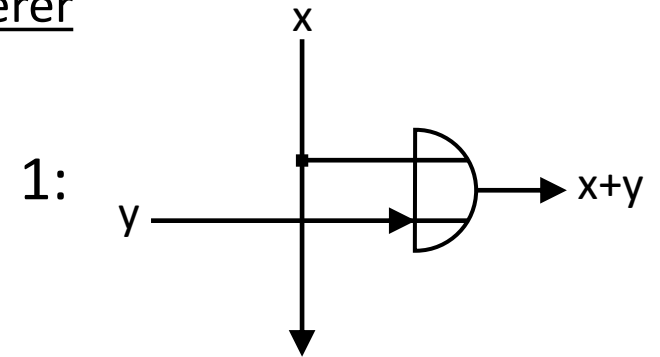
© G. Lakemeyer, W. Oberschelp, G. Vossen

Realisierung der Bausteintypen

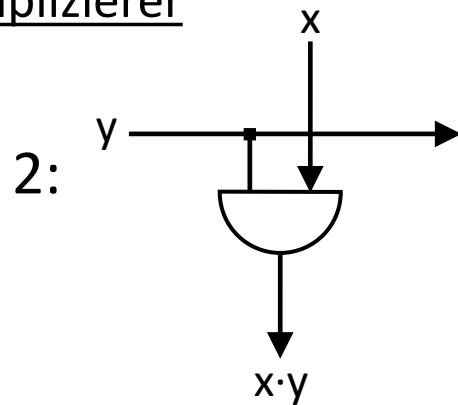
Identer



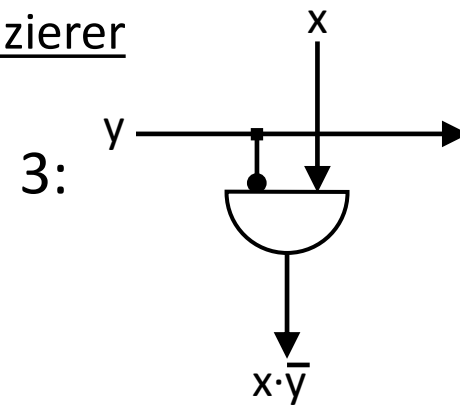
Addierer



Multiplizierer



Negat-Multiplizierer



© G. Lakemeyer, W. Oberschelp, G. Vossen

Beispiel: Realisierung von Schaltfunktionen durch PLAs

Es soll

$F: B^3 \rightarrow B^2$, definiert durch

$$F(x, y, z) := (\underbrace{\bar{y}z + xyz}_u, \underbrace{xz + xy\bar{z}}_v)$$

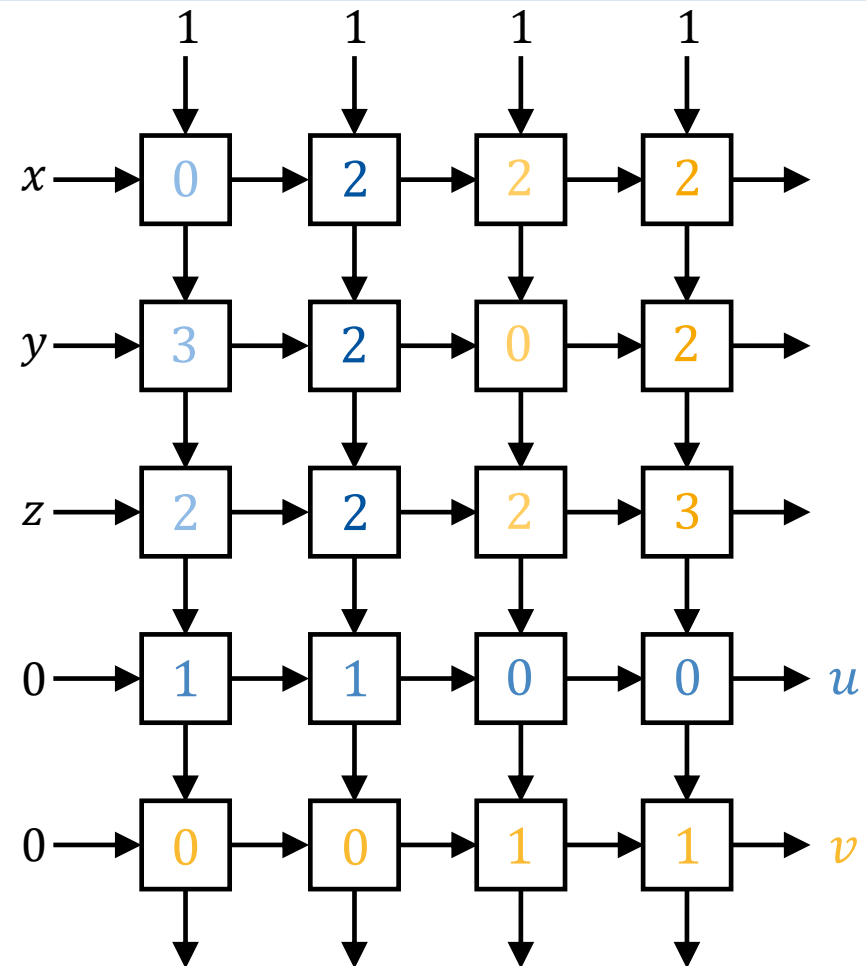
realisiert werden.

© G. Lakemeyer, W. Oberschelp, G. Vossen

Beispiel: Realisierung von Schaltfunktionen durch PLAs

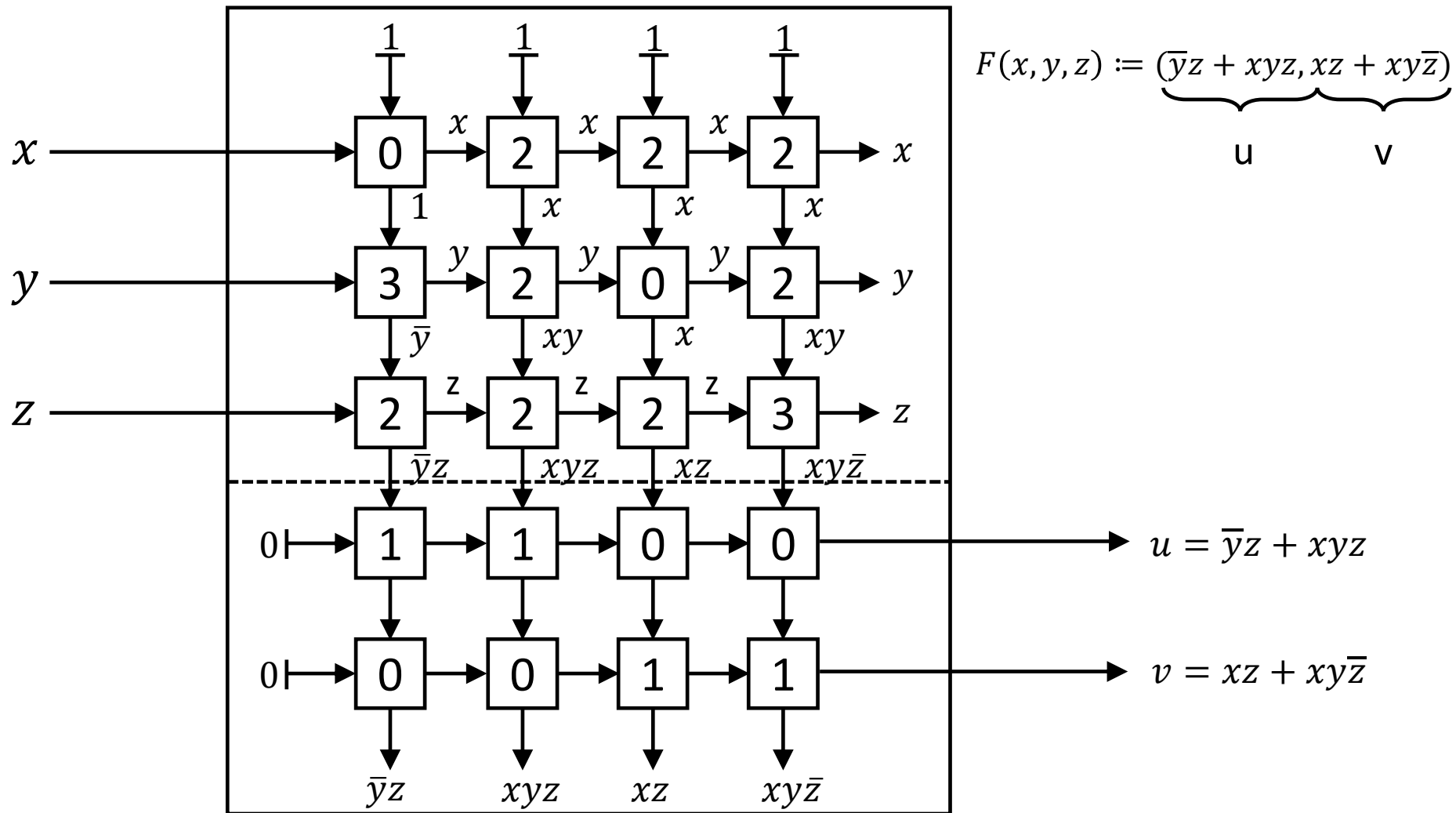
- $n=5$ Inputs an der linken Seite
- $m=5$ Outputs an der rechten Seite
- $k=4$ Spalten
- #Zeilen = #Variablen + #Outputs
- #Spalten = #disjunkten konjunktiv verknüpften Terme

$$F(x, y, z) := \underbrace{(\bar{y}z + xyz)}_u \underbrace{xz + xy\bar{z}}_v$$



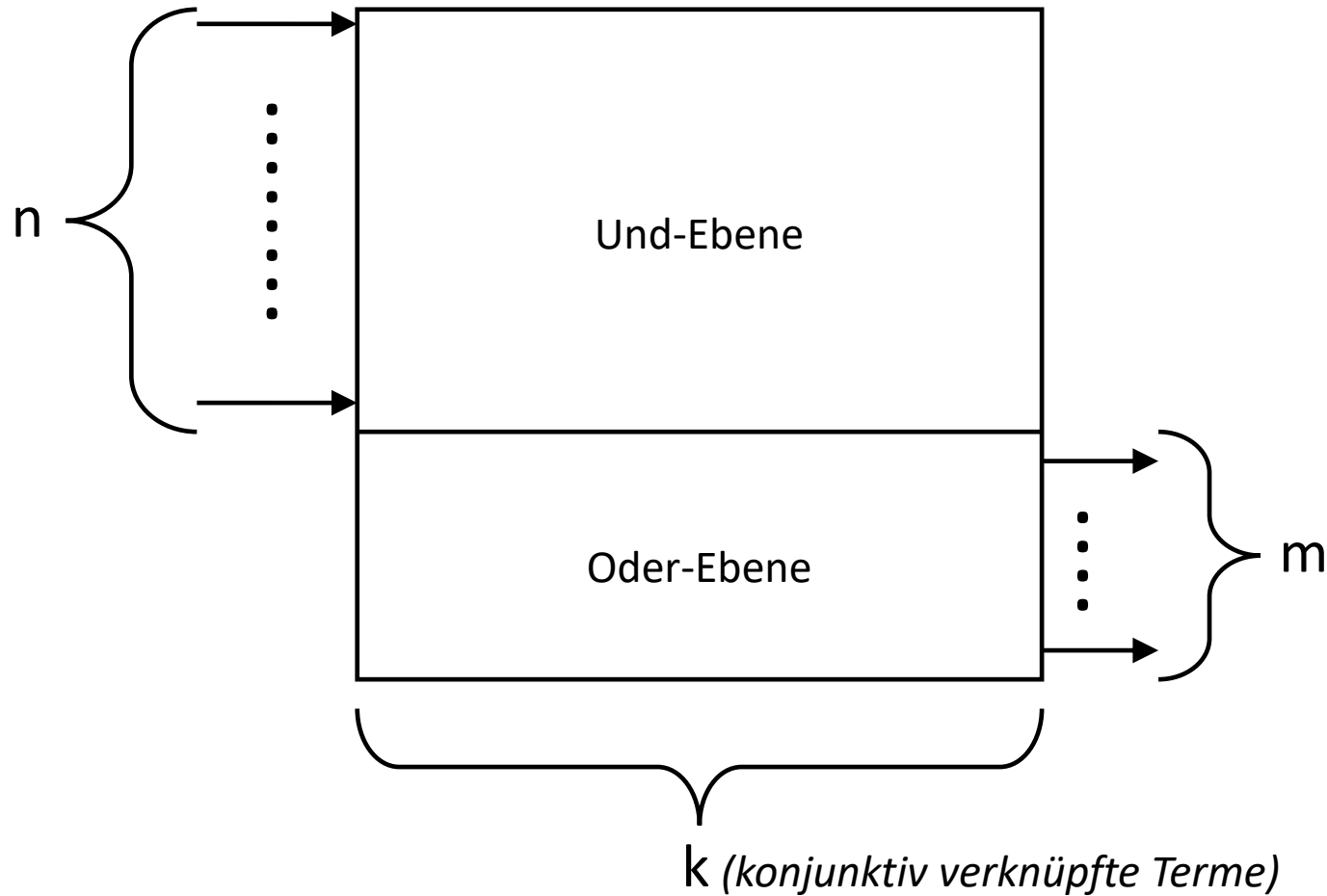
© G. Lakemeyer, W. Oberschelp, G. Vossen

Beispiel: Realisierung von Schaltfunktionen durch PLAs



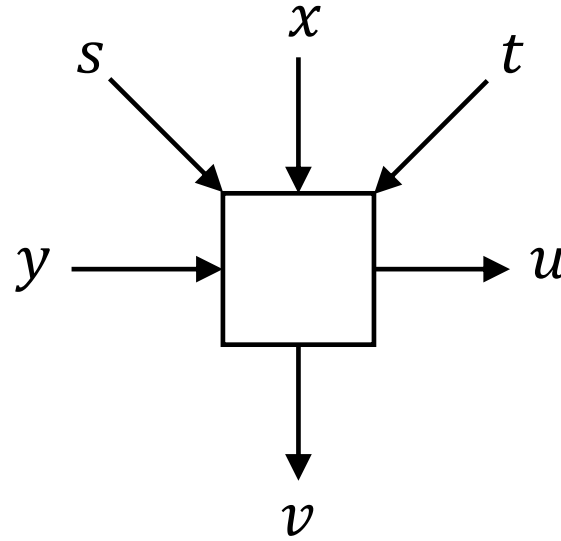
© G. Lakemeyer, W. Oberschelp, G. Vossen

Allgemeiner PLA-Aufbau



© G. Lakemeyer, W. Oberschelp, G. Vossen

Zur Programmierung von PLAs

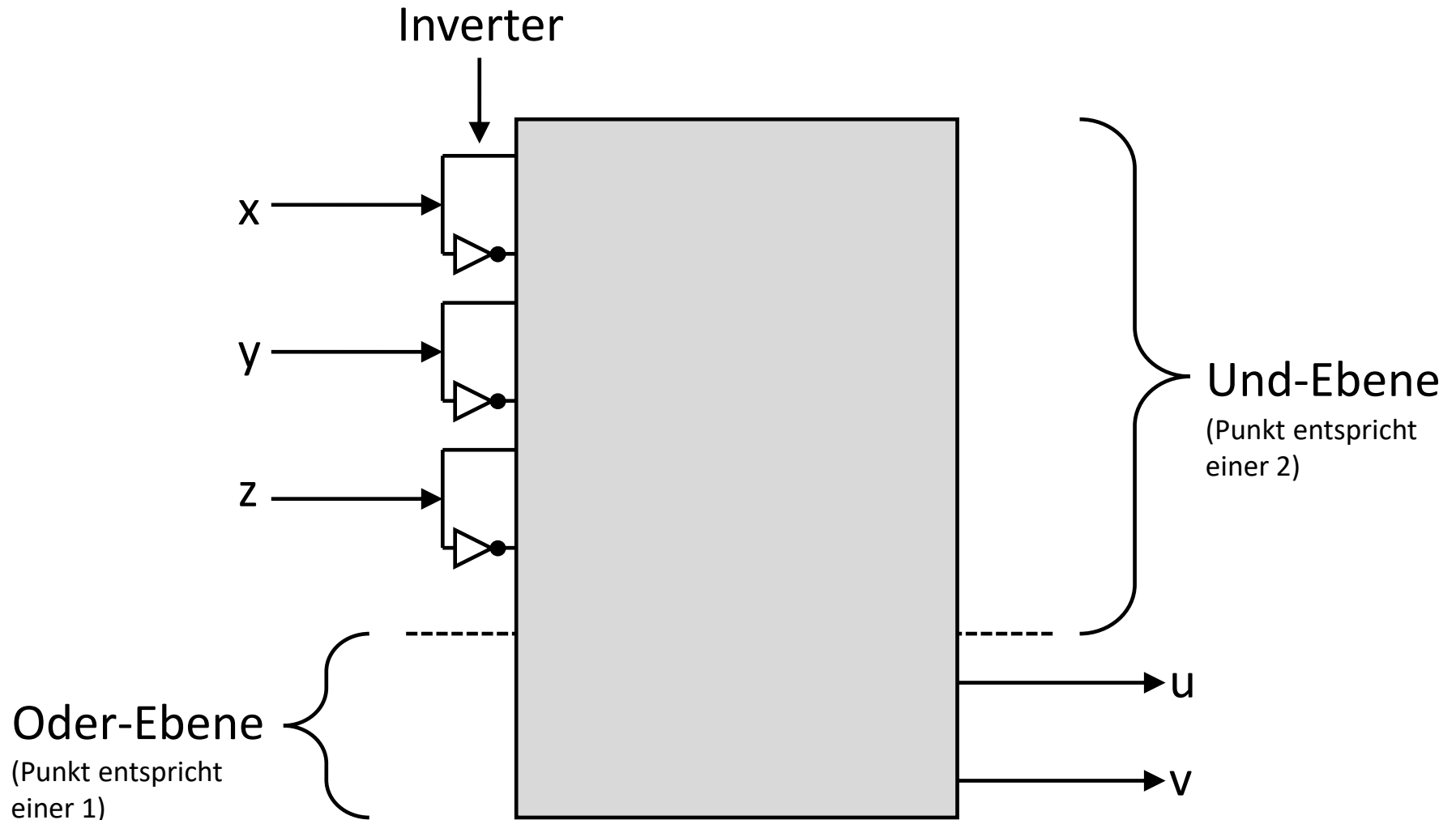


Baustein-Typ	s	t	v	u
0	0	0	x	y
1	0	1	x	$x + y$
2	1	0	$x \cdot y$	y
3	1	1	$x \cdot \bar{y}$	y

Daraus liest man ab: $u = y + \bar{s}tx$, $v = \bar{s}x + sx(t \oplus y)$

© G. Lakemeyer, W. Oberschelp, G. Vossen

Punkt-orientierte PLA-Darstellung

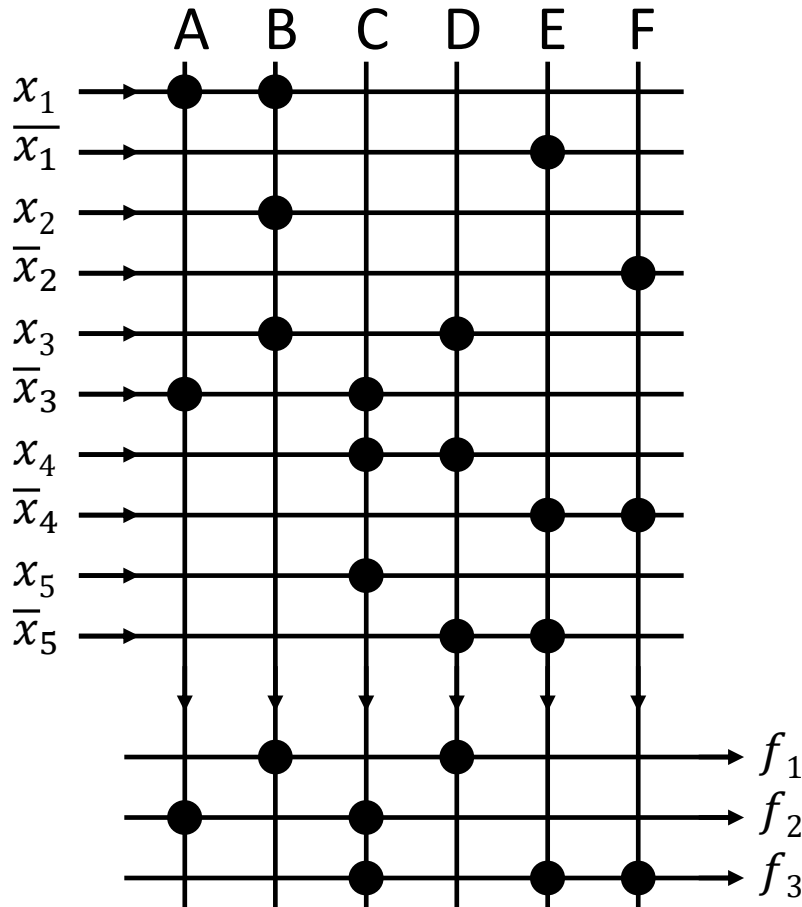


© G. Lakemeyer, W. Oberschelp, G. Vossen

Faltung von PLAs

PLA für eine Funktion $F: B^5 \rightarrow B^3$:

$$F(x_1, x_2, x_3, x_4, x_5) := (x_1 x_2 x_3 + x_3 x_4 \bar{x}_5, \\ x_1 \bar{x}_3 + \bar{x}_3 x_4 x_5, \bar{x}_3 x_4 x_5 + \bar{x}_1 \bar{x}_4 \bar{x}_5 + \bar{x}_2 \bar{x}_4)$$



© G. Lakemeyer, W. Oberschelp, G. Vossen

Überdeckungsmatrix

		A	B	C	D	E	F
1	x_1	1	1				
2	\bar{x}_1					1	
3	x_2		1				
4	\bar{x}_2						1
5	x_3		1		1		
6	\bar{x}_3	1		1			
7	x_4			1	1		
8	\bar{x}_4					1	1
9	x_5			1			
10	\bar{x}_5				1	1	

© G. Lakemeyer, W. Oberschelp, G. Vossen


Überdeckungsmatrix

		A	B	C	D	E	F
1	x_1	1	1				
2	\bar{x}_1					1	
3	x_2		1				
4	\bar{x}_2						1
5	x_3		1		1		
6	\bar{x}_3	1		1			
7	x_4			1	1		
8	\bar{x}_4					1	1
9	x_5			1			
10	\bar{x}_5				1	1	



		A	B	C	D	E	F	
1	x_1	1	1	1	1			x_4
2	\bar{x}_1					1		
3	x_2		1					
4	\bar{x}_2						1	
5	x_3		1		1			
6	\bar{x}_3	1		1				
8	\bar{x}_4					1	1	
9	x_5			1				
10	\bar{x}_5				1	1		

Überdeckungsmatrix

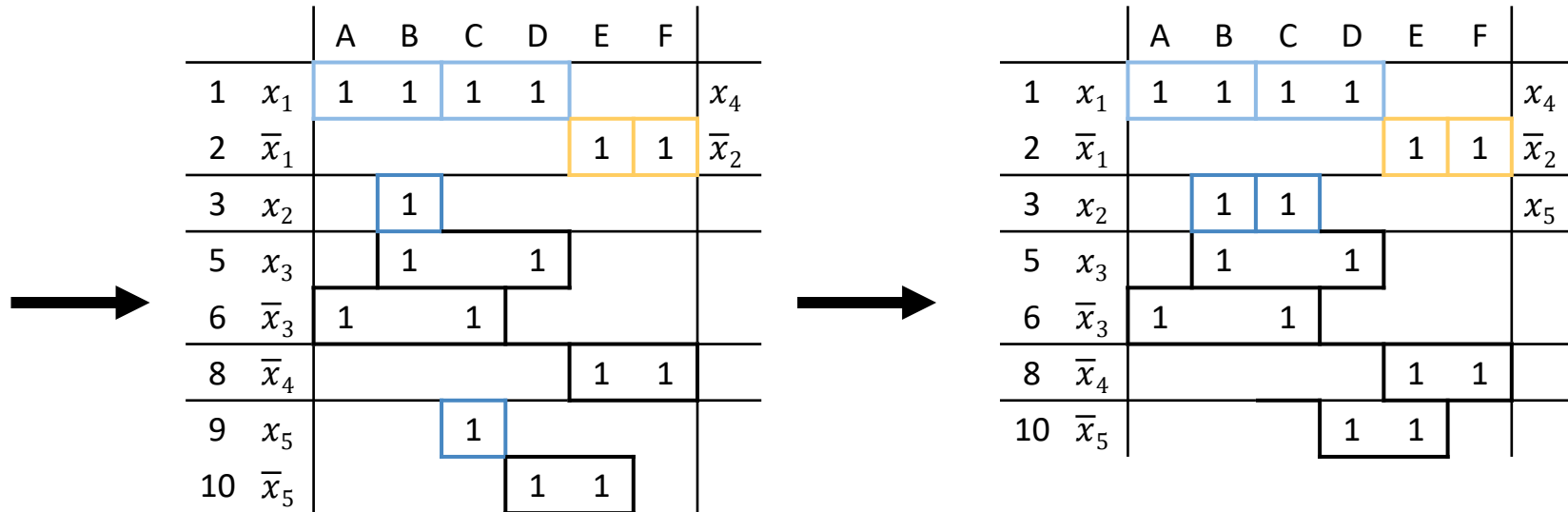


		A	B	C	D	E	F	
1	x_1	1	1	1	1			x_4
2	\bar{x}_1					1		
3	x_2		1					
4	\bar{x}_2						1	
5	x_3		1		1			
6	\bar{x}_3	1		1				
8	\bar{x}_4					1	1	
9	x_5			1				
10	\bar{x}_5				1	1		



		A	B	C	D	E	F	
1	x_1	1	1	1	1			x_4
2	\bar{x}_1					1	1	\bar{x}_2
3	x_2		1					
5	x_3		1		1			
6	\bar{x}_3	1		1				
8	\bar{x}_4					1	1	
9	x_5			1				
10	\bar{x}_5				1	1		

Überdeckungsmatrix



Überdeckungsmatrix

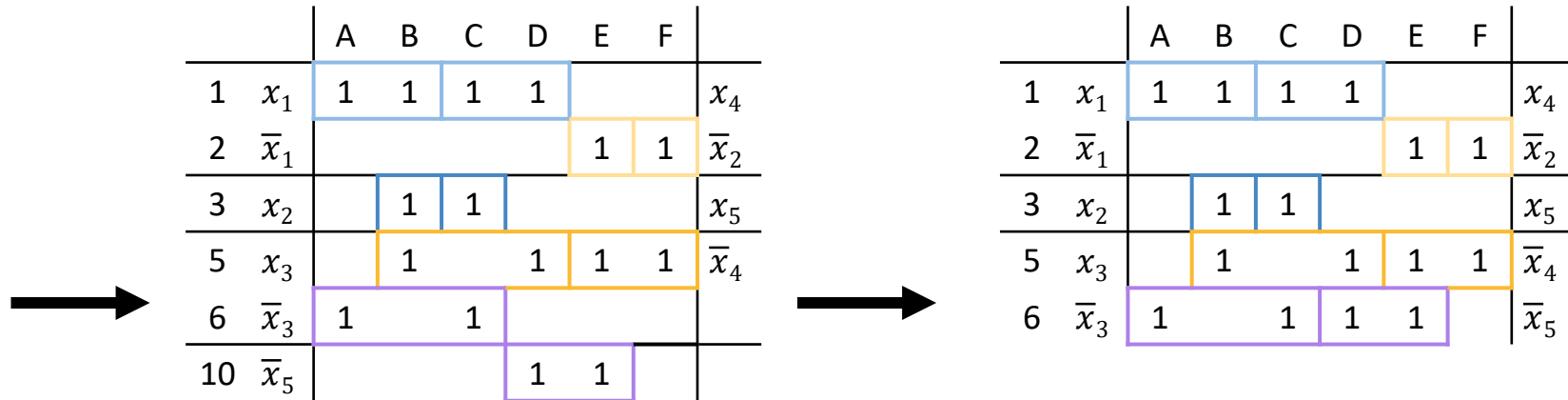


		A	B	C	D	E	F	
1	x_1	1	1	1	1			x_4
2	\bar{x}_1					1	1	\bar{x}_2
3	x_2		1	1				x_5
5	x_3		1		1			
6	\bar{x}_3	1		1				
8	\bar{x}_4					1	1	
10	\bar{x}_5				1	1		

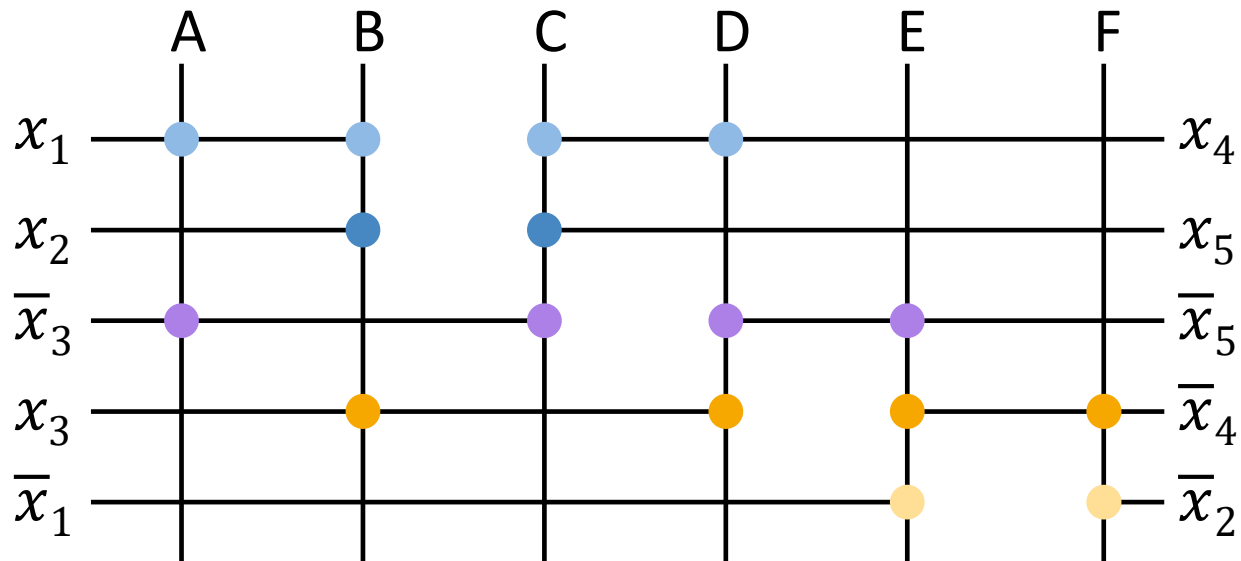


		A	B	C	D	E	F	
1	x_1	1	1	1	1			x_4
2	\bar{x}_1					1	1	\bar{x}_2
3	x_2		1	1				x_5
5	x_3		1		1	1	1	\bar{x}_4
6	\bar{x}_3	1		1				
10	\bar{x}_5				1	1		

Überdeckungsmatrix

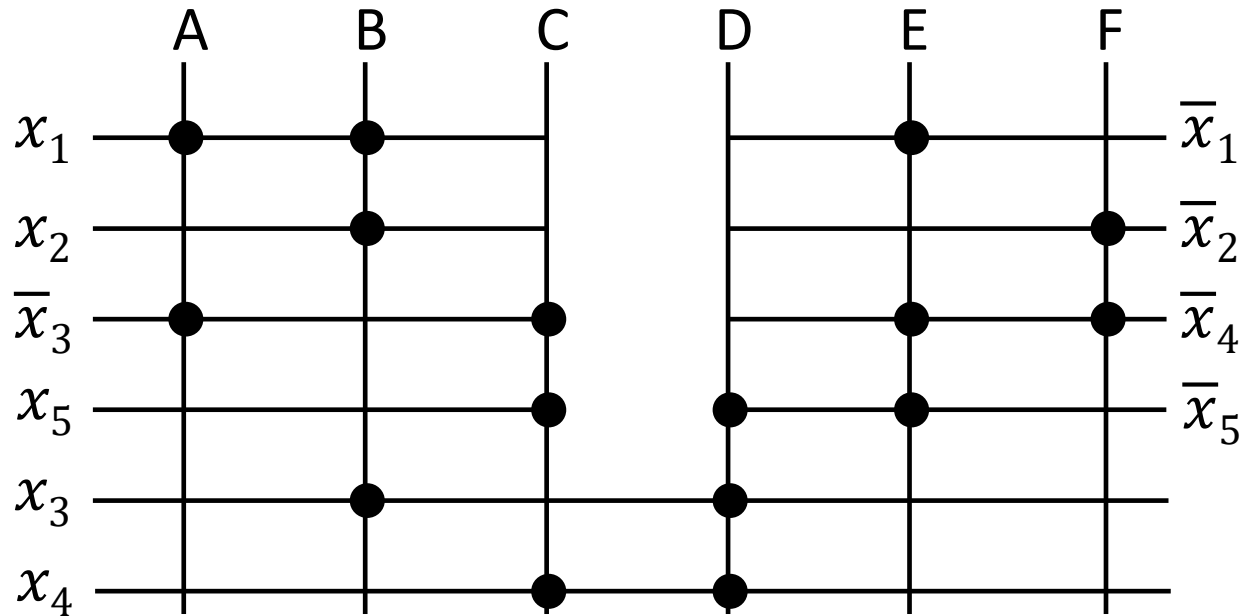


Faltung der UND-Ebene



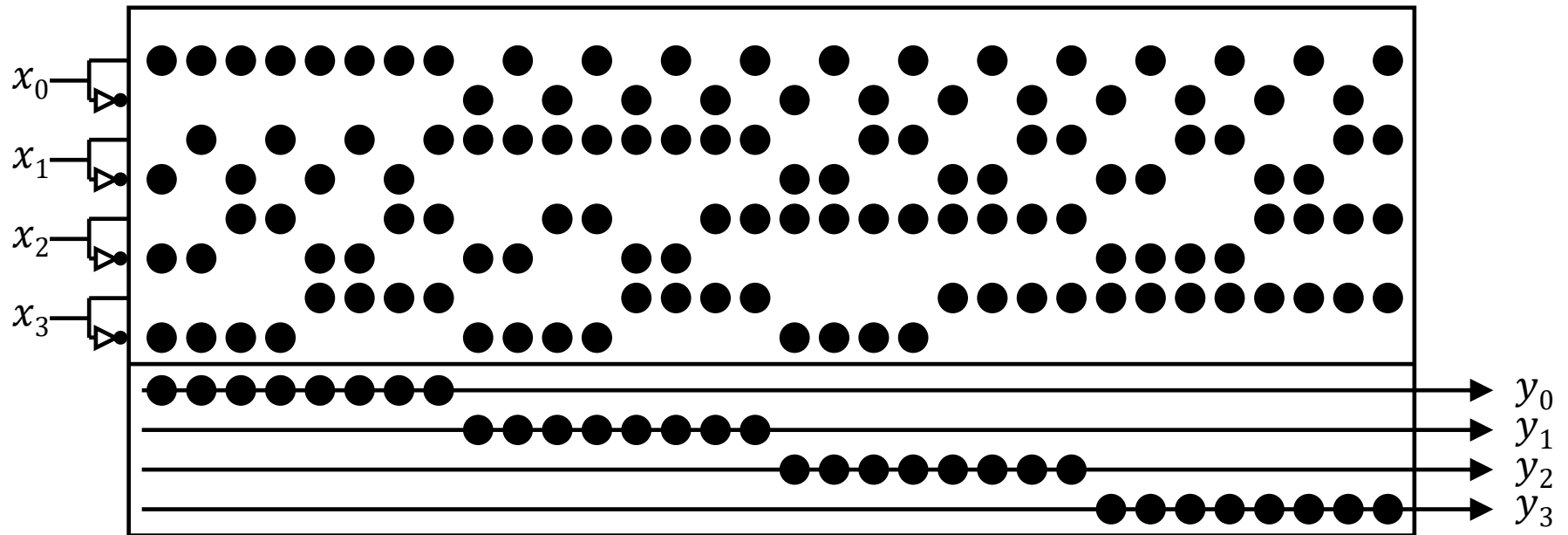
© G. Lakemeyer, W. Oberschelp, G. Vossen

Block-Faltung



© G. Lakemeyer, W. Oberschelp, G. Vossen

Programmable Array Logic (PAL)



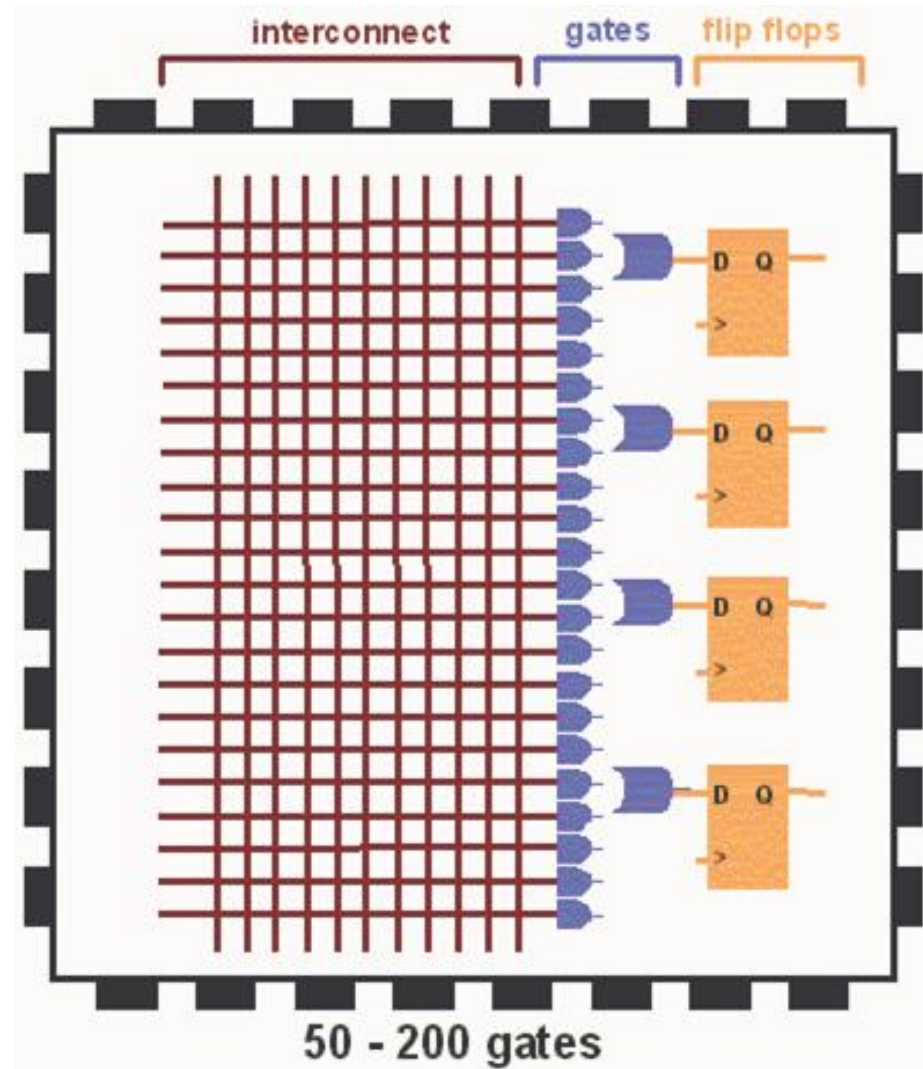
© G. Lakemeyer, W. Oberschelp, G. Vossen

Abschnitt 12.5

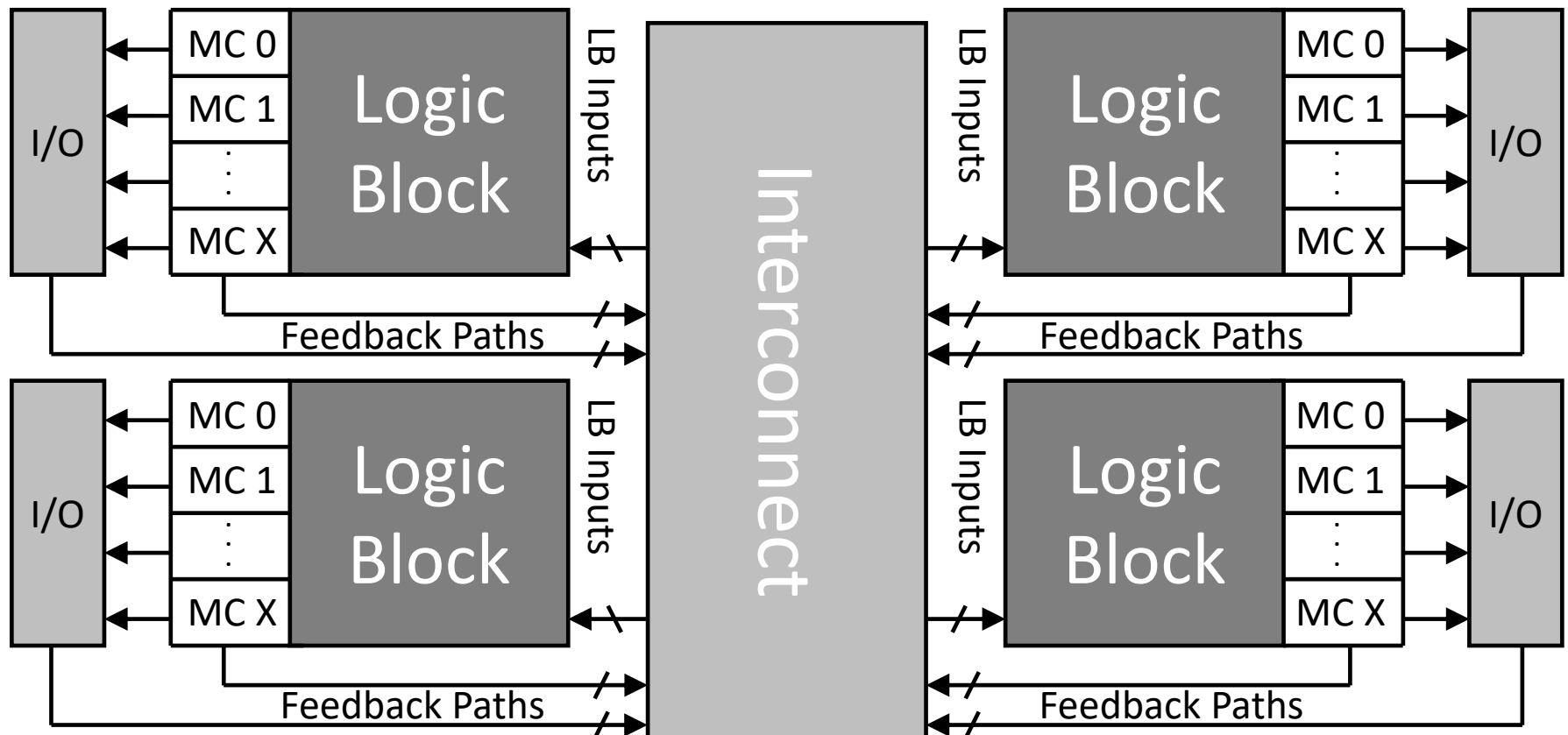
Complex Programmable Logic Devices

- ▶ CPLDs
- ▶ Prinzipielle FPGA-Struktur
- ▶ Spartan-FPGAs

- Zentrale Verbindungsmatrix
- Einfaches Routing
- Einfaches, deterministisches Zeitverhalten
- Tools müssen nur Verbindungspunkte setzen

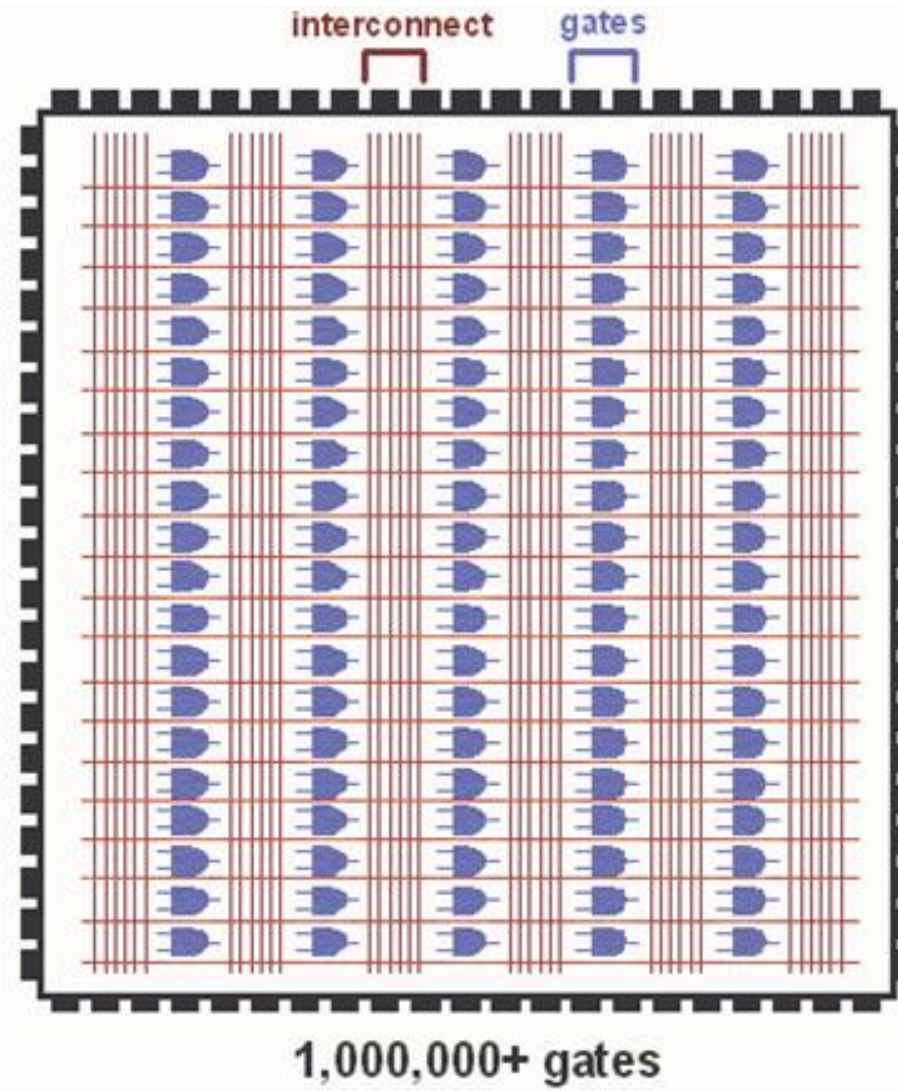


Prinzipschaltbild eines CPLDs

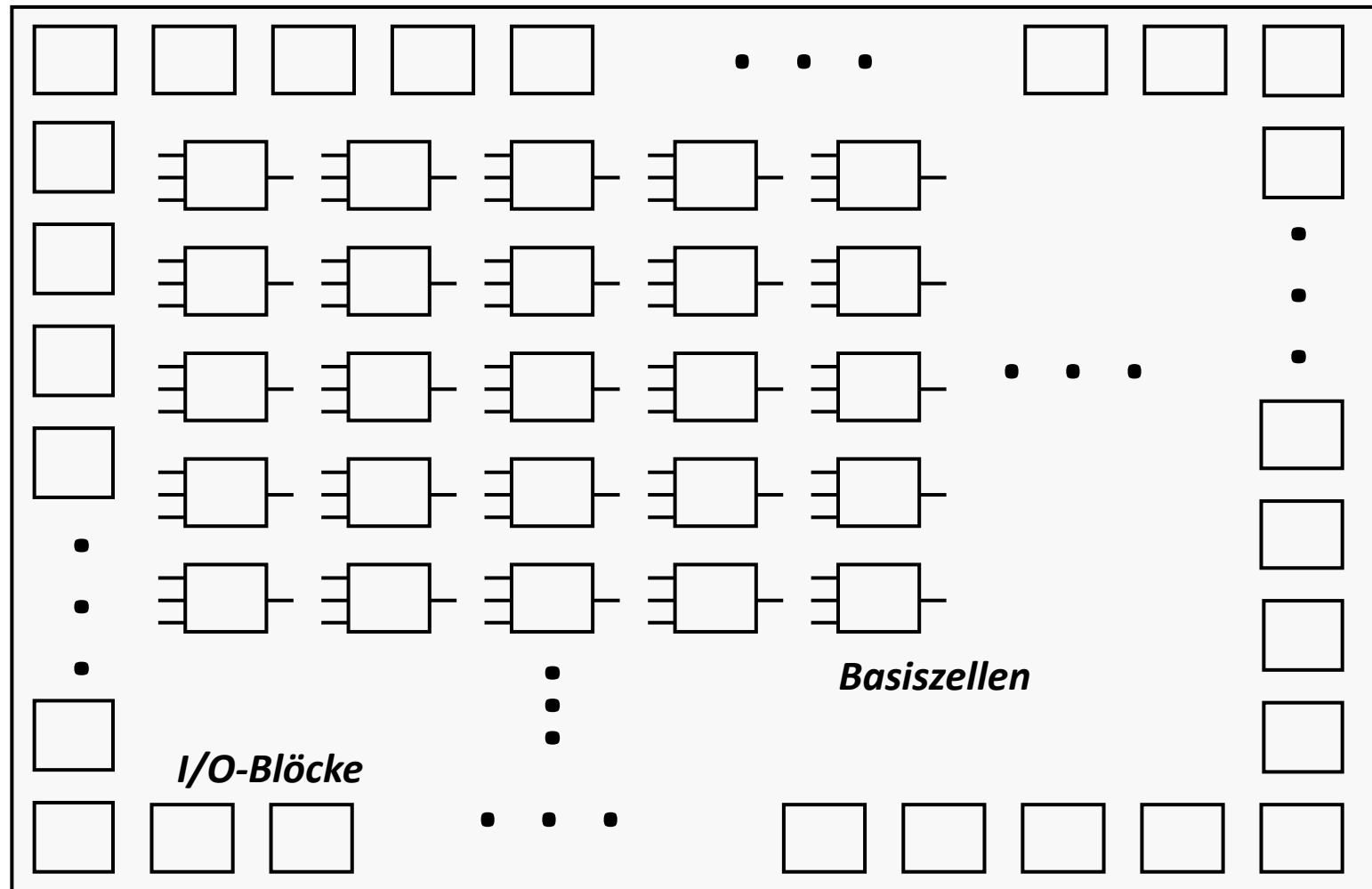


© W. Oberschelp, G. Vossen

- Kanalbasierte Verbindungen
- Komplexes Routing
- Zeitverhalten erst nach Design bekannt
- Tools müssen komplexe Synthesen durchführen

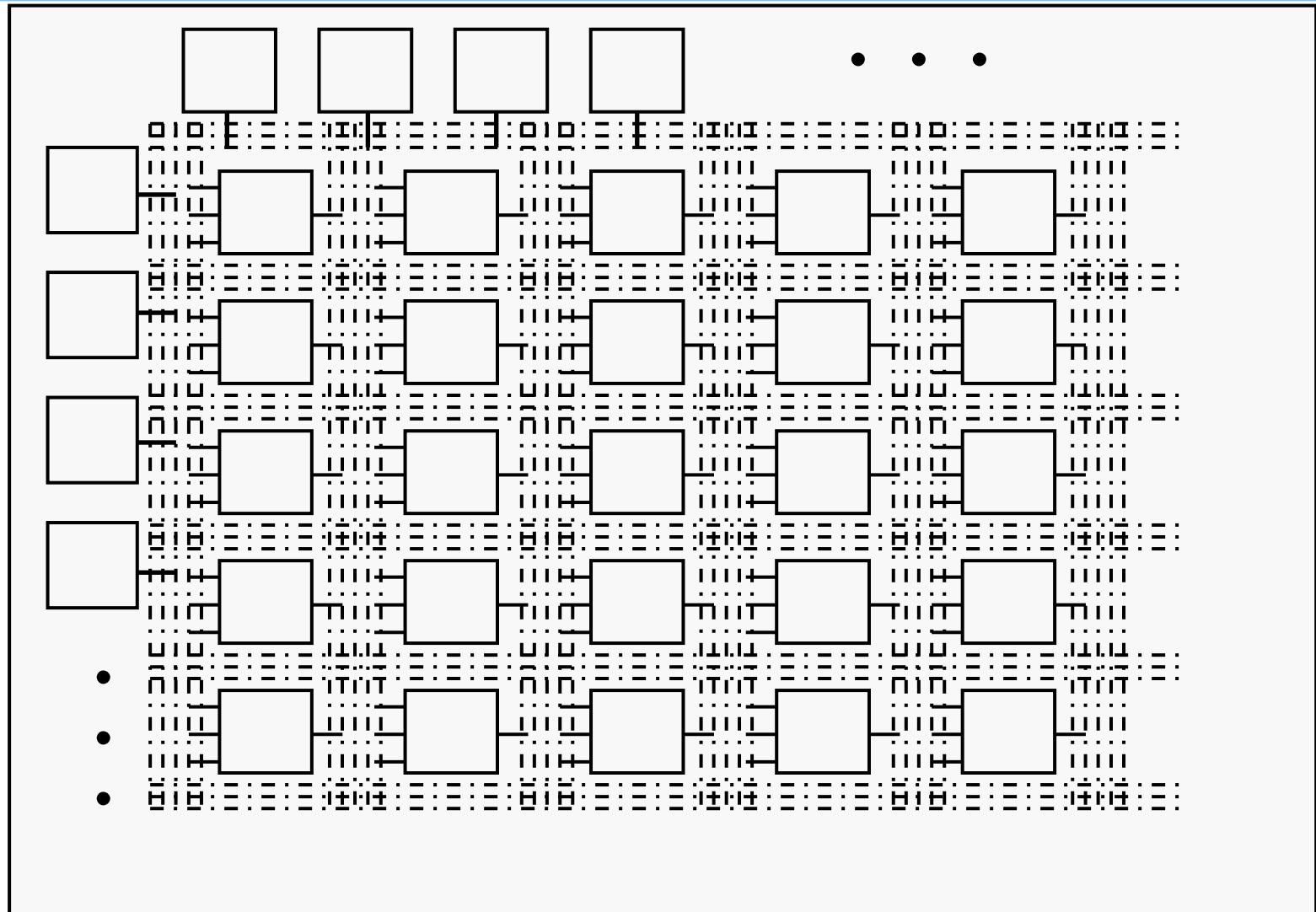


Prinzipielle FPGA-Struktur



© W. Oberschelp, G. Vossen

Prinzip der FPGA-Verbindungsstruktur

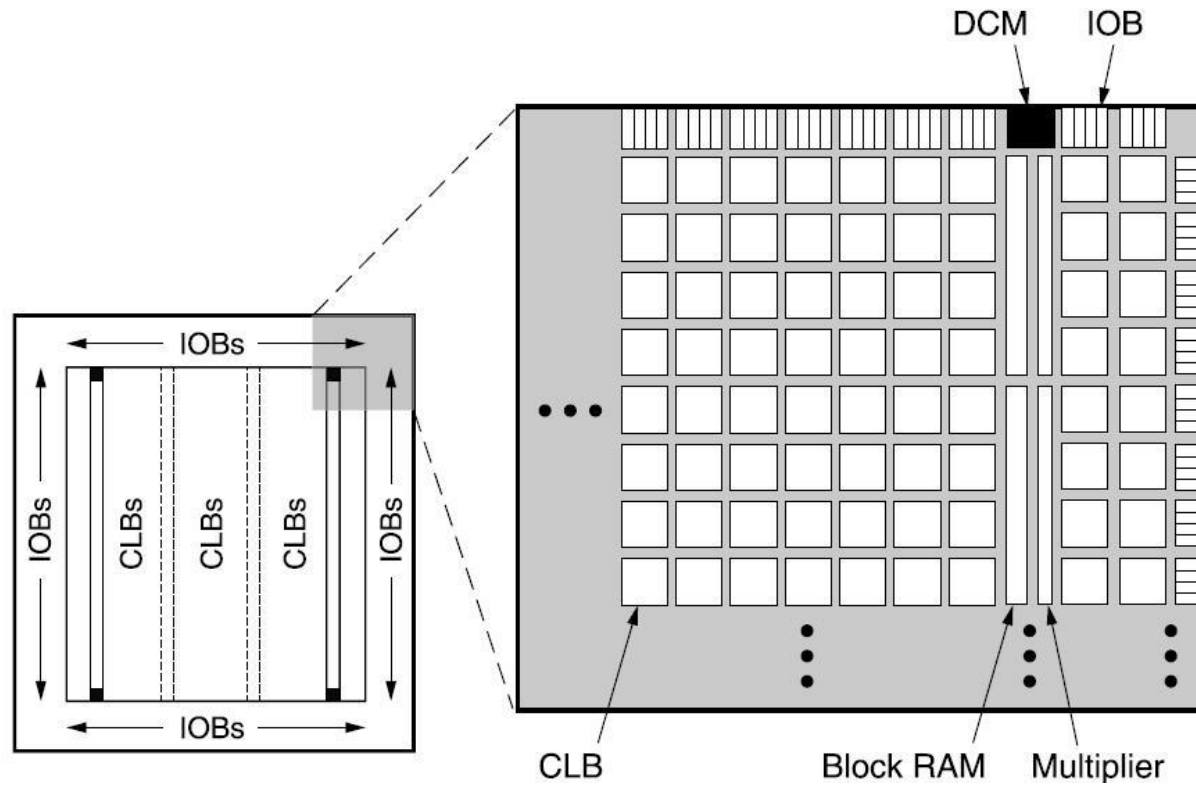


© W. Oberschelp, G. Vossen

Spartan-3 XC3S200FT256 FPGA

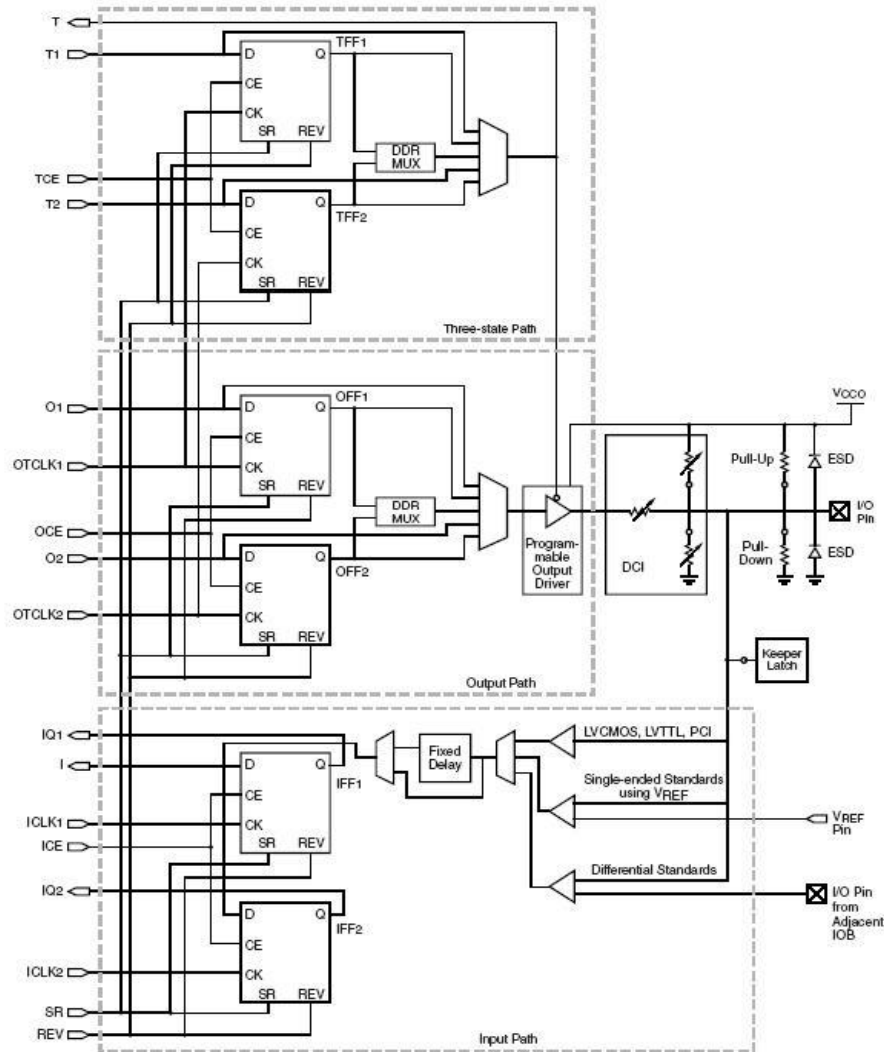
- Taktfrequenz von bis zu 165MHz
- Bis zu 173 benutzerdefinierte Ein-/Ausgabesignale
- 1,2V Kern
- Spannungsregelbare Ein-/Ausgabeoperation: 1,2 - 3,3V
- 4320 gleichwertige logische Zellen
- Interne Multiplizierer und Multiplexer
- Speicherung der Konfiguration in SRAM (flüchtig)
 - Kann auch in externem Flash-Speicher abgelegt werden
- Weitere Informationen: www.xilinx.com

Spartan-3 XC3S200FT256 Architektur



Spartan-3 XC3S200FT256 Architektur

■ I/O Block:



- Complex Logic Block:
LUT (Lookup Table)



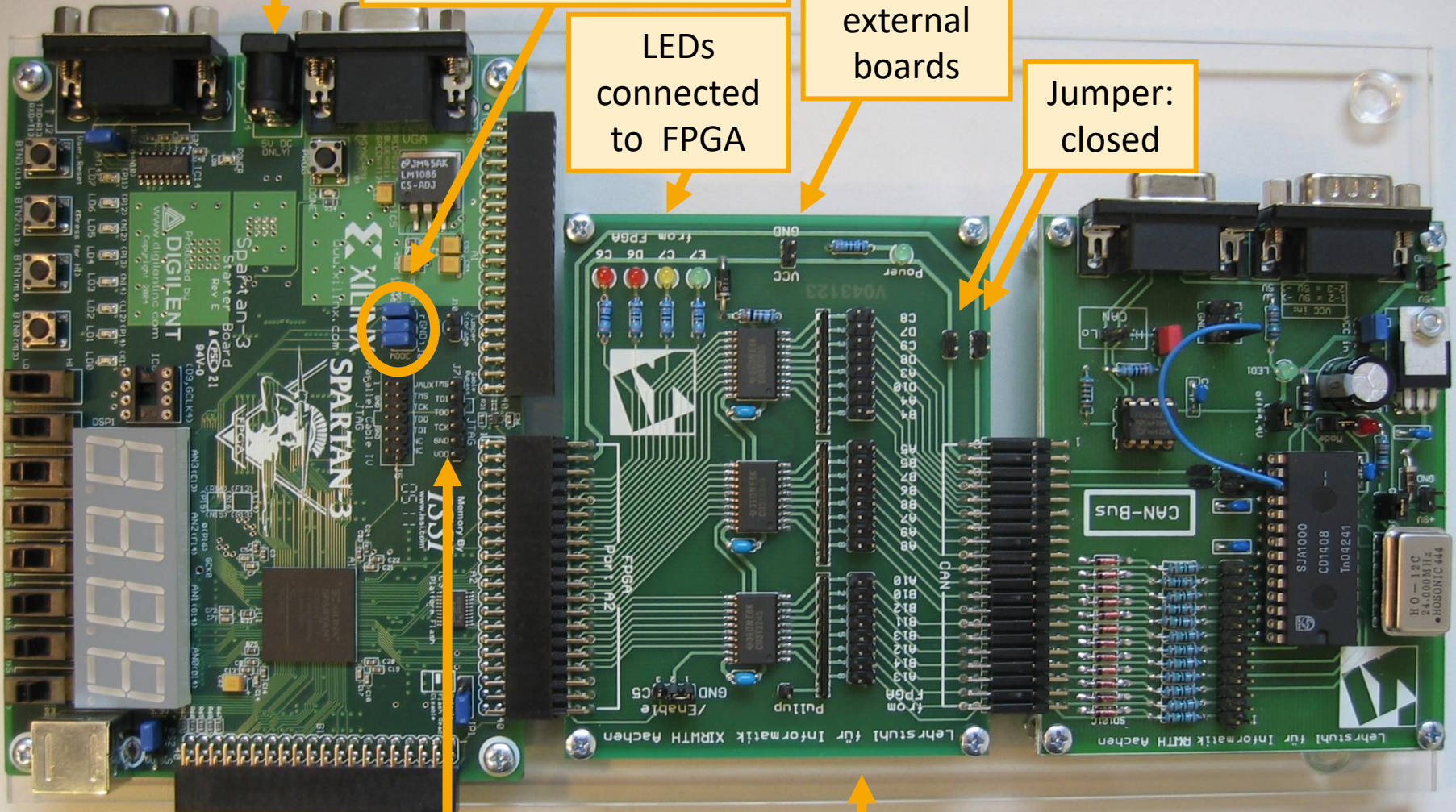
6 V

Jumper: M0&M2: offen,
M1: geschlossen

Power
supply for
external
boards

LEDs
connected
to FPGA

Jumper:
closed



Programming cable to
parallel port

Access to FPGA-Pins

CAN board

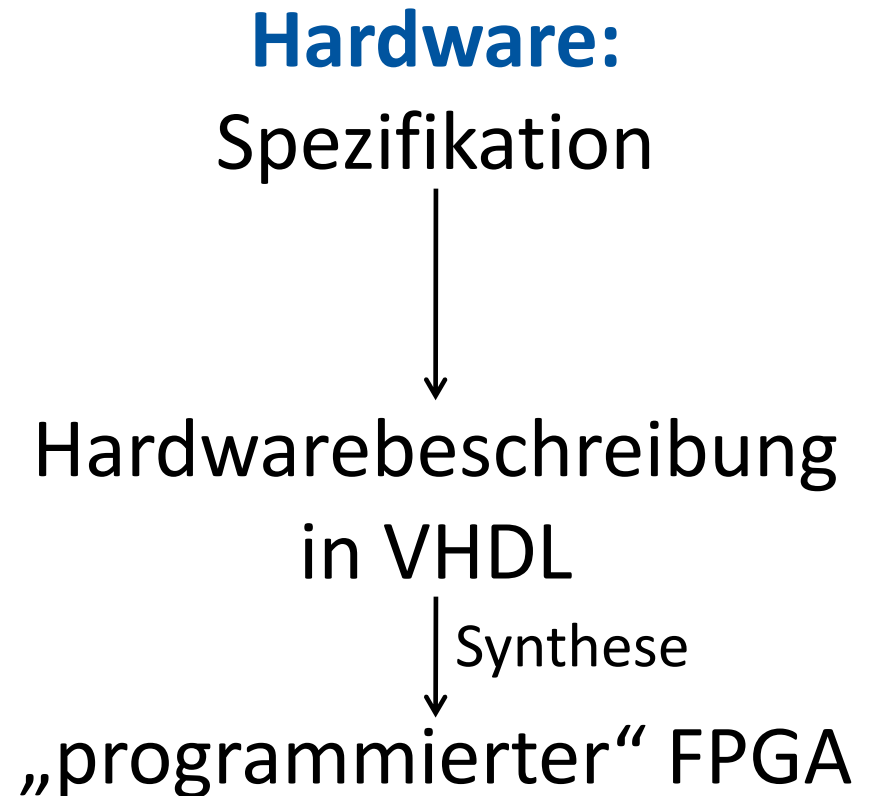
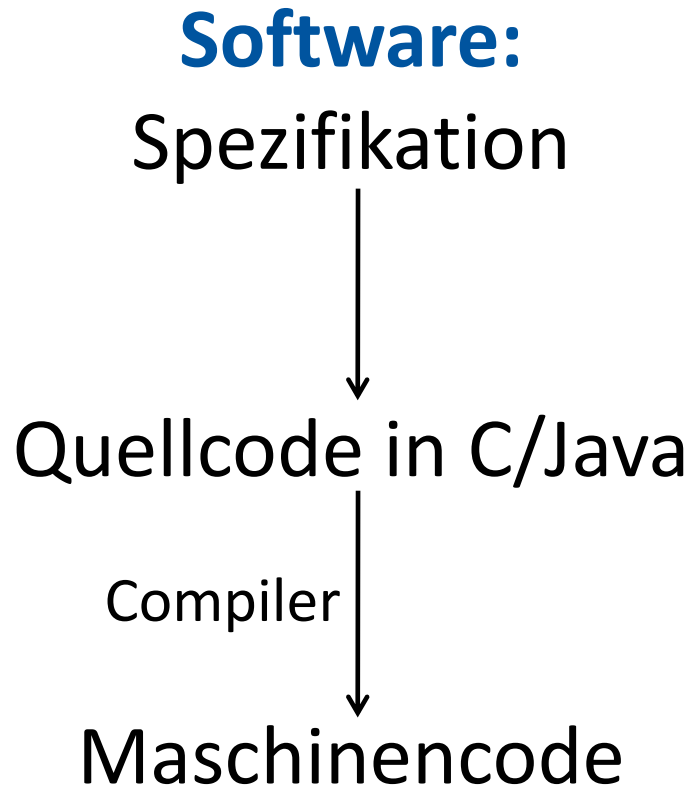
Abschnitt 12.7

VHSIC Hardware Description Language

- ▶ Motivation
- ▶ Crashkurs
- ▶ Komplexes Beispiel

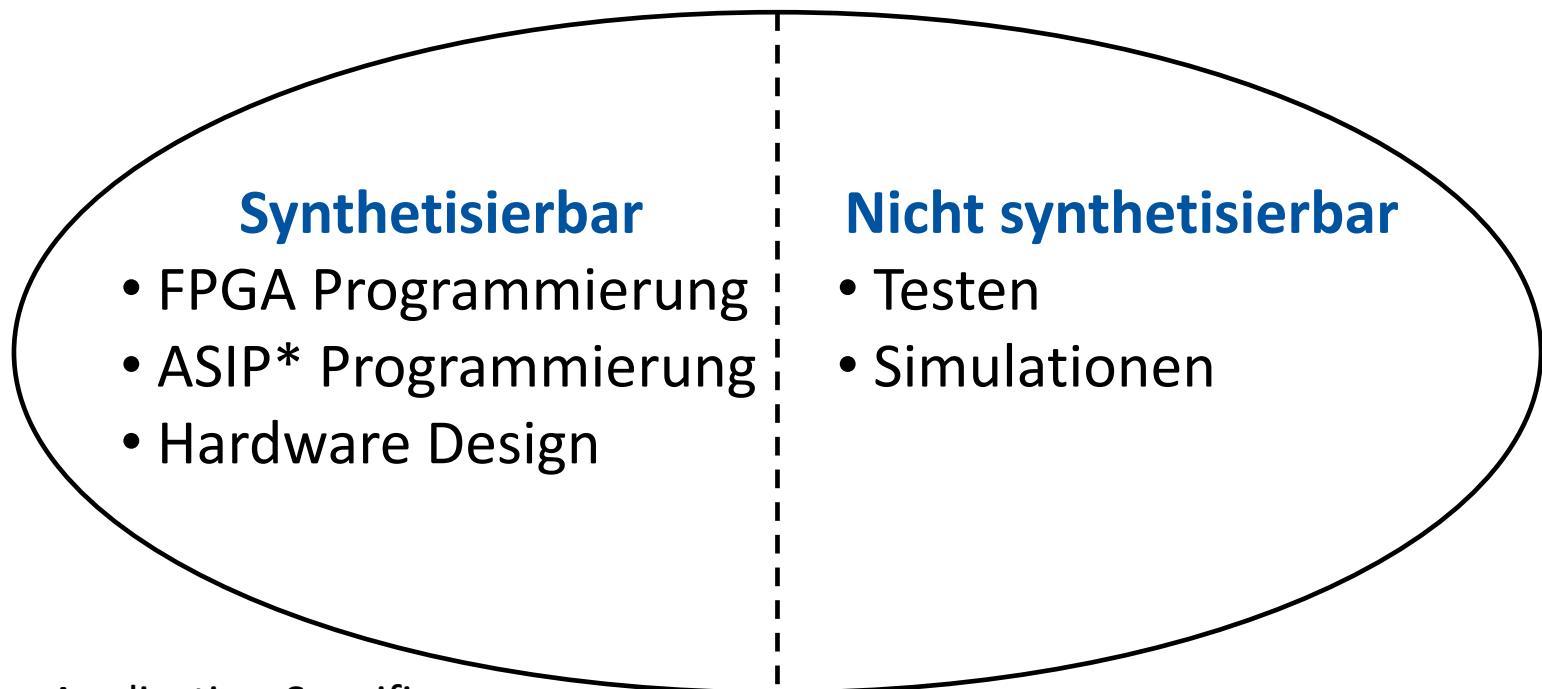
- Schematics nicht geeignet für CPLDs und FPGAs
- Low-Level Beschreibung der Programmierung (einzelne Verbindungen) auch nicht angemessen
- → Hardwarebeschreibungssprachen

Prinzip: Hardware wie Software entwickeln



- vor 1987: > 100 HDLs
- heute vor allem zwei HDL:
 - **VHDL**
 - an Ada angelehnt
 - seit 1987 standardisiert
 - **Verilog**
 - an C angelehnt
 - seit 1997 standardisiert
- Trend: stärkere Orientierung an C, z.B. **System-C**

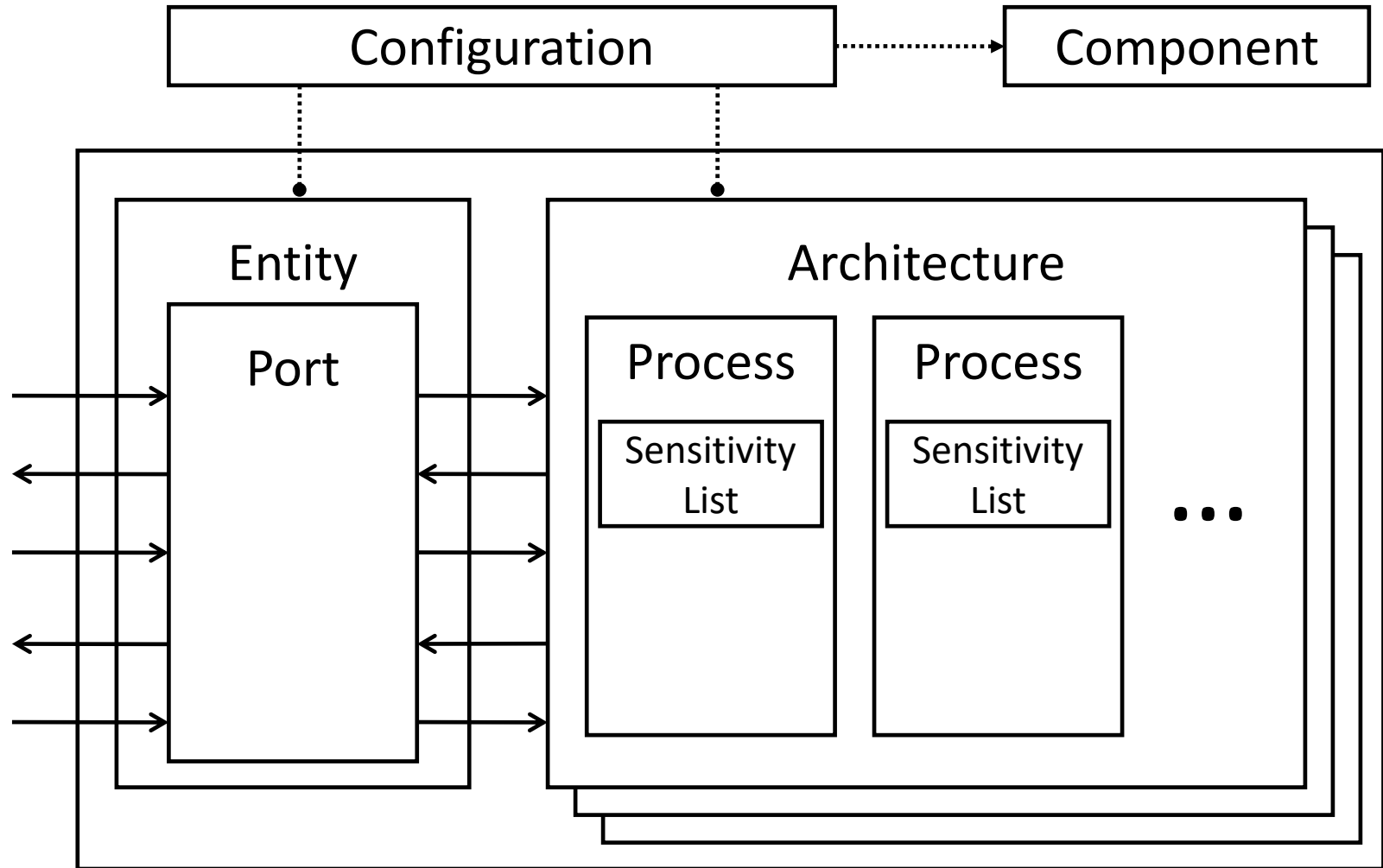
- **V**HSIC **H**ardware **D**escription **L**anguage
- VHSIC = Very High Speed Integrated Circuits



*ASIP = Application-Specific
Instruction Set Processor

- Basiselemente:
 - **Entity**: Black-Box Interface
 - **Architecture**: Implementierung
 - **Configuration**: Zuordnung Entity → Architecture
 - **Package**: globale Konstanten, Hilfsfunktionen, etc.
- Hierarchischer Aufbau
 - Top-down Entwurfsprozess
 - Platine → Bauteil → Modul → ALU → Gatter → Transistor
- Verschiedene Modelle
 - Mehrere Architectures pro Entity
 - Bindung durch Configurations

Struktur einer VHDL-Beschreibung



Bestandteile einer Architecture

- **Process**

- Laufen parallel
- **Kommunizieren über synchronisierte Signale**
- Beobachten Liste von Signalen (Sensitivity List)

- **Signal**

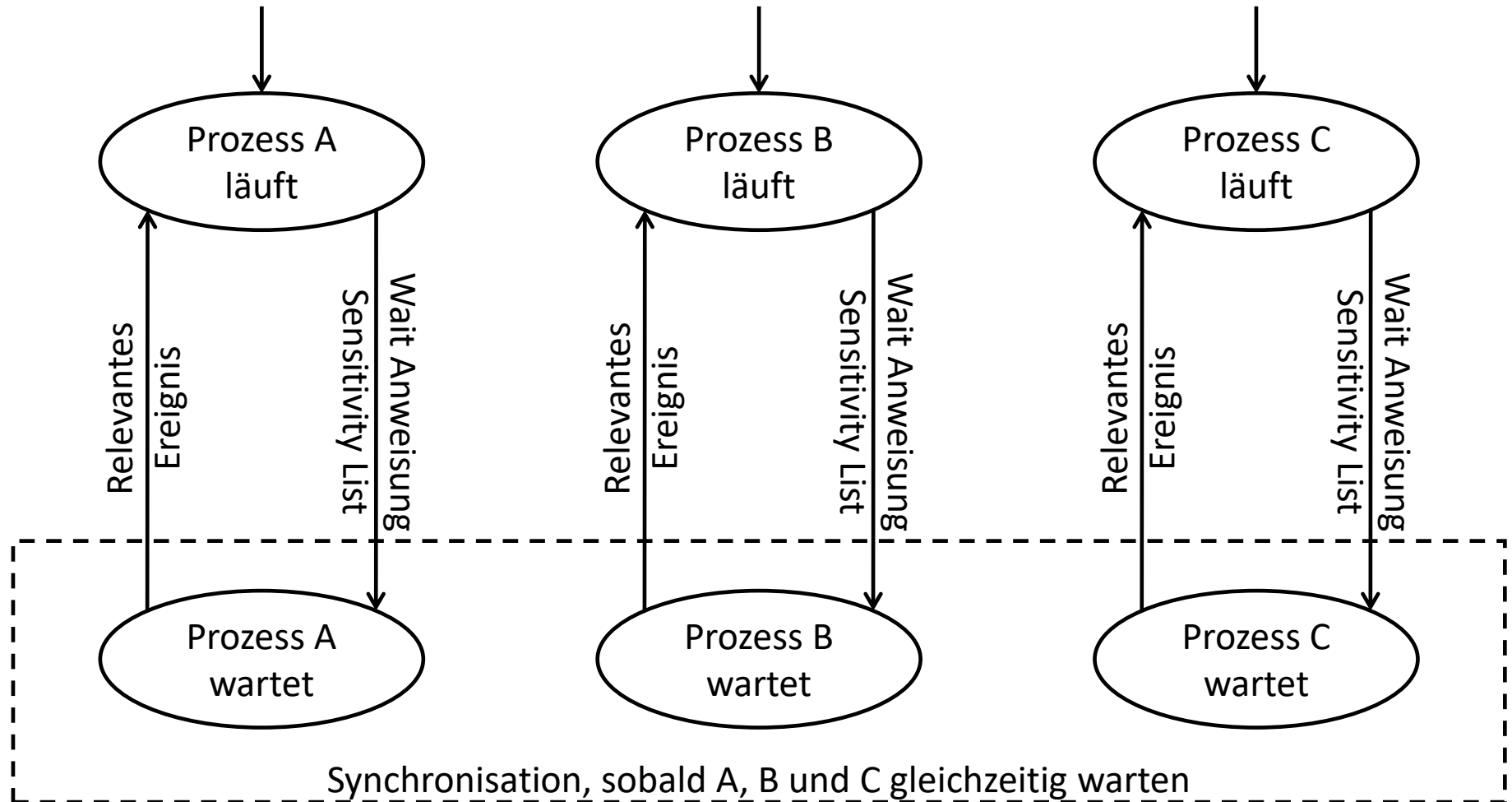
- Architecture intern und synchronisiert
- Mapping nach außen über Port-Deklaration
- **Nur ein Treiber pro Signal (d.h. nur ein Prozess darf das Signal ändern)**

- **Variablen**

- **Process intern**
- Nicht synchronisiert

- Alle Prozesse einer Architecture laufen parallel
- Prozesse arbeiten Aufgabe ab und warten
- Warten wird durch Ereignis beendet
 - Sensitivity List
 - Zeitschranke
 - Beliebig komplexe Ausdrücke
- Wenn alle Prozesse warten, werden Signaländerungen übernommen

Beispiel



- Standard: boolean, integer, char, string, real, ...
- Aufzählungstypen: (rot, gelb, gruen)
- Subtypen: natural (0...n)
- Komplex: array, record, file
- Zeiger: access
- Physikalisch: Zeit (z.B. 2 ns)
- Technische: std_logic, std_logic_vector
0 oder 1 z.B. (1,0,0)

- Definiert in IEEE 1164

```
library ieee;
```

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

- Technisches Signal

- U noch nicht initialisiert
- X treibend unbekannt
- 0 treibend logische 0
- 1 treibend logische 1
- Z hochohmig (für Busse mit Three-State)
- W schwach unbekannt
- L schwach logische 0
- H schwach logische 1
- - egal (für Logiksynthese)

Deklarationen

```
constant SIZE: integer := 5;
```

```
variable U,V: std_logic;
```

```
variable W: std_logic := '1';
```

```
variable I: integer 0 to 255 := 0;
```

```
signal A: std_logic_vector(3 downto 0) := "0000";
```

```
-- Initialisierung mit :=
```

```
-- Zuweisung (s.u.) mit <=
```


Operatoren

- **Logisch:** `and or not nand nor xor xnor`
- **Relational:** `= /= < <= > >=`

*shift left logically** *right* *arithmetically** *rotate left*

The diagram shows six shift operators: `sll`, `srl`, `sla`, `sra`, `rol`, and `ror`. Above them are four labels: *shift left logically**, *right*, *arithmetically**, and *rotate left*. Dotted arrows point from the operators to these labels: `sll` points to *shift left logically**, `srl` points to *right*, `sla` points to *arithmetically**, `sra` points to *arithmetically**, `rol` points to *rotate left*, and `ror` points to *rotate left*.

- **Schieben:** `sll srl sla sra rol ror`

*logically = mit Nullen auffüllen

*arithmetically = mit dem Vorzeichen auffüllen

- **Arithmetisch:** `+ - * / mod rem abs **`
- Für `std_logic` und `std_logic_vector` überladen

Anweisungen 1/3

<code>V := expr;</code>	<code>-- Variablenzuweisung</code>
<code>S <= expr;</code>	<code>-- Signalzuweisung</code>
<code>if cond1 then</code> <code>statements1</code> <code>elsif cond2 then</code> <code>statements2</code> <code>else</code> <code>statements3</code> <code>end if;</code>	<code>-- Verzweigung</code>

Anweisungen 2/3

```
case expr is      -- Mehrfachverzweigung
    when value1 => statements1
    when value2 => statements2
    when value3 => statements3
    when others  => statements0
end case;
```

Anweisungen 3/3

```
while expr loop
```

```
-- While-Schleife
```

```
    statements
```

```
end loop;
```

```
for I in 1 to 10 loop
```

```
-- For-Schleife
```

```
    statements
```

```
end loop;
```

```
loop
```

```
-- Endlos-Schleife
```

```
    statements
```

```
    exit when cond
```

```
-- Abbruchbedingung
```

```
    next when cond
```

```
-- Nächste Iteration
```

```
end loop;
```

Beispiel: D-FlipFlop

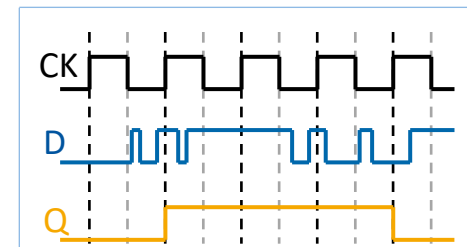
```
library ieee;
use ieee.std_logic_1164.all;

entity DFlipFlop is
    port (D, CLK : in  std_logic;
          Q      : out std_logic := '0');
end entity DFlipFlop;

architecture Verhalten of DFlipFlop is
begin
    Schalten : process (CLK)
    begin
        if rising_edge (CLK) then
            Q <= D;
        end if;
    end process Schalten;
end architecture Verhalten;
```

-- Eingangssignale
-- Ausgangssignal

-- Prozess mit
-- Sensitivity List
-- Hilfsfunktion
-- Signalzuweisung



Kurzschreibweise

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

entity DFlipFlop is
    port (D, CLK : in  std_logic;           -- Eingangssignale
          Q       : out std_logic := '0');  -- Ausgangssignal
end entity DFlipFlop;

architecture Verhalten of DFlipFlop is
begin
    Q <= D when rising_edge(CLK);
end architecture Verhalten;

-- kein Prozess, aber (fast) äquivalent
-- bedingte, parallele Signalzuweisung
-- implizite Sensitivity List mit allen Eingängen
```

- Verbinden mehrerer Entities
- Kann mit Process gemischt werden
- Grundprinzip der Hierarchie
- Mapping von Ein- und Ausgängen
- Wird vom „Compiler“ aufgelöst
- Bei mehr als einer Architecture:
 - Komplizierte automatische Auswahl (Fehlerquelle)
 - Explizit Auswahl durch Configuration

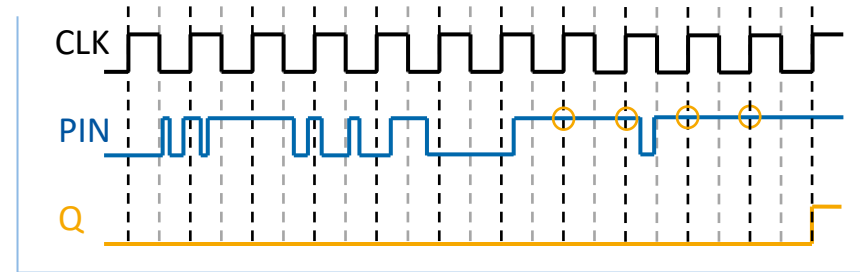
Beispiel: Debouncer 1/2

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

entity Debouncer is
    port (PIN, CLK : in  std_logic;
          Q       : out std_logic := '0');
end entity Debouncer;

architecture Aufbau of Debouncer is
    signal Q1, Q2, Q3, Q4 : std_logic := '0';

    component DFlipFlop
        port (D, CLK : in  std_logic;
              Q      : out std_logic := '0');
    end component;
```



Beispiel: Debouncer 2/2

```
function all_equal(A, B, C, D : std_logic) return boolean is
```

```
begin
```

```
    return A = B and B = C and C = D;
```

```
end function all_equal;
```

```
begin
```

```
    DFF1 : DFlipFlop port map (PIN, CLK, Q1);
```

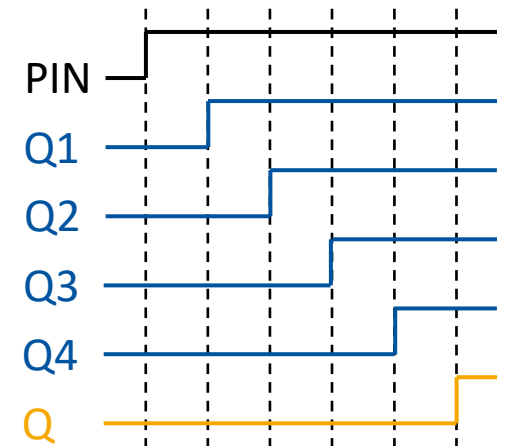
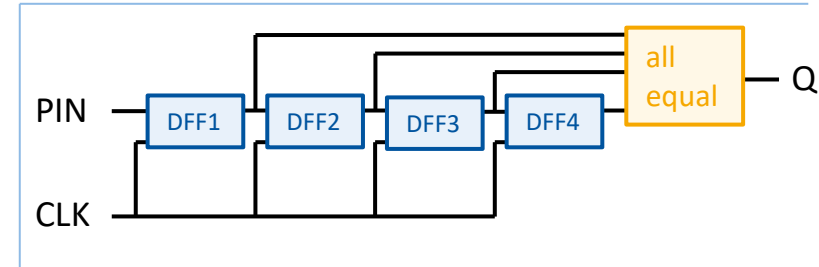
```
    DFF2 : DFlipFlop port map (Q1, CLK, Q2);
```

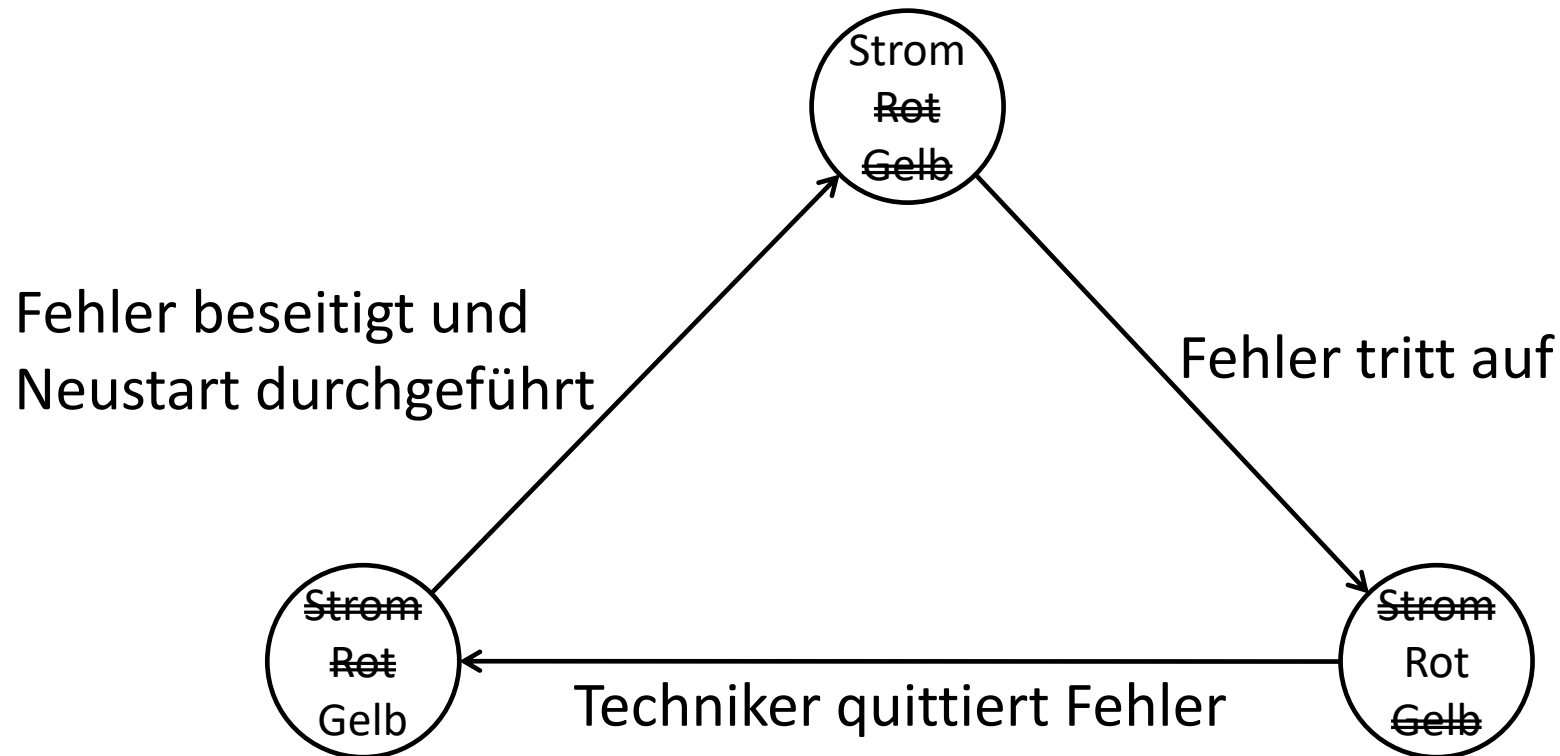
```
    DFF3 : DFlipFlop port map (Q2, CLK, Q3);
```

```
    DFF4 : DFlipFlop port map (Q3, CLK, Q4);
```

```
    Q <= Q1 when all_equal(Q1, Q2, Q3, Q4);
```

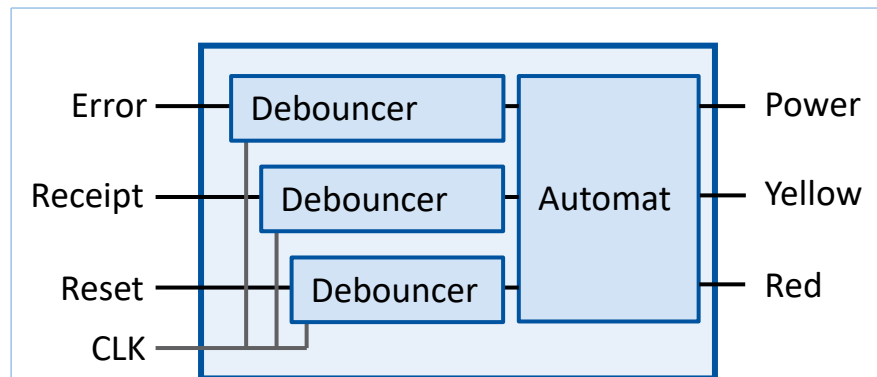
```
end architecture Aufbau;
```





Beispiel: Alarmkontrolle (Entity)

```
library ieee;  
  
use ieee.std_logic_1164.all;  
  
entity AlarmController is  
    port (ERROR, RECEIPT, RESET, CLK : in  std_logic;  
          POWER, YELLOW, RED       : out std_logic);  
end entity;
```



Beispiel: Alarmkontrolle (Components)

```
architecture Kombi of AlarmController is
    signal D_ERROR, D_RECEIPT, D_RESET : std_logic;

    component DFlipFlop
        port (
            D, CLK : in  std_logic;
            Q       : out std_logic := '0');
    end component;

    component Debouncer
        port (
            PIN, CLK : in  std_logic;
            Q        : out std_logic := '0');
    end component;
```

Beispiel: Alarmkontrolle (Mapping)

```
DB1 : Debouncer port map (ERROR, CLK, D_ERROR);  
DB2 : Debouncer port map (RECEIPT, CLK, D_RECEIPT);  
DB3 : Debouncer port map (RESET, CLK, D_RESET);
```

Beispiel: Alarmkontrolle (Process)

```
Schalten : process (CLK)
    variable state : integer := 1;
begin
    if rising_edge (CLK) then
        case state is
            when 1      => POWER <= '1'; YELLOW <= '0'; RED <= '0';
            when 2      => POWER <= '0'; YELLOW <= '0'; RED <= '1';
            when 3      => POWER <= '0'; YELLOW <= '1'; RED <= '0';
            when others => POWER <= '0'; YELLOW <= '1'; RED <= '1';
        end case;
        if state = 1 and D_ERROR = '1' then state := 2;
        elsif state = 2 and D_RECEIPT = '1' then state := 3;
        elsif state = 3 and D_RESET = '1' and D_ERROR = '0' then
            state := 1;
        end if;
    end if;
end process;
end architecture;
```

Beispiel: Alarmkontrolle (Configuration)

```
use work.all;

configuration Binding of AlarmController is
  for Kombi
    for all : Debouncer
      use entity work.Debouncer (Aufbau);
      for Aufbau
        for all : DFlipFlop
          use entity work.DFlipFlop (Verhalten);
        end for;
      end for;
    end for;
  end for;
end Binding;
```

Es gibt noch viel mehr...

- Generische Entities
 - Variable Wordgröße
 - Variable Pufferlänge
- Automatische Configurations
- Umfangreiche Bibliotheken
- Hardwaresynthese
- Analoge Schaltungen
- Quellen
 - VHDL Kompakt: <http://tams.informatik.uni-hamburg.de>
 - Standard: IEC 61691-1-1 04