Logischer Entwurf digitaler Systeme

Falk Lesser, Volker Lindenstruth
Institut für Hochenergiephysik

Ziele des Kurses

- Die Syntax und Struktur einer VHDL-Beschreibung zu erlernen
- Die Fundamente der Modellierung und Synthese aus einfachen Modellen zu entwickeln.
- Den Umgang mit den Synthesewerkzeugen (Synopsys, Orca) zu verstehen
- Die selbst entworfenen VHDL-Modelle auf mehreren Entwurfsebenen zu simulieren.

Kursübersicht

- **Allgemein:** 50 % Unterricht, 50 % Übungen
- **Tag 1:** Grundlagen der Technischen Informatik, Elemente der Hardwarebeschreibungssprache VHDL
- **Tag 2:** Syntax und Struktur einer VHDL-Beschreibung Synthesewerkzeuge des digitalen Designflows
- **Tag 3:** Synchrone Logik, Synthesegerechte Beschreibung in VHDL
- **Tag 4:** Endliche Automaten, Synthesetools des Orca-Designflows
- **Tag 5:** Tips und Tricks, Selbstständige Designaufgabe

Inhalt des ersten Tages

- Grundlagen der Technischen Informatik
- Entwurf digitaler Systeme
- Hardwaredesign mittels einer Hardwarebeschreibungssprache (HDL)
- Grundelemente einer VHDL-Beschreibung
- Syntax und Struktur einer VHDL-Beschreibung

Warum Digital

- Ein Computer speichert, verarbeitet und produziert Informationen
- Alle Informationsverarbeitungen müssen als Funktion der Anfangswerte reproduzierbar sein

➔ Informationsspeicherung und Verarbeitung müssen exakt sein

- Probleme: Noise, Crosstalk, Abschwächung

➔ Es gibt keine exakte Datenübertragung oder Datenspeicherung

➔ Quantisierung der Informationsspeicherung mit Signal groß gegenüber maximale Störung

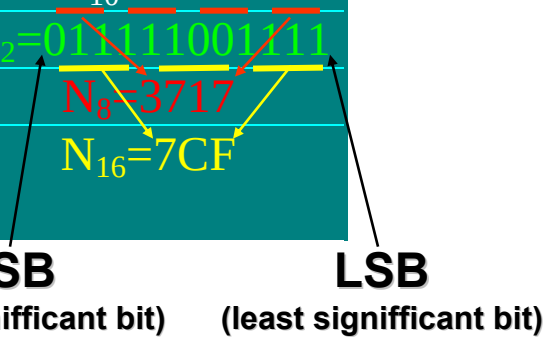
- Binäre Codierung (nur zwei Zustände) ist die einfachste Signal Quantisierung, das *BIT* die einfachste binäre Datenmenge

Positive Ganze Zahlen

B: Basis des Zahlensystems
S: Anzahl der Stellen
i: Stelle
z_i: Ziffer an der i. Stelle

Verschiedene Zahlensysteme sind in Gebrauch:

System	Basis	Ziffern	Beispiel
Dezimal	10	0…9	N_{10}=1999
Binär	2	0…1	N_2=011111001111
Oktal	8	0…7	N_8=3717
Hexadezimal	16	0…9, A,B,C,D,E,E,F	N_{16}=7CF



→ Oktal und Hexadezimal sind Zusammenfassungen von Binär

Binärarithmetik

Arithmetische Operationen sind unabhängig von der Basis des zugrundeliegenden Zahlensystems

Binäraddition

$938_{10}+140_{10} = 1078_{10}$

	1110101010	Erster Summand
+	10001100	Zweiter Summand
	1110001000_	Übertrag (Carry)
	10000110110	Summe

Binärmultiplikation

$13_{10} \cdot 10_{10} = 130_{10}$

1010 • 1101	Faktoren
0000	0 • 1101
1101	1 • 1101
0000	0 • 1101
1101	1 • 1101
111000_	Übertrag (Carry)
10000010	Summe

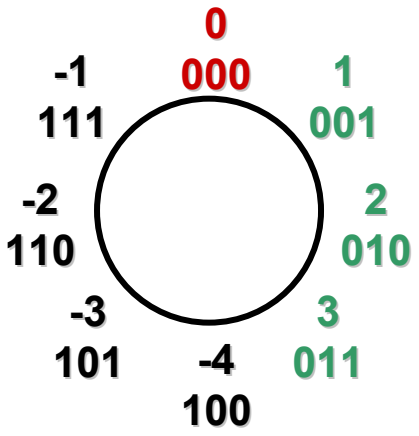
Merke:

- Multiplikation mal 2 == links shift
- Division durch 2 == rechts shift (modulo rest = Carry)
- n-Bit Addition == n Binäradditionen
- n x m-Bit Multiplikation == n m-Bit Additionen

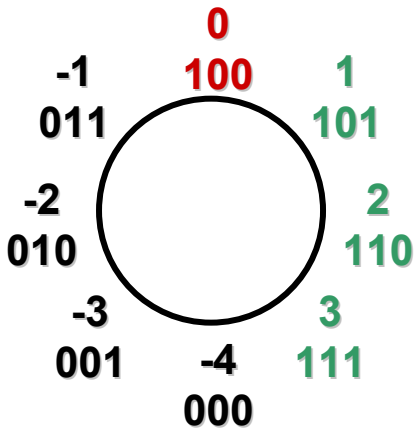
Negative Zahlen, Subtraktion

Ansatz 1: Vorzeichen Bit (negative Zahl entsteht durch Setzen des Vorzeichens)
Ansatz 2: Einerkomplement (negative Zahl entsteht durch Invertieren aller Bits)
Nachteil: Null ist doppelt definiert (+0, -0)
→ Fallunterscheidungen komplizieren Arithmetik
Ansatz 3: Bias (Binärrepresentation ist Zahl plus Bias)
Ansatz 4: Zweierkomplement (= Einerkomplement + 1)
Vorteile: Vorzeichen am ersten Bit erkennbar
Addition negativer Zahlen identisch, Addition positiver Zahlen (A-B == A+(-B)) - in Hardware sehr einfach

Zweierkomplement



Bias (4)



Beispiel: 3-2:

Zweierkomplement

$3_{10} = 011_2$

$-2_{10} = 101_2 + 1 = 110_2$

→ $3_{10}-2_{10} = 011_2 + 110_2 = c\ 001_2$

Bias

$3_{10} = 4_{10} + 3_{10} = 111_2$

$-2_{10} = 4_{10} - 2_{10} = 010_2$

→ $3_{10}-2_{10} = 111_2 + 010_2 = c\ 101_2$

Verarbeitung binärer Daten

B^n B = {0, 1} B^m

Def. Schaltnetz: Binäre Operation, die kein Gedächtnis ihrer Vorgeschichte hat

Def. Schaltfunktion: Binär Operation   f: $B^n \rightarrow B^m$

Def. Boolesche Funktion (n-stellig):   f: $B^n \rightarrow B$

Merke: Schaltfunktion = Vektor Boolscher Funktionen

Satz: es gibt 2^{2^n} n-stellige Boolsche Funktionen

mögliche Ergebnisvektoren

i_1	i_0	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	*0*	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	*1*	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	*0*	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	*1*	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

2^n Eingabe Kombinationen

2^n Stelliger ErgebnisVektor

AND (f_1)

XOR (f_6)

OR (f_7)

NOR (NOT OR) (f_8)

NAND (NOT AND) (f_{14})

Boolesche Operatoren

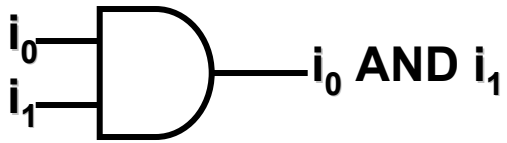
Std. Name	*Dt. Name*	*Symbole*	*Beispiel*
NOT	Negation	~,!,/,¬, ‾	$\overline{A}$
AND	Konjunktion	∧, &, •	A•B
NAND			$\overline{A \bullet B}$
OR	Disjunktion	∨,#,+	A+B
NOR			$\overline{A+B}$
XOR		⊕	A⊕B

Merke:

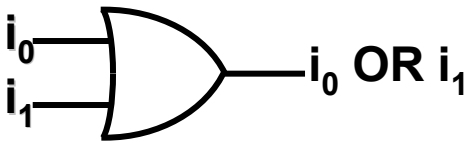
A NAND B	**= NOT (A AND B)**	$= \overline{A \bullet B}$
A NOR B	**= NOT (A OR B)**	$= \overline{A + B}$
A XOR B	**= (A AND (NOT B)) OR ((NOT A) AND B)**	$= (A \bullet \overline{B}) + (\overline{A} \bullet B)$

Darstellung Boolescher Operatoren

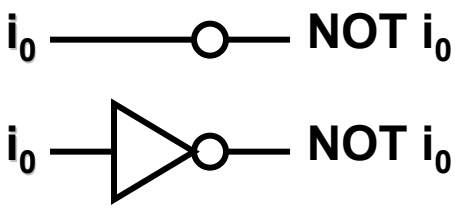
i_1	i_0	i_1 AND i_0
0	0	0
0	1	0
1	0	0
1	1	1



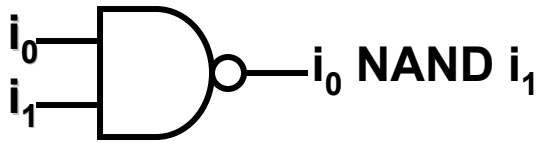
i_1	i_0	i_1 OR i_0
0	0	0
0	1	1
1	0	1
1	1	1



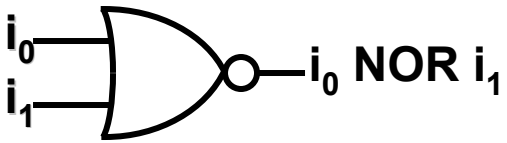
i_0	NOT i_0
0	1
1	0



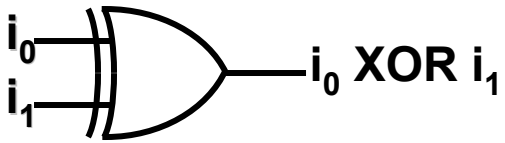
i_1	i_0	i_1 NAND i_0
0	0	1
0	1	1
1	0	1
1	1	0



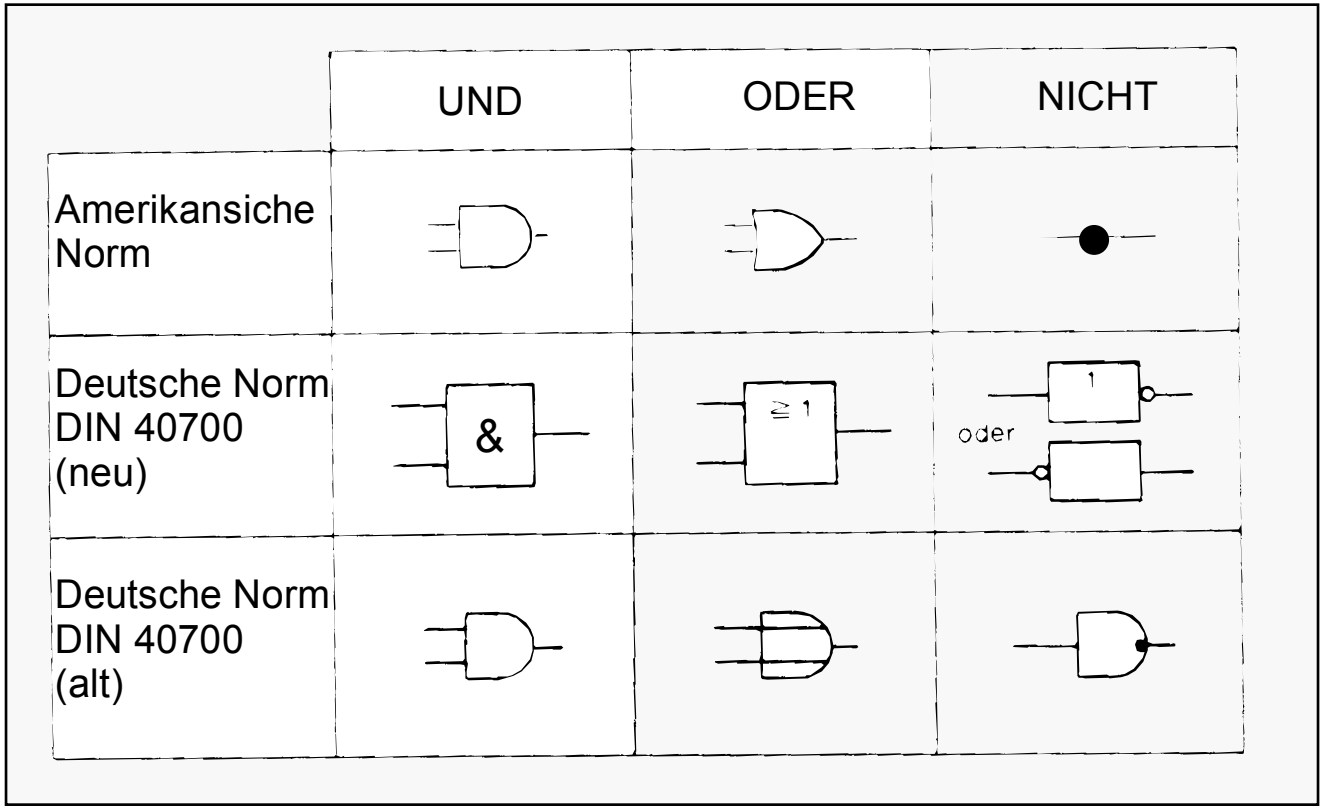
i_1	i_0	i_1 NOR i_0
0	0	1
0	1	0
1	0	0
1	1	0



i_1	i_0	i_1 XOR i_0
0	0	0
0	1	1
1	0	1
1	1	0



Symbole einiger Industrienormen



ANSI: American National Standards Institute
IEEE: Istitute of Electrical and Electronic Engineers
IEC: Insternational Electrotechnical Commission
DIN: Deutsche Industrie Norm

Im Folgenden:

ANSI/IEEE Standard 91-1984:

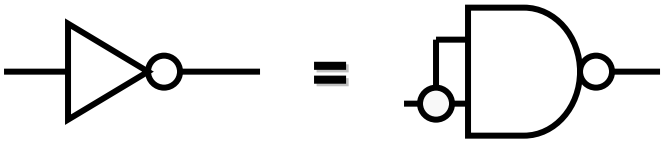
Standard Graphic Symbols for Logic Functions

Kompatibel mit IEC 617 Standard

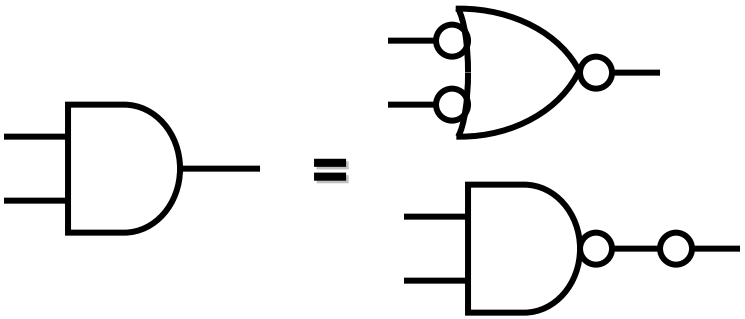
Rechenregeln Boolescher Algebra

- Seien x, y und z Boolesche Variablen, d.h. Variablen, die die Werte 0 oder 1 annehmen können.
- + und $\oplus$ sind assoziativ, d.h. (x o y) o z = x o (y o z); für o $\in \{\bullet, +, \oplus\}$.
- + und $\oplus$ sind kommutativ, d.h. x o y = y o x; für o $\in \{\bullet, +, \oplus\}$.
- ist distributiv über + d.h. $x \bullet (y + z) = (x \bullet y) + (x \bullet z)$.
- ist distributiv über $\oplus$.
- $\overline{\overline{x}} \equiv x \equiv x + x \equiv x \bullet x$
- $\overline{x \bullet y} \equiv \overline{x} + \overline{y}$ und $\overline{x + y} \equiv \overline{x} \bullet \overline{y}$ (de Morgan Regel)
- $(x + y) \bullet x \equiv x$ und $(x \bullet y) + x \equiv x$ (Verschmelzungsregel)
- $x + (y \bullet \overline{y}) \equiv x$ und $x \bullet (y + \overline{y}) \equiv x$ (Komplementregel)
- $x + 0 \equiv x$, $x \bullet 0 \equiv 0$, $x + 1 \equiv 1$, $x \bullet 1 \equiv x$ (Absorptionsregeln)

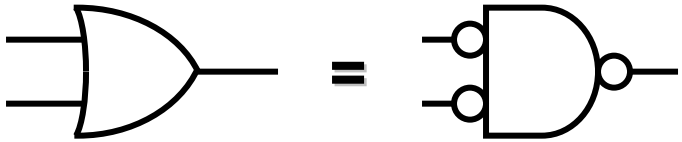
Redundanz der Operatoren



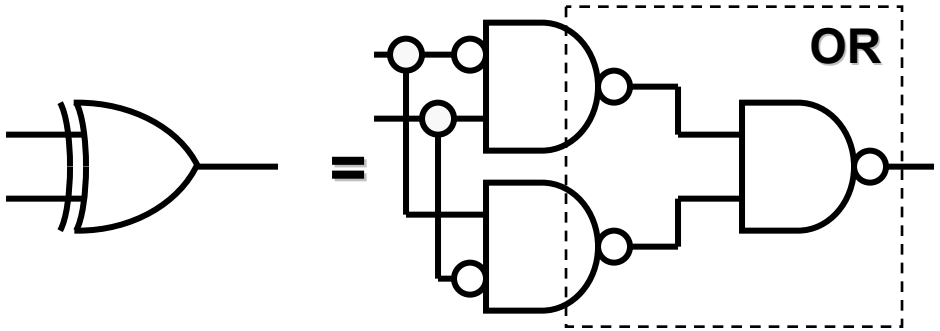
$\overline{A}$ = A NAND A



A AND B = $\overline{(A \text{ NAND } B)}$



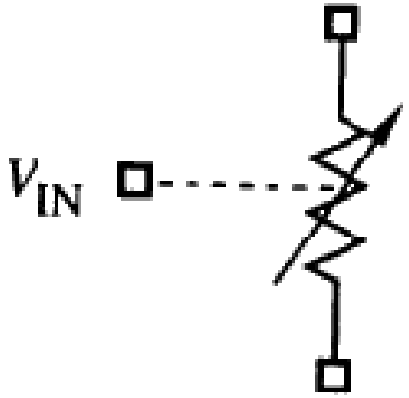
A OR B = $\overline{A}$ NAND $\overline{B}$



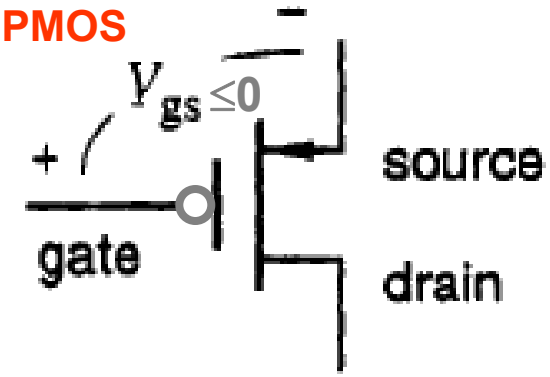
A XOR B = ($\overline{A}$ NAND B) NAND
($\overline{A}$ NAND $\overline{B}$)

Transistor

Def.: MOS Metall-Oxid-Silizium

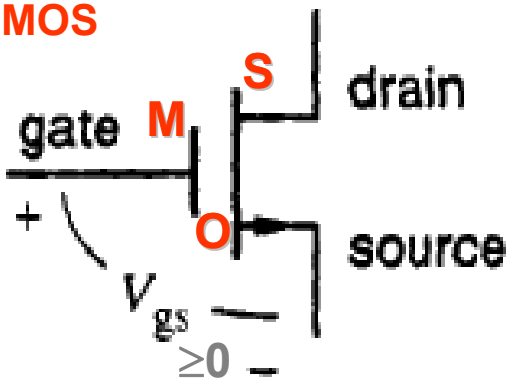


PMOS



kleineres V_{gs} → kleineres R_{ds}

NMOS



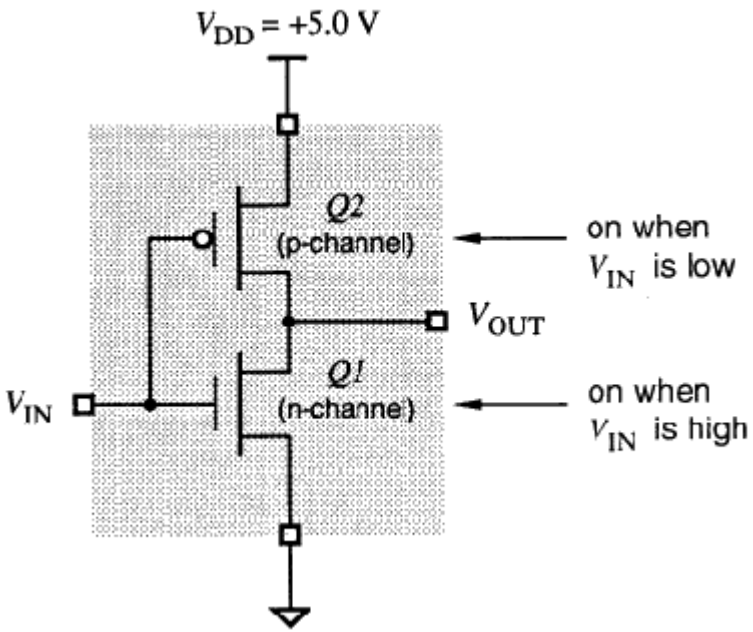
größeres V_{gs} → kleineres R_{ds}

Ein MOS Transistor ist ein Spannungsgesteuerter Schalter

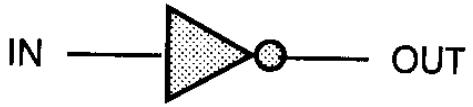
Merke: n-Kanal CMOS Transistoren haben eine kleinere Durchlaßimpedanz als p-Kanal CMOS Transistoren

CMOS Inverter

Transistor Schaltbild



Gattersymbol

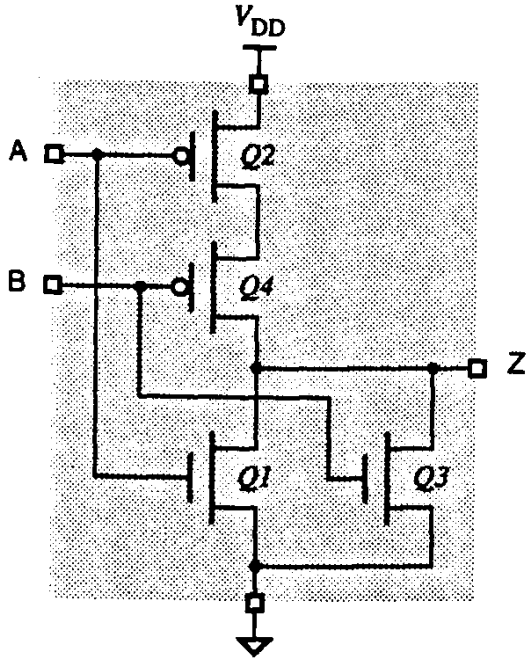


Wahrheitstabelle

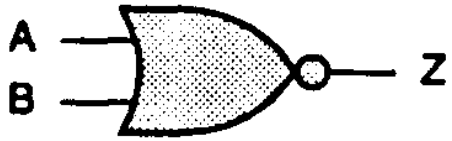
V_{IN}	Q1	Q2	V_{OUT}
0.0 (L)	off	on	5.0 (H)
5.0 (H)	on	off	0.0 (L)

CMOS NOR

Transistor Schaltbild



Gattersymbol

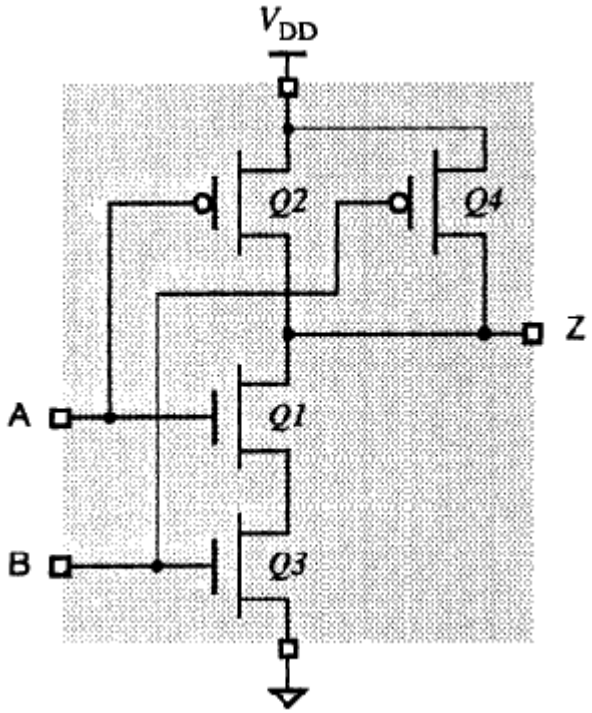


Wahrheitstabelle

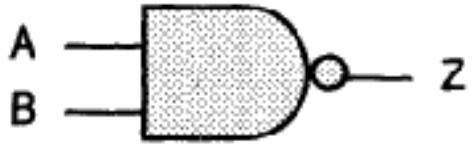
A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	L
H	L	on	off	off	on	L
H	H	on	off	on	off	L

CMOS NAND

Transistor Schaltbild



Gattersymbol



Wahrheitstabelle

A	B	*Q1*	*Q2*	*Q3*	*Q4*	Z
L	L	off	on	off	on	H
L	H	off	on	on	off	H
H	L	on	off	off	on	H
H	H	on	off	on	off	L

Erinnerung: n-Kanal CMOS Transistoren haben eine kleinere Durchlaßimpedanz als p-Kanal CMOS Transistoren

➔ CMOS NAND Gatter sind schneller als NOR Gatter

Darstellung Boolescher Funktionen

Wertetabelle

i	x_2	x_1	x_0	$f_S(x_2,x_1,x_0)$
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	0
4	1	0	0	1
5	1	0	1	0
6	1	1	0	0
7	1	1	1	1

Beispiel: f_S sei 1 wenn die Anzahl der Argumente, die 1 sind, ungerade ist

Die Zeilennummern i mit Funktionsergebnis 1 heißen einschlägige Indizes (im Beispiel: 1,2,4,7)

Aussagelogik

$$f_S(A, B, C) = (\bar{x}_2 \bullet \bar{x}_1 \bullet x_0) + (\bar{x}_2 \bullet x_1 \bullet \bar{x}_0) + (x_2 \bullet \bar{x}_1 \bullet \bar{x}_0) + (x_2 \bullet x_1 \bullet x_0)$$
$$= x_2 \oplus x_1 \oplus x_0$$

Verschiedene Repräsentationen derselben Funktion möglich, Wertetabelle eindeutig.

Minterm

Definition: Minterm m_i :

Es seien f eine Boolesche Funktion, i ein einschlägiger Index von $f(x_1,..,x_n)$ und $i_1,..,i_n$ die Dualdarstellung von i.

Eine Funktion $m_i: B^n \rightarrow B$ mit

$$m_i(x_1,..,x_n) = l_1 \bullet l_2 \bullet l_3 \bullet .. \bullet l_n$$

und

$$l_i = x_j, \text{ falls } i_i = 1$$
$$l_i = \overline{x}_j, \text{ falls } i_i = 0$$

heißt i - ter Minterm von f.

Merke: Ein Minterm m_i nimmt genau an der Stelle i den Wert 1 an und liefert sonst immer 0.

Beispiel: $m_1 = \overline{x}_2 \bullet \overline{x}_1 \bullet x_0$

Disjunktive Normalform (DNF)

Satz: Jede Boolesche Funktion f: $B^n \rightarrow B$ kann eindeutig als ODER - Verknüpfung der Minterme ihrer einschlägigen Indizes dargestellt werden.

Beispiel:

i	x_2	x_1	x_0	$f_S(x_2,x_1,x_0)$	
0	0	0	0	0	
1	0	0	1	1	m_1
2	0	1	0	1	m_2
3	0	1	1	0	
4	1	0	0	1	m_4
5	1	0	1	0	
6	1	1	0	0	
7	1	1	1	1	m_7

einschlägige Indizes: 1(001), 2(010), 4(100), 7(111)

Minterme dazu: $x_2 \bullet \overline{x_1} \bullet \overline{x_0}$ $\quad \overline{x_2} \bullet x_1 \bullet \overline{x_0}$ $\quad \overline{x_2} \bullet \overline{x_1} \bullet x_0$ $\quad x_2 \bullet x_1 \bullet x_0$

die DNF lautet: $f = (\overline{x_2} \bullet \overline{x_1} \bullet x_0) + (\overline{x_2} \bullet x_1 \bullet \overline{x_0}) + (x_2 \bullet \overline{x_1} \bullet \overline{x_0}) + (x_2 \bullet x_1 \bullet x_0)$

Operatorensysteme, Vollständigkeit

Definition: Ein Operationensystem ist vollständig wenn jede beliebige Boolesche Funktion f: $B^n \rightarrow B$ durch diese Operatoren ausgedrückt werden kann.

Merke: Jede beliebige Boolesche Funktion hat eindeutige DNF Darstellung

➔ Operatoren {AND, OR, NOT} Vollständig

Merke: AND, OR, NOT können durch NAND oder auch NOR ausgedrückt werden

➔ Operatoren {NAND}, {NOR} Vollständig
➔ Alle digitalen Schaltungen (z.B. auch Pentium Prozessor) können auf eine Kombination von NAND oder NOR Operatoren zurückgeführt werden!

Halbaddierer

Def. Halbaddierer: Berechnung der Summe und Übertrag (Carry) von zwei Eingängen

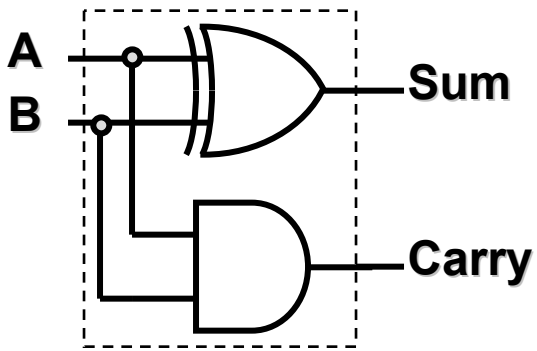
Einfaches Beispiel: 1-bit Halbaddierer

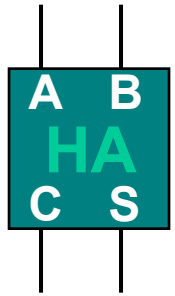
Wertetabelle

A	B	Carry	Sum	Dec
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	0	2

Gatterrepräsentation

Sum = A ⊕ B
Carry = A • B





Binäre Addition

$938_{10} + 140_{10} = 1078_{10}$

	1110101010	Erster Summand
+	10001100	Zweiter Summand
	1110001000_	Übertrag (Carry)
	10000110110	Summe

Merke: C_{out} von Stufe (Bit) N ist C_{in} von Stufe N+1

➔ **Volladdierer**

C_{in}	A	B	C_{out}	Sum	Dec
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	1
0	1	1	1	0	2
1	0	0	0	1	1
1	0	1	1	0	2
1	1	0	1	0	2
1	1	1	1	1	3

Gatterrepräsentation

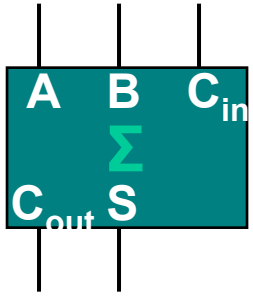
$$\text{Sum} = A \oplus B \oplus C_{in}$$
$$C_{out} = A \bullet B + A \bullet C_{in} + B \bullet C_{in}$$

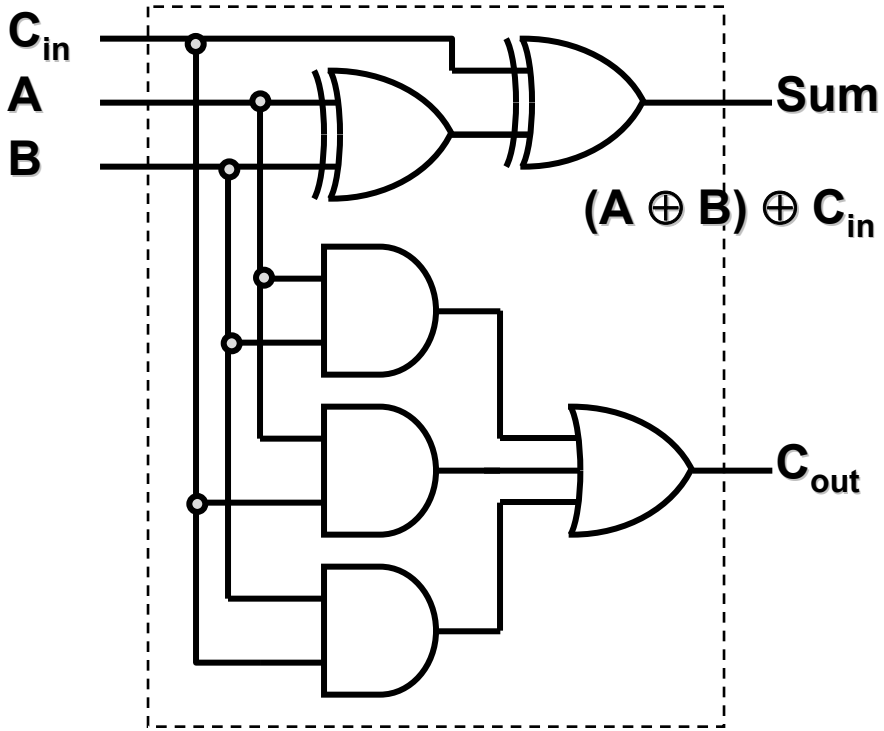
Volladdierer

Gatterrepräsentation

$\text{Sum} = A \oplus B \oplus C_{in}$

$C_{out} = A \bullet B + A \bullet C_{in} + B \bullet C_{in}$

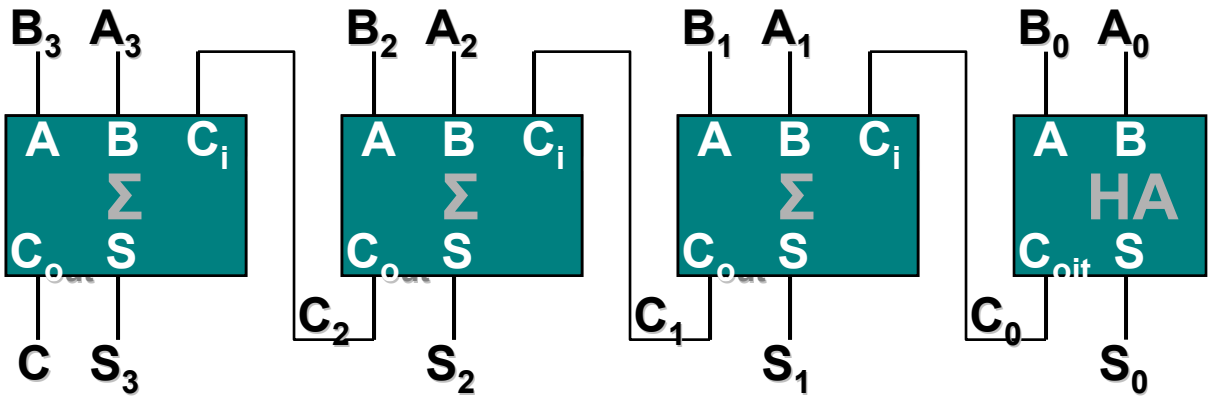




Merke: Standard Gatterlaufzeiten:

XOR:	2
A,B → Sum:	4
Cin → Sum:	2
Cout:	2

4-Bit Addierer



$11_{10}+5_{10} = 16_{10}$

```
   0101  A
+  1011  B
-------------
  1111_  Übertrag (Carry)
  10000  Summe
```

t	*A[3:0]*	*B[3:0]*	C_{out}*[2:0]*	*C*	*S[3:0]*
0	0101	1011	UUU	U	UUUU
2	0101	1011	001	0	UUUU
4	0101	1011	011	0	1100
6	0101	1011	111	0	1000
8	0101	1011	111	1	0000
10	0101	1011	111	1	0000

Ripple Carry Adder

Zusammenfassung Schaltnetze

- Die Zahlenbasis sind binäre Werte {0, 1}
- Logische Funktionen werden mit Hilfe der Booleschen Algebra beschrieben
- Logische Funktionen werden mit den Rechenregeln der Booleschen Algebra vereinfacht
- Boolesche Operatoren haben einen direkten Verwandten als elektronische Schaltung
- Alle digitalen Schaltungen bestehen aus einer Zusammenschaltung von logischen Grundgattern
- Grundgatter bestehen aus Transistoren mit realen Verzögerungszeiten.

Grundlagen des mikroelektronischen Systementwurfs

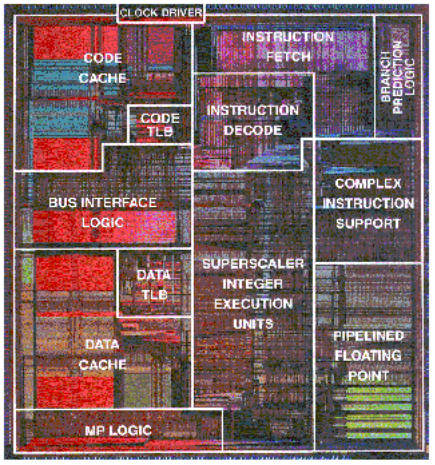
Falk Lesser, Volker Lindenstruth,
Institut für Hochenergiephysik

Einleitung

- Mehrere Millionen Transistoren auf einem Chip
  - Kurze Entwicklungszeiten (*time-to-market*)
  - Überschaubare Entwicklungskosten

Pentium mit mehreren Millionen Transistroren

Entwicklungszeit 16 Monate, 200 Designer

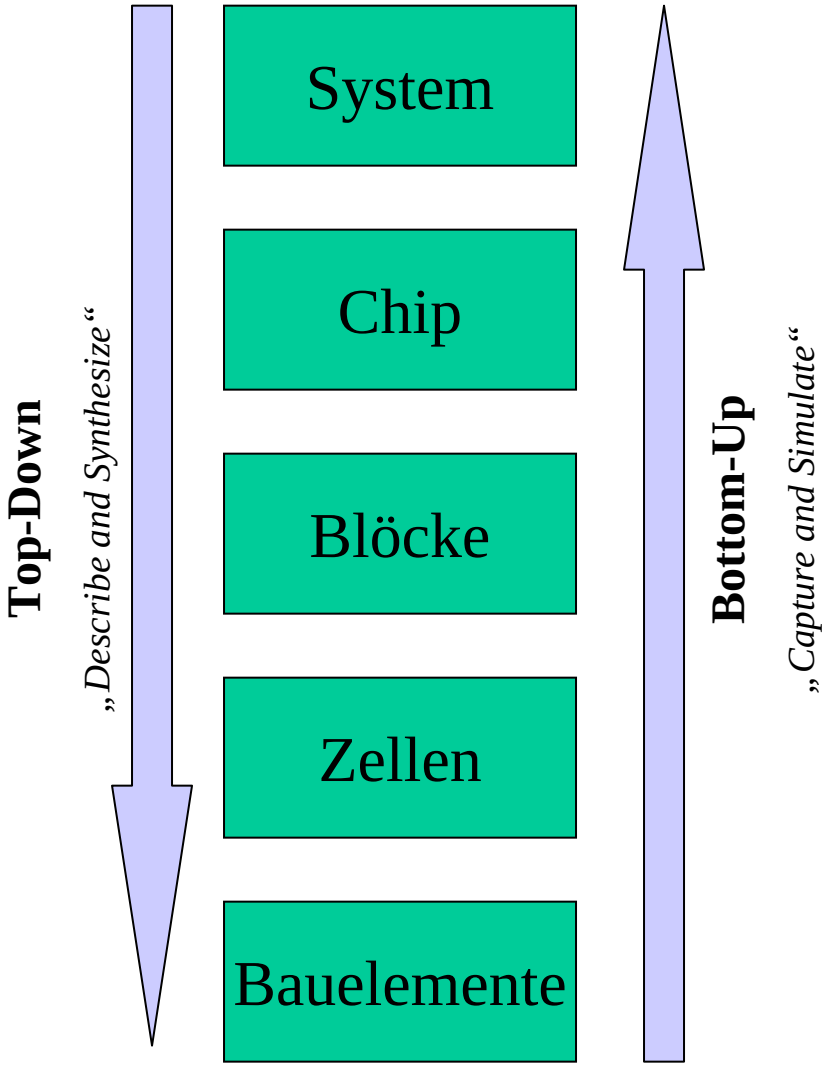


- Plazieren Verschalten und Verdrahten ist von Hand nicht mehr möglich
- Festgelegte Design-Regeln sichern den erfolgreichen Entwurf

Anforderungen

- Verifikation der:
  - Designregeln und der Funktionalität
- Abschätzung der:
  - Leistungsfähigkeit und der Kosten möglichst zu einem frühen Stadium des Entwurfs
- Möglichst hohe Abstraktionsebene der Schaltungsbeschreibung
  - gewährleistet die Technologieunabhängigkeit (Beispiel: 0,25 => 0,18 μm)

Enwurfssichten



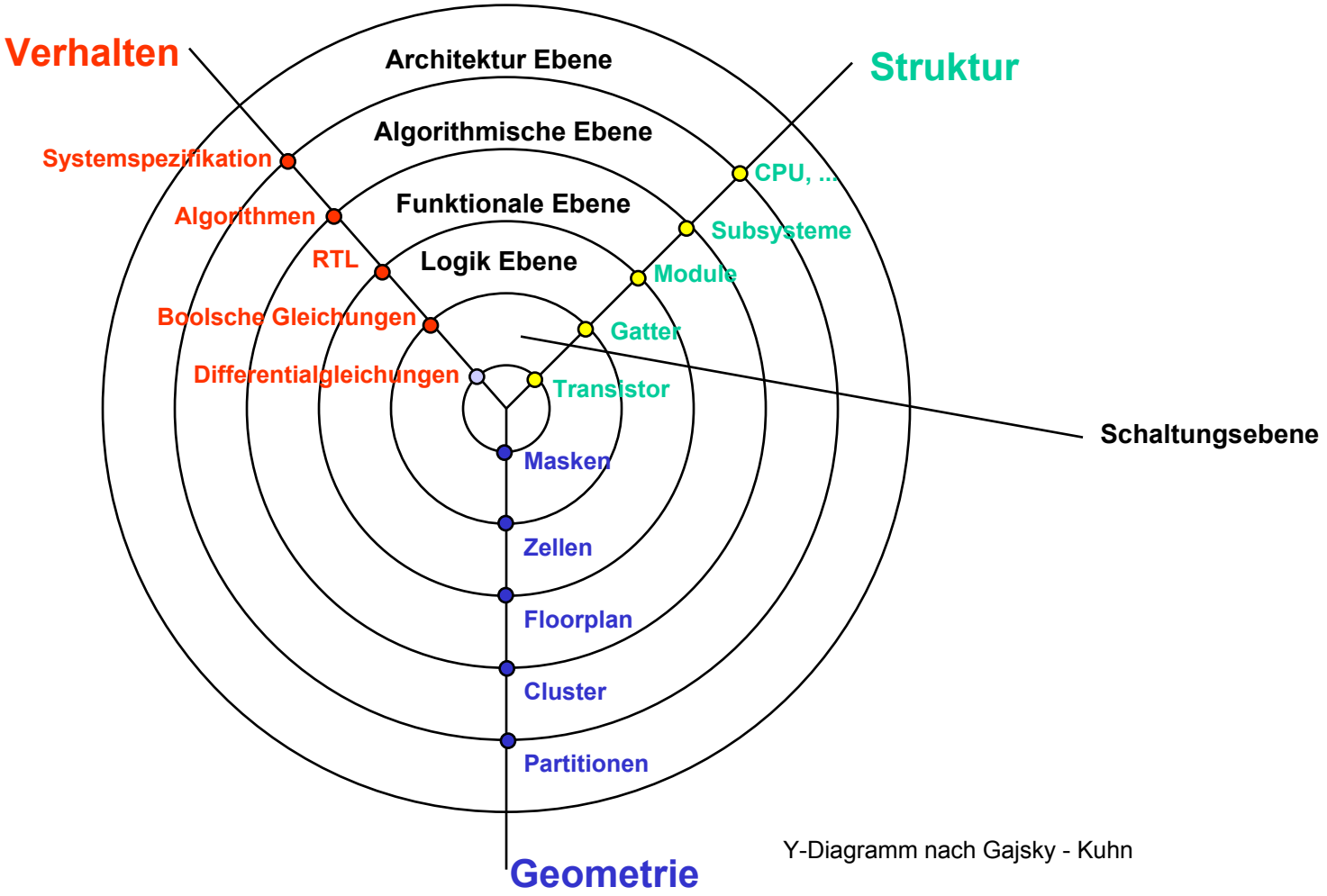
Top-Down:

- schnell und technologieunabhängig
- Ausgehend von einem hohen Abstraktionsniveau stetige Optimierung mittels Automationstools
- nicht optimiert

Bottom-Up:

- Transistoren werden von Hand gezeichnet und funktional verschaltet
- Die konstruierten Gatter werden zu Blöcken verschaltet usw.

Entwurfssichten



Y-Diagramm nach Gajsky - Kuhn

Entwurfsebenen

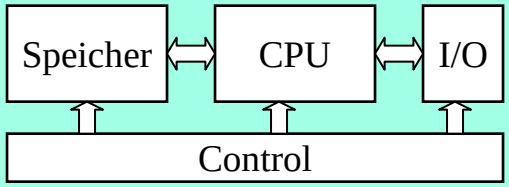
- Systemebene
  - Beschreibt die Charakteristika eines Systems durch Blöcke wie Speicher, Prozessoren und Interface-Einheiten
  - Keine Aussage über Signale, Zeitverhalten und funktionales Verhalten
  - Dient der Partitionierung der gesamten Schaltfunktion
- Algorithmische Ebene
  - Beschreibung des Systems durch nebenläufige Algorithmen wie Funktionen, Prozeduren und Prozesse
- Register-Transfer-Ebene
  - Beschreibung einer Schaltung durch Operationen (z.B. Addition) und durch den Transfer der verarbeiteten Daten zwischen Registern

Entwurfsebenen

- Logikebene
  - Beschreibung durch logische Verknüpfungen und deren zeitlichen Verhalten (z.B. Verzögerungszeiten)
  - Signalverläufe sind wertediskret, d.h. die Signale nehmen nur bestimmte, vordefinierte Logikwerte an (z.B. high, low, undefiniert)
- Schaltkreisebene
  - Beschreibung durch elektronische Bauelemente wie Transistoren, Widerstände und Kapazitäten.
  - Einzelne Module werden nicht mehr durch eine logische Funktion mit Verzögerungszeiten beschrieben, sondern durch ihre tatsächlichen Aufbau aus den Bauelementen.

Ebenen des Systementwurfs

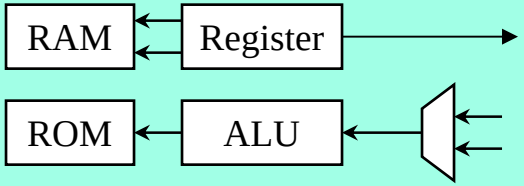
Systemebene



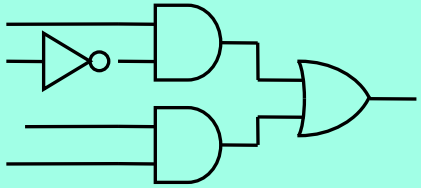
Algorithmische Ebene

```
A := 5 * B + C
i (D = true) then
        A := A + 1
else A := A -1
end if
```

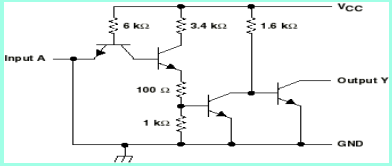
RT-Ebene



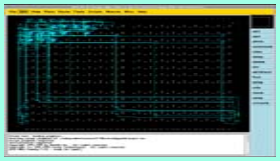
Logikebene



Schaltkreisebene (Struktur)



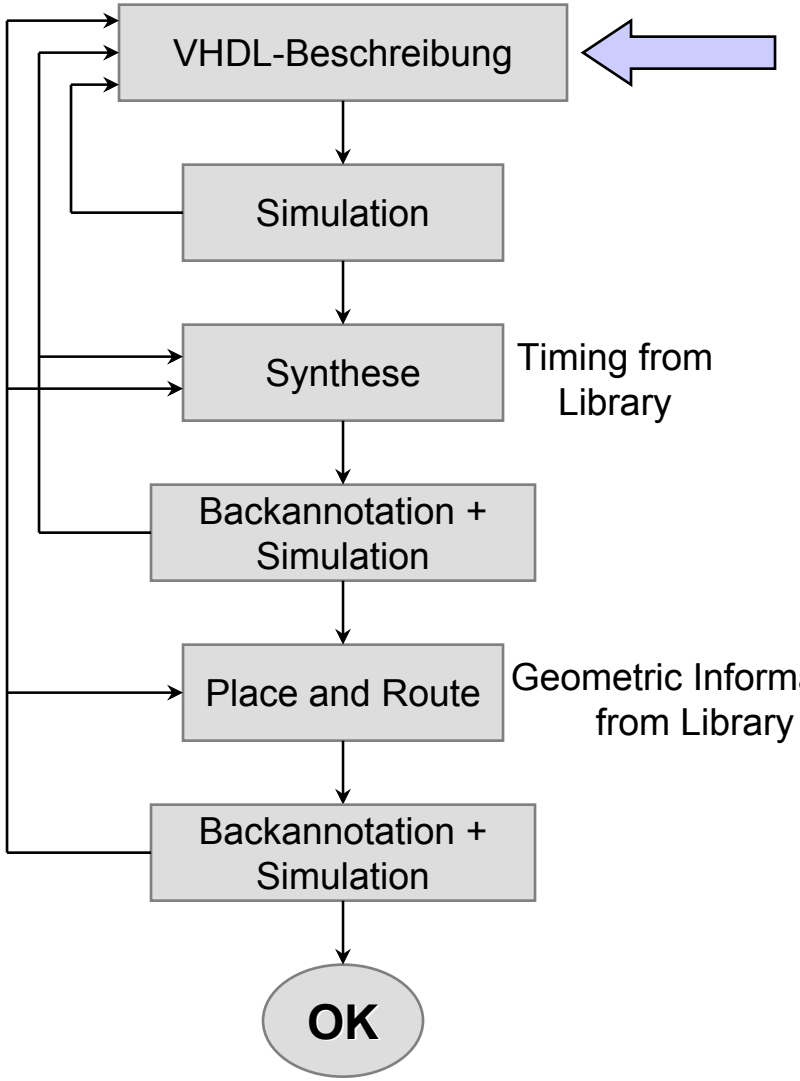
Schaltkreisebene (Geometrie)



Der digitale Designflow

- Möglichst automatisierter Entwurfsablauf
- Beschreibung einer Schaltung muß technologieunabhängig sein, da Strukturbreiten der verwendeten Zieltechnologie ständig schrumpft und Integrationsdichte dementsprechend steigt
- Die Beschreibung einer Schaltung geschieht auf einem hohen Abstraktionsniveau mittels einer Hardwarebeschreibungs-sprache (VHDL, Verilog)
- Simulation auf mehreren Entwurfsebenen
- Synthese (Übersetzung) der Schaltung in eine Zieltechnologie

Digitaler Designflow



Hier geht's jetzt weiter

Überprüfung der Funktionalität, wobei Gatterlaufzeiten, Treiberleistungen usw. unberücksichtigt bleiben.

Übersetzung der (von der Zieltechnologie unabhängigen) boolschen Beschreibung in eine Netzliste, basierend auf Elementen der Zielarchitektur

Entwurfsmethodiken

- Ein digitales System kann auf verschiedene Weise beschrieben werden
  - Beschreibung als Grafik (z.B. Schaltplaneingabe)
  - Textuelle Beschreibung als Netzliste (z.B. EDIF)
  - Textuelle Beschreibung auf Basis einer Hardwarebeschreibungssprache (z.B. VHDL, Verilog)

- Vorteile der Beschreibung einer Schaltung mittels einer HDL
  - Technologieunabhängige Beschreibung (Spezifikation)
  - Ausgangspunkt für die Simulation und Synthese einer Schaltung
  - HDLs dienen als Kommunikationsmedium zwischen Hardwareentwicklern innerhalb eines Projekts

Allgemeines zu VHDL

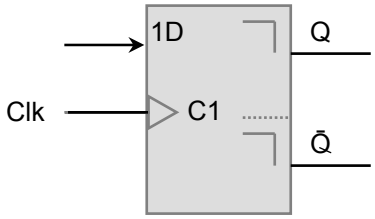
- VHDL = VHSIC Hardware Description Language
- VHSIC = Very High Speed Integrated Circuit
- seit 1987 IEEE-Standard (IEEE 1076-1987/1993)
- VHDL ist geeignet für Spezifikation, Simulation sowie als Ein- und Ausgabesprache für die Synthese
- VHDL ist Programm- und Architekturunabhängig
- Modellierungsmöglichkeiten (auf Algorithmischer Ebene, RTL, Logik und Kombinationen
- VHDL ist lesbar (wenig Dokumentationsaufwand)
- Sprachkonstrukte zur Parametrisierung von Modellen erlauben ein unkompliziertes Variantendesign
- VHDL enthält Konstrukte die sich nicht in eine Hardware-Realisierung umsetzen lassen

Aufbau einer VHDL-Beschreibung

VHDL-Beschreibung eines Designs besteht meist aus vier Teilen

Package	Typen, oft gebrauchte Funktionen, Prozeduren, Komponenten, Konstanten
Entity	Schnittstellenbeschreibung (I/O-Signale und Attribute)
Architecture	Strukturale Beschreibung oder Verhaltungsbeschreibung
Configuration	Zuordnung der Architekturvarianten und der Submodule der Entity

Beispiel:




```
Library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_components.all;
```

```
entity NAND is
port(          a, b:          in   std_logic;
               y    :          out std_logic);
end NAND;
```

```
architecture structural of NAND is
begin
      y <= not (a and b)
end structural;
```

```
configuration NAND_CFG of NAND is
   for structural
   end for;
end structural_CFG;
```

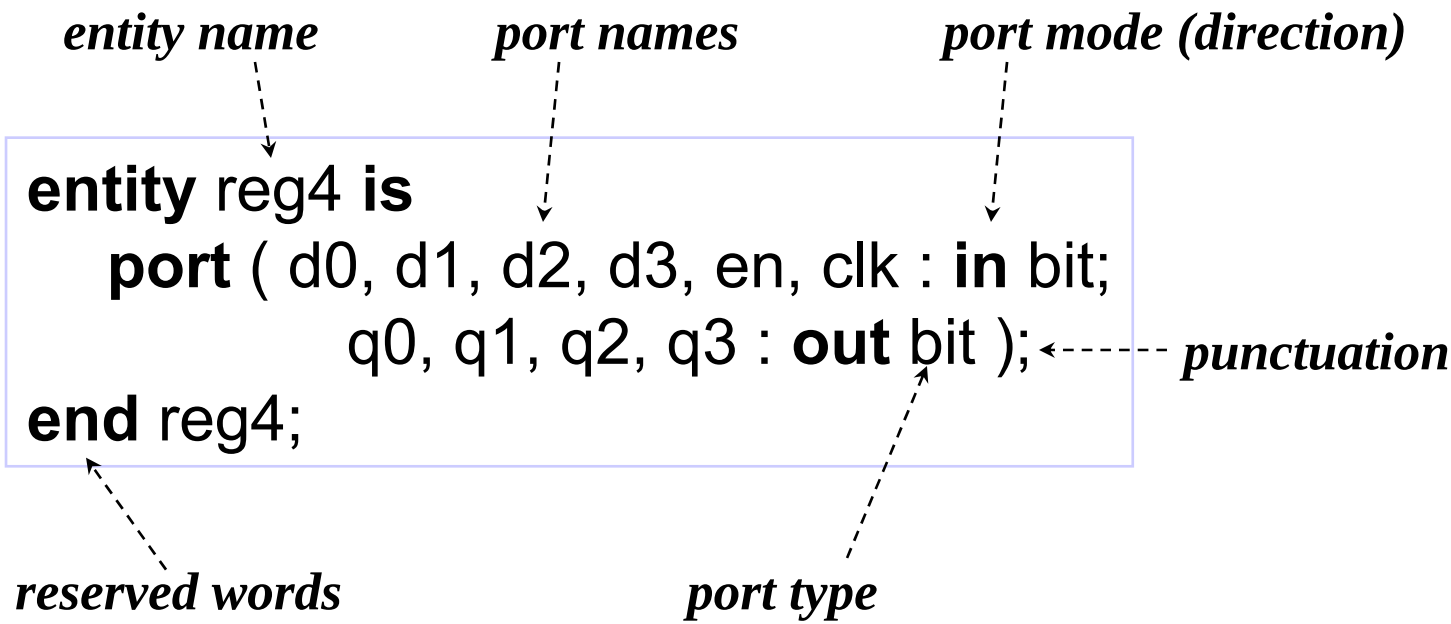
Aufbau einer VHDL-Beschreibung

- Entity (Schnittstellenbeschreibung)
  - Beschreibt die Ein- und Ausgänge sowie Konstanten, Unterprogramme und sonstige Vereinbarungen, die für alle dieser Entity zugeordneten Architekturen gelten soll
- Architecture (Architektur)
  - Enthält die Beschreibung der Funktionalität eines Modells
  - Entweder Verhaltens- oder strukturale Beschreibung oder Kombinationen aus Beiden
  - Mehrere Architekturen für eine Entity möglich
- Configuration (Konfiguration)
  - Die Konfiguration legt fest, welche Architektur einer bestimmten Entity zugeordnet ist und welche Zuordnungen für die verwendeten Submodule in der Architektur gelten

Schnittstellenbeschreibung (Entity)

- Schnittstelle des Systems zur „Außenwelt“
- Wichtigstes Element sind die **Eingänge** und **Ausgänge** (ports)
- Konstanten, Unterprogramme und sonstige Vereinbarungen
- Entspricht einer Modulbeschreibung oder einer Funktionsdeklaration in der Programmiersprache C

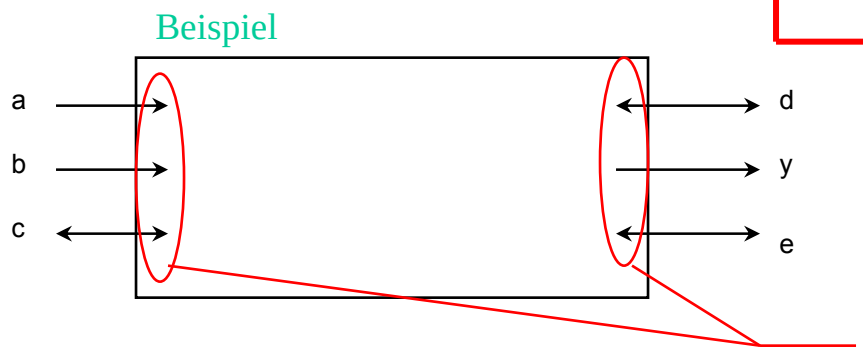
Beispiel:



Entity

Mit jeder Port-Deklaration der Entity wird implizit ein Signal gleichen Typs und gleichen Namens deklariert, das in den zugehörigen Architekturen verwendet werden kann.

```
Entity Beispiel is
generic(        delay:      time := 2.2 ns );
port(           a, b:       in   std_logic;     -- ports können nicht geschrieben werden
                y    :      out std_logic       -- ports können nicht gelesen werden
                c, d:       inout std_logic     -- ports können gelesen und geschrieben werden
                e:          buffer std_logic);  -- ports können gelesen und nur von einer Quelle
                                                   geschrieben werden
end Beispiel;
```

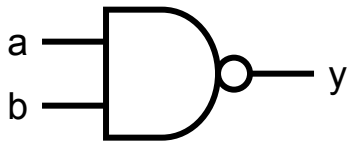


Zusätzlich zu den Ports werden in der entity die Parameter deklariert, die dem Modell übergeben werden können (sog. Generics)

Die **Außen** sichtbaren Ports stehen **innerhalb** des Designs ebenfalls zur Verdrahtung zur Verfügung.

Architektur (Architecture)

- Enthält die Beschreibung der Funktionalität einer entity
  1. Verhaltensbeschreibung
  2. Datenflußbeschreibung
  3. Strukturelle Beschreibung
- Eine entity kann **mehrere** Architekturen enthalten, d.h. es können für eine Komponentenschnittstelle mehrere Beschreibungen auf unterschiedlichen Abstraktionsebenen oder verschiedenen Entwurfsalternativen eingebunden werden




```
architecture sequentiell of NAND is
begin
        process (a , b)
        begin
            if a = ‘1‘ and b = ‘1‘ then
                        y <= ‘0‘;
            else
                        y <= ‘1‘;
            end if;
        end process;
end sequentiell;
```

1. Verhaltensbeschreibung

```
architecture behavioral of NAND is
begin
            y <=  not (a and b);
end behavioral_par;
```

2. Datenflußbeschreibung

Coding Style

- Verhaltensbeschreibung

  Prozeß mit Kontrollanweisungen.

  (if, case, for ,while, ...)

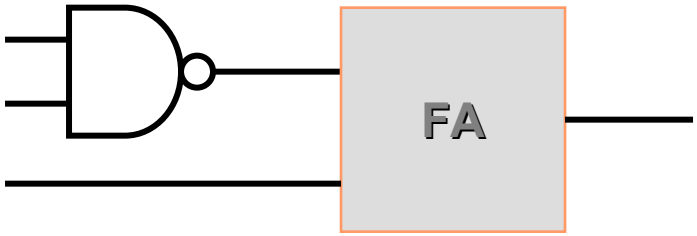
- Datenflußbeschreibung

  Beschreibung über logische Funktionen, die in VHDL definiert sind.

  (and, or, not, ...)

- Strukturbeschreibung

  Beschreibung über eine Netzliste, deren Komponenten aus einer Bauteilebibliothek stammen.



Verhaltensbeschreibung in VHDL

- Beschreibung des Verhaltens eines Modells durch die Reaktion der Ausgangssignale auf Änderungen der Eingangssignale.
- Verhalten wird ***sequentiell*** beschrieben

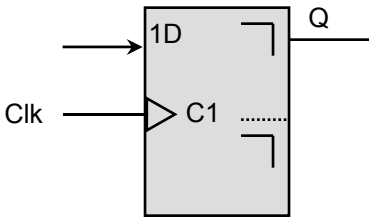
Beispiel:

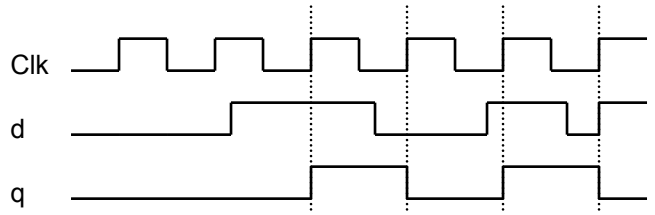
```
architecture sequentiell of D-FF is
begin
    process (d)
    begin
        if reset = '1' then
            q <= '0';
        elsif clk'event and clk = '1' then
            q <= d;
        end if;
    end process;
end sequentiell;
```

Leitet einen sequentiellen *Prozess* ein

Sensitivity-List: *Prozess* wird immer genau dann durchlaufen, wenn eine der aufgeführten Variablen ihren Wert ändert.

Alle aufgeführten Anweisungen werden sequentiell durchlaufen





Datenflußbeschreibung

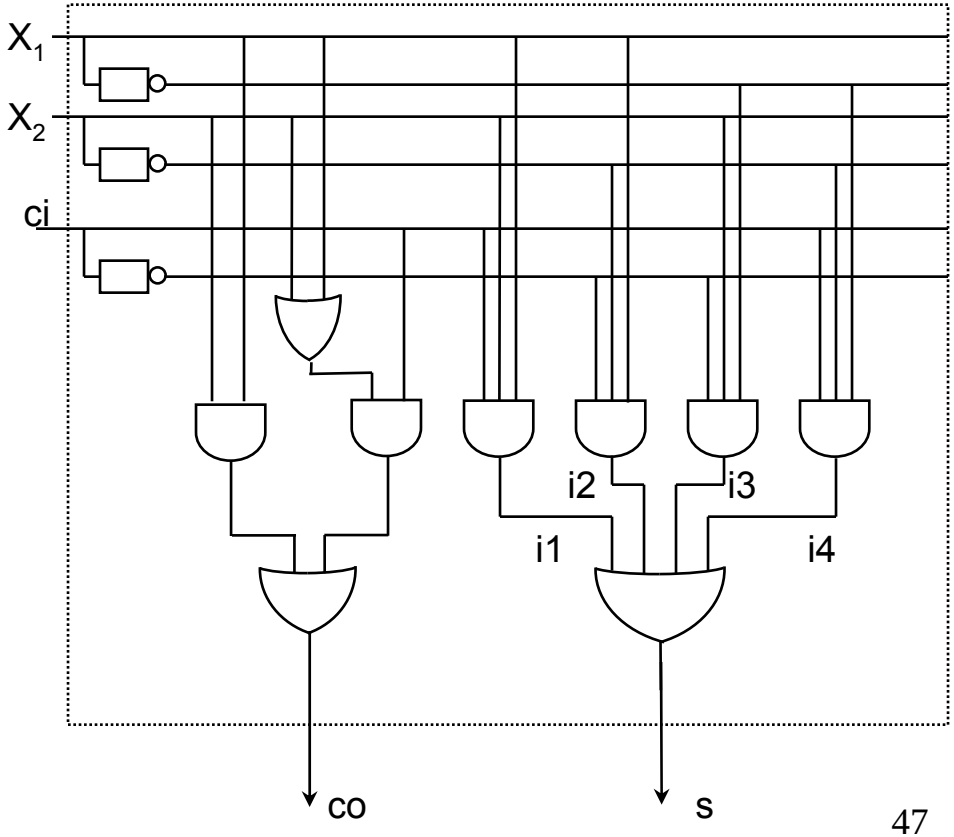
- Das Verhalten eines Modells wird durch seinen inneren Aufbau aus Standardkomponenten beschrieben
- Wertzuweisung erfolgt ***parallel***

Beispiel:

Signale sind als interne „Leitungen“ zu verstehen

Alle aufgeführten Anweisungen werden **parallel** ausgeführt

```
architecture parallel of FullAdder is

  signal x1n, x2n      : STD_LOGIC;
  signal i1, i2, i3, i4 : STD_LOGIC;
  signal cin           : STD_LOGIC;

begin
 cin <= not ci;
 x1n <= not x1; x2n <= not x2;
 i1 <= x1 and x2n and cin;
 i2 <= x1 and x2 and ci;
 i3 <= x1n and x2n and ci;
 i4 <= x1n and x2 and cin;
 co <= (x1 and x2) or (ci and (x1 or x2));
 s <= i1 or i2 or i3 or i4;
end parallel;
```

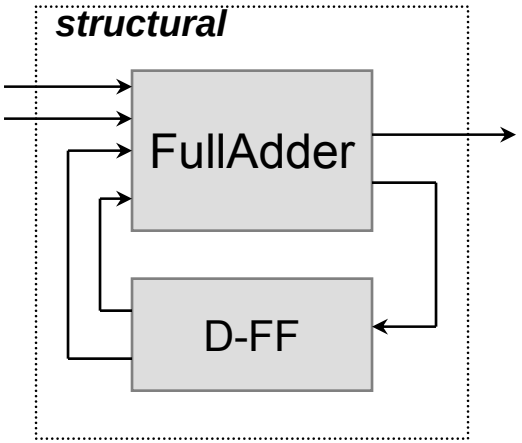


Strukturbeschreibung

- Aufbau eines Modells aus Unterkomponenten
- Die Komponenten müssen in einer gesonderten Bibliothek beschrieben sein
- Wichtig bei Elementen, die sich nicht aus einer VHDL-Beschreibung synthetisieren lassen oder ausschließlich als Hard-Makro eines ASIC-Herstellers vorliegen.

Beispiel:

```
architecture structural of SerialAdder is

  component FullAdder
    port(x1, x2, ci : in STD_LOGIC;
         s, co      : out STD_LOGIC);
  end component;

  component DFF
   port(d, clk, reset : in STD_LOGIC;
        q             : out STD_LOGIC);
  end component;

 signal FACin  : STD_LOGIC;
 signal FACout : STD_LOGIC;

begin
  FA: FullAdder port map (x1 => x1, x2 => x2, ci => FACin,
                          s => sum, co => FACout);
  FF: DFF port map (d => FACout, clk => clk, reset => reset,
                    q => FACin);
  scy <= FACin;
end structural;
```



Konfiguration (Configuration)

- Legt fest, welche der beschriebenen Architekturvarianten einer bestimmten Entity zugeordnet werden
- Zusätzlich werden Parameter (z.B. Verzögerungszeiten) an Instanzen übergeben

Beispiel:

```
configuration SerialAdder_CFG of SerialAdder is
 for structural
    for FA: FullAdder use configuration WORK.FullAdder_CFG; end for;
    for FF: DFF use configuration WORK.DFF_CFG; end for;
 end for;
end SerialAdder_CFG;
```

Das Package

- Enthält Anweisungen wie Typ- oder Objektdeklarationen und die Beschreibung von Prozeduren und Funktionen die in mehreren VHDL-Beschreibungen verwendet werden sollen

Syntax

```
package identifier is
      package_declarative_item
end package identifier;
```

Beispiel

```
package cpu_types is
constant word_size : integer := 16;
subtype word is bit_vector (word_size downto 0);
end package cpu_types;
```

- Um ein Package zu verwenden, muß es eingelesen werden. Dies geschieht mit der library-Anweisung.

Syntax

```
library package_name;
```

Ansprechen der Objekte innerhalb der Library

```
library_name.package_name. objekt_name;
```

Ein kleines Beispiel

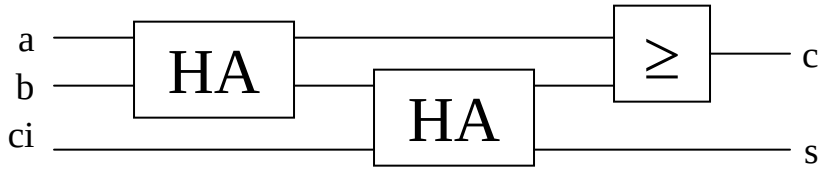
- full adder

```
ENTITY f_adder IS
PORT( a, b, ci : in bit;          -- a, b : data in, ci : carry in
s, c : out bit;                   -- c : carry out, s : sum
p, g : out bit);                  -- p : propagate, g : generate
END f_adder;
```

```
-- sample architecture for full adder
ARCHITECTURE f_adder_architecture OF f_adder IS

COMPONENT h_adder
PORT( a, b : in bit;
s, c : out bit);
END COMPONENT;

SIGNAL s1, c1, c2 : bit;

BEGIN

      ha_1 : h_adder
      PORT MAP (s=>s1, c=>c1, a=>a, b=>b);

      ha_2 : h_adder
      PORT MAP (s=>s, c=>c2, a=>ci, b=>s1);

p <= s1;
g <= c1;

c <= c1 OR c2;
END f_adder_architecture;
```

```
Configuration f_adder_CFG of f_adder is
      for f_adder_architecture
            for all:
                  h_adder use entity WORK.h_adder(h_adder_s)
            end for;
      end for;
end configuration;
```

Volladdierer aus zwei Halbaddieren



In der Konfiguration werden den **Instanzen** die verwendeten Submodule zugewiesen.

Hier: Die beiden Halbaddierer verwenden das Modul h_adder, welches sich im Arbeitsverzeichnis WORK befindet.

Zusammenfassung

Entity:

eine Schnittstelle

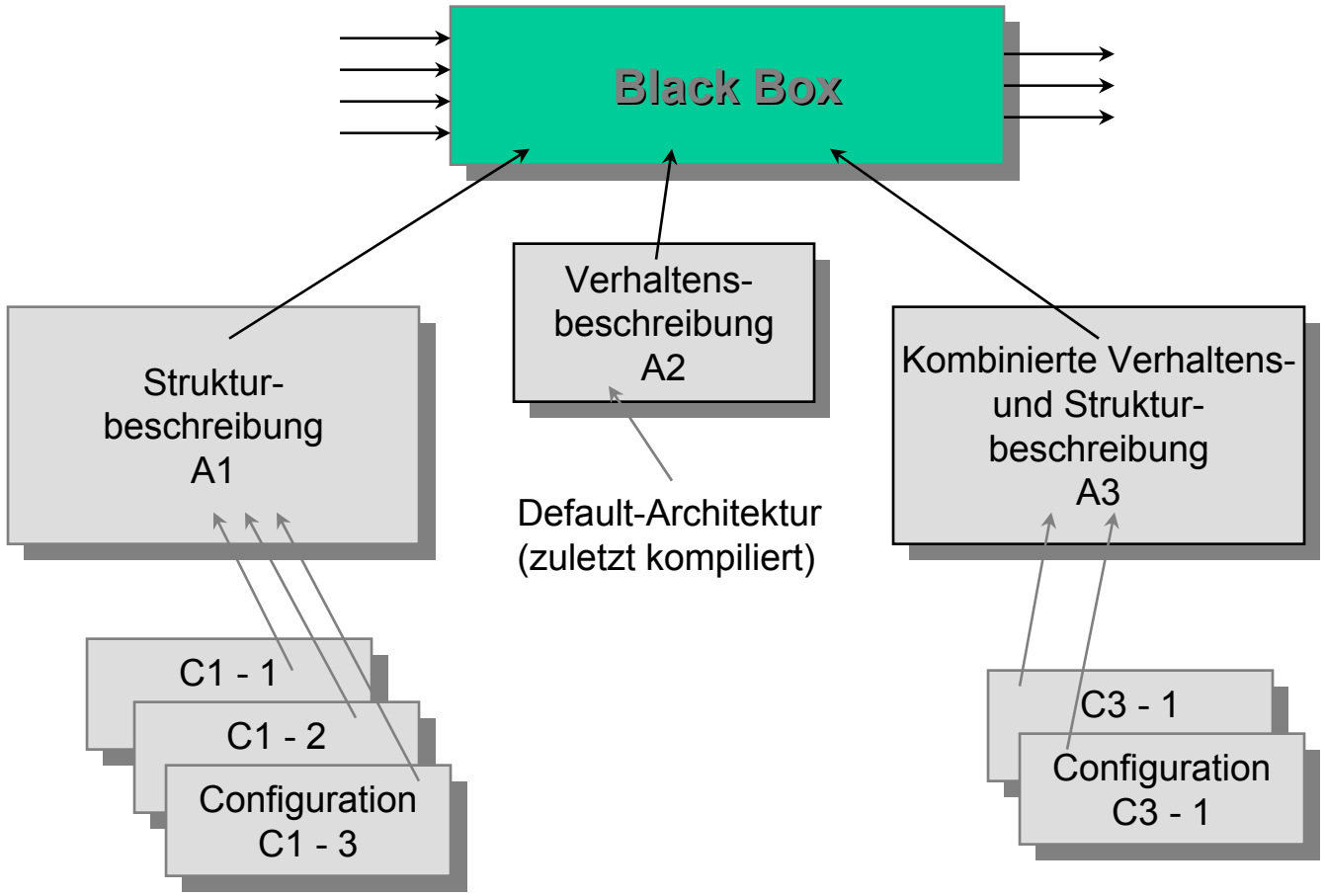
Architecture:

Verhalten oder Struktur;

eine oder mehrere Alternativen

Configuration:

beliebige Anzahl von Konfigurationen



Inhalt des zweiten Tages

- Grundelemente der Sprache VHDL
- Syntax und Struktur einer VHDL-Beschreibung
- Synthesewerkzeuge des digitalen Designflows
- Übungen

Elemente der Sprache VHDL

Kommentare: Kommentare werden bei VHDL mit zwei aufeinanderfolgenden Bindestrichen dargestellt.

Syntax: --

Zahlen: Zahlen können als Dezimalzahl oder als Zahl in Zusammenhang mit einer Basis dargestellt werden.

Syntax: decimal literal
integer[.integer][exponent]
based literal
based # based_integer[based_integer]#

Beispiel:
1, 2, 3, ...
1.3, 4.711
2#10010011# Dual

Zeichen: Zeichen werden mit einfachen Anführungszeichen beschrieben

Syntax: 'character'

Beispiel:
'F', '*', '?', ...

Zeichenketten: Zeichenketten werden mit doppelten Anführungszeichen beschrieben

Syntax: "string"

Beispiel:
"Technische Informatik"

Objekte

- Alle Daten in VHDL werden über Objekte verwaltet. Objekte besitzen einen definierten Datentyp und einen Bezeichner - den *identifier*

Man unterscheidet:

Konstanten: Wert wird nur einmal zugewiesen

Variablen: Wert kann gelesen und neu zugewiesen werden

Signale: Wert kann gelesen und verändert werden. Zusätzlich wird der zeitliche Verlauf gespeichert, so daß auch auf Werte in der Vergangenheit zugegriffen werden kann, bzw. Werte für die Zukunft vorbestimmt werden können. (x <= '0' after 10 ns;)

Beispiel:

```
constant my_constant :    bit_vector(7 downto 0):= '00000000';
variable my_variable :    bit_vector(7 downto 0):= '00000000';
signal my_signal:         bit_vector(7 downto 0):= '00000000';
```

Objektdeklarationen

Objektdeklarationen

Bevor ein Objekt in einem VHDL-Modell verwendet werden kann, muß es deklariert werden.

Syntax
object_class var_name **:** *var_type := init_value*;

Beispiel:
variable error : **integer** := 0;
signal net1 : **bit**;

Einen Sonderfall bei der Deklaration bilden ganzzahlige Laufvariablen in Schleifenkonstrukten. Diese Variablen
müssen nicht explizit deklariert werden bevor sie benutzt werden.
Objekte werden über ihren Namen (oder Alias) referenziert.

Beispiel:

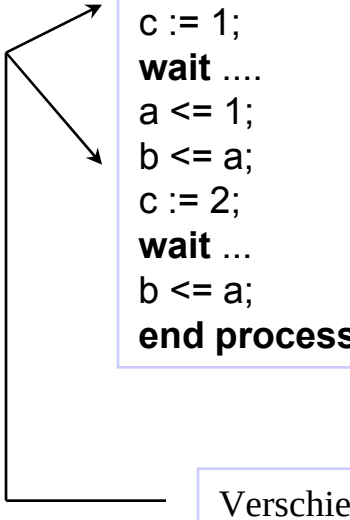
```
signal x, y,z:          std_logic_vector(7 downto 0)
begin
    for  i  in 1 to 1000 generate
            DFF_instance : DFF_socket;
            port map (r_in(i), clk, r_out(i));
    end generate;
end;
```

Signale versus Variable

Signale erhalten nach einer Zuweisung ihren neuen Wert erst nach einem Δt. Variablen erhalten nach einer Zuweisung den neuen Wert sofort.

Beispiel:

```
signal a, b : bit;
variable c  : integer;
process
begin
a <= 0;
c := 1;
wait ....
a <= 1;
b <= a;                    -- b = 0!
c := 2;                    -- c = 2!
wait ...
b <= a;                    -- b = 1!
end process;
```



Verschiedene Zuweisungen beachten!!!!!!

Signale dienen dazu, Daten zwischen parallel arbeitenden Modulen auszutauschen. Verschiedene Module können auf das gleiche Signal schreiben. Dadurch können Busleitungen modelliert werden. Signale entsprechen, vereinfacht gesprochen, Verbindungsleitungen in elektronischen Schaltungen.

Variablen sind (prinzipiell) nur innerhalb **eines** Prozesses gültig.

Datentypen und Typdeklarationen

Vorbestimmte Datentypen:

Die Sprache VHDL kennt zwei vordefinierte Datentypen:

- Real -- Floating Point
- Integer -- Fix Point

Beispiel:

```
var_1   : real;
var_2   : integer;
```

Datentypen der IEEE Standard-Bibliothek:

- bit, bit_vector, std_logic_vector
- boolean
- string
- character

Beispiel:

```
var_3   : bit_vector (3 downto 0);
var_4   : std_logic_vector (3 downto 0);
```

Benutzerdefinierte Datentypen:

type *my_type* **is integer**

type *new_typ* **is range** *range_low* **to** *range_high*

type *new_typ* **is range** *range_high* **downto** *range_low*

Beispiel:

```
type my_type is range 0 to 99 of integer
```

Weitere Datentypen

Aufzählungstype

n: Objekte vom Aufzählungstyp können nur bestimmte, in einer Liste angegebene Werte, annehmen

Syntax
type *enum_type* **is (***v 1* {,*v 2* }**)**;

Beispiel:
type boolean **is** (false, true);
type bit **is** (’0’, ’1’);
type character **is** (...); -- alle 128 Zeichen
type state **is** (s1, s2, s3);

Physikalische Typen:

Physikalische Zahlen bestehen aus einem Zahlenwert und einer Einheit.
In der Typdeklaration eines phsikalischen Types kann die Einheit festgelegt werden und - wenn nötig - auch noch Ableitungen.

Syntax
type *phys_type* **is range** *range_low* **to** *range_high*
units
base_unit;
{ *derive_unit* = *multiplier* * *unit*; }
end units;

Beispiel
type capacitance **is range** 0 to 1E12
units ff;
pf = 1000 ff;
nf = 1000 pf;
mf = 1000 nf;
f = 1000 mf;
end units;

Weitere Datentypen

Feldtypen

Vektoren können als eindimensionales Feld aufgefaßt werden.

Syntax
type *vector_type* **is array** *index_constraint* **of** *base_typ*;
Als Index index_constraint können beliebige diskrete Zahlentypen (ganzzahlige Zahlen, Aufzählungen,...) verwendet werden.

Beispiel
type a **is array** (1 to 10) **of character**;

Mehrdimensionale Felder

Matrizen können als zweidimensionales Feld aufgefaßt werden.

Syntax
type *vector_type* **is array** *index_constraint* **of** *base_typ*;

Beispiel

type vector **is array** (1 to 10) **of integer**;
type matrix1 **is array** (1 to 5) **of array**;
type matrix2 **is array** (1 to 10, 1 downto 100) **of integer**;

Variablendeklaration

Beispiel 1:

```
variable m1                         : matrix1;
variable m2                         : matrix2;
variable x              : integer;
variable v              : vector;

-- Zuweisungen zu den einzelnen Elementen

x := m2(1,1);           -- korrekt
x := m1(1,1);           -- ERROR
x := m1(1)(1);          -- korrekt
v := m1(1)              -- korrekt
```

Beispiel 2:

```
signal my_sig           : std_logic;
constant my_const       : std_logic_vector(7 downto 0);
type my_typ              is          (idle, fetch decode,
execute, wb);
signal my_state         : my_type;

type date is record
      tag               : integer range 1 to 31;
      monat             : monatsname;
      jahr              : integer range 0 to 2500;
end record;

type lieferung is record
      datum             : date;
      anzahl            : positive;
      ware              : sorte;
end record;

signal paket                        : lieferung;
variable month          : monatsname;
month                   := paket.datum.monat;
```

Operatoren

Logische Operatoren

In VHDL sind folgende logischen Operatoren definiert:

- NOT
- AND
- NAND
- OR
- XOR
- XNOR

Beispiel:

```
a := NOT (x AND y AND z)        -- NAND3
b := x OR y OR z                -- OR3
c := y XNOR z                   -- XNOR2
```

Vergleichsoperatoren

Neben den logischen Operatoren werden folgenden Vergleichsoperatoren definiert:

- = : gleich
- /= : ungleich
- < : kleiner
- >= : kleiner gleich
- > : größer
- <= : größer gleich

Das Schlüsselwort "process" leitet eine sequentielle Verarbeitung ein

Variablen sind nur innerhalb eines Prozesses gültig

Beispiel:

```
beispiel1 : process (a,b,c)
        variable x      : real;
        begin
                if a /= b then
                        x := 5;
                end if;
        end process;
```

Operatoren

Arithmetische Operatoren

In VHDL sind folgende arithmetische Operatoren definiert:

- + -- Addition
- - -- Subtraktion
- & -- Zusammensetzen
- * -- Multiplikation
- / -- Division
- MOD -- Modulo-Operator
- REM -- Remainder-Operator
- ** -- Exponentation
- ABS -- Absolutwertbildung

Schiebe- und Rotieroperatoren

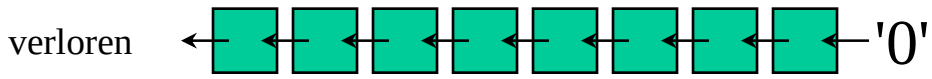
VHDL93 bietet sechs Schiebe- und Rotieroperatoren die auf Vektoren angewandt werden

- SLL -- Schiebe logisch links
- SRL -- Schiebe logisch rechts
- SLA -- Schiebe arithmetisch links
- SRA -- Schiebe arithmetisch rechts
- ROL -- Rotiere links
- ROR -- Rotiere rechts

Bei "logischen Schiebeoperationen" um eine Stelle geht jeweils das linke bzw.. rechte Vektorelement verloren. Auf der gegenüberliegenden Seite wird der Vektor mit dem Initialisierungswert des Basistyps aufgefüllt.

Beispiel:

```
signal x, y, z:        std_logic_vector (7 downto 0)
begin
    y <= SLL x;
    z <= x and y;
end;
```



Attribute

Typen und Objekte in VHDL können neben ihrem eigentlich Wert oder der Bedeutung noch zusätzliche Informationen - sogenannte Attribute - besitzen.
Attribute werden über die Notation ’ angesprochen.
Es existieren eine Reihe von vordefinierten Attributen:

Auswahl einiger Attribute:

Name	*Funktion*
T’left	Linke Grenze von T
T’right	Rechte Grenze von T
T’low	untere Grenze von T
T’high	obere Grenze von T
T’pos(x)	Nummer der Position von x in T
T’val(n)	Wert der Position n in T
T’pred(x)	Wert des Elementes welches sich eine Position unter dem Element x befindet.
T’succ(x)	Wert des Elementes welches sich eine Position über dem Element x befindet.
T’event	Änderung von T im aktuellen Simulationszyklus (Signalattribut)
T’stable	Keine Flanke im aktuellen Simulationszyklus (Signalattribut)

Nebenläufige Anweisungen

Signalzuweisung
Nebenläufige Signalzuweisungen sind nur außerhalb einer Prozeßbeschreibung möglich:
Aufeinanderfolgende Signalzuweisungen der Form:

Syntax
signal_name <= value **AFTER** time_value **ns**;
werden normalerweise gleichzeitig, innerhalb eines Prozesses aber sequentiell ausgeführt.

Beispiel:

```
signal a, b, c, d, e, f:        std_logic;
begin
        a <= x and y;
        b <= x or z;
        c <= u xor v;
sequential ; process (d, e, f)
begin
        a <= e and f;
        b <= d or e
end;
```

Die Signale "a", "b" und "c" bekommen **gleichzeitig** ihren Wert zugewiesen.

Die Signale "a" und "b" bekommen ihren Wert **nacheinander** zugewiesen.

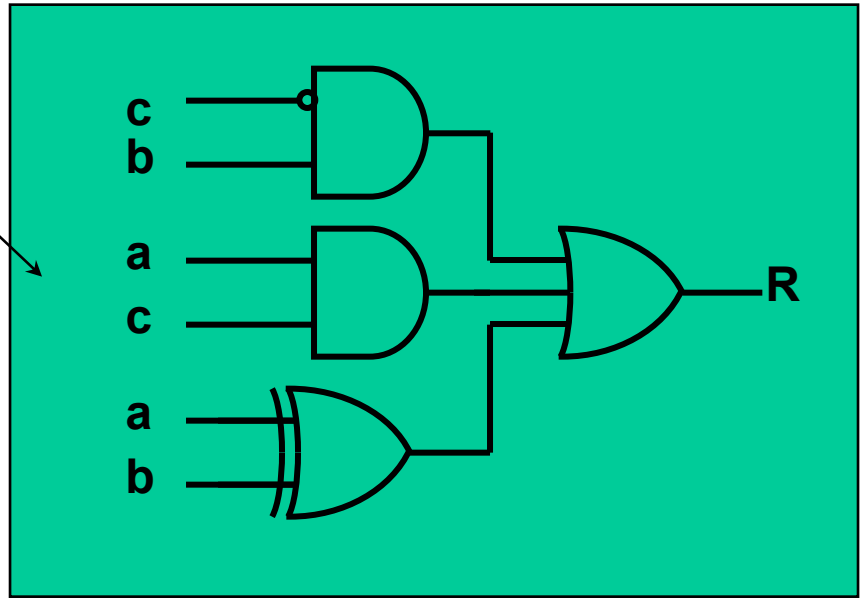
Nebenläufige Anweisungen 1

Beispiel:

```
architecture kombi of demo is
signal x, y, z, R:                std_logic;
begin
        x <=  a xor b;
        y <= b and not c;
        z <= a and c
        R <= x or y or z;
end;
```

Die Aufgabenrealisierung ist auf der Ebene der "architecture" ***nebenläufig***!

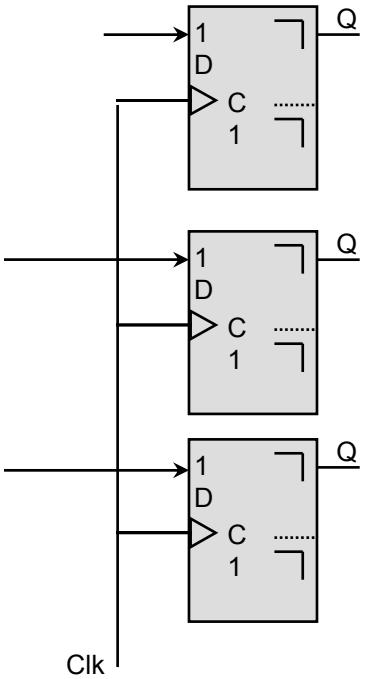
=> "concurrent statement"



Nebenläufige Anweisungen 2

- VHDL bietet die Möglichkeit Submodule in die Beschreibung einzufügen.
- Diese Komponenteninstanzen liegen **nebeneinander** und kommunizieren über Signale miteinander.
- Alle Komponenten arbeiten **parallel**.
- Jede Instanz arbeitet selbständig und führt eine Aufgabe aus.

```
ARCHITECTURE structural OF n_bit_registe

  COMPONENT d_ff_sock
    PORT (d, clk : IN bit; q : OUT bit
  END COMPONENT

BEGIN

  reg : FOR i IN n-1 DOWNTO 0 GENERA

    d_ff_instance: d_ff_socket                    -- n-fache Instantiie
      PORT MAP (reg_in(i), clk, reg_out(i

  END GENERATE

END structural
```



Allgemeines zu Prozessen

Prozesse:

Prozesse dienen als Umgebung für **sequentielle** Befehle. Prozesse selbst gelten als **nebenläufige** Anweisungen, d.h. innerhalb einer Architektur können mehrere Prozesse definiert werden, die gleichzeitig aktiv sind. Prozesse werden durch die folgenden Elemente aktiviert oder gestoppt:

- **Sensitivity-Liste**

Ein Prozeß wird bei der Initialisierung der Simulation einmal durchlaufen und danach erst wieder aktiviert, wenn sich ein oder mehrere Signale in der Sensitivity-Liste ändern.

- **Wait-Anweisung**

Bei der Initialisierung wird der Prozeß bis zur ersten Wait-Anweisung durchlaufen. Danach wird der Prozeß erst dann wieder aktiviert, wenn die Bedingung der Wait-Anweisung erfüllt ist.

- **Prozesse ohne Wait-Anweisung oder Sensitivity-List**

Diese Prozesse werden ständig zyklisch durchlaufen.

- Prozesse entsprechen Programmen konventioneller Programmiersprachen wie C
- Ein Prozeß tauscht nur über Signale Informationen mit seiner Umwelt aus
- Einem Prozeß wird ein Zeit- bzw. Kausalitätsverhalten zugeordnet

Eigenschaften von Prozessen

- Sequentielle Anweisungen
  - Definieren Algorithmen
  - werden ***nacheinander*** abgearbeitet
  - können "normale" Software repräsentieren
  - sind der Inhalt vom Prozeß-Anweisungsteil
  - werden ohne Zeitverzug ausgeführt
- Ein Prozeß entspricht einer Hardwarekomponente die auf Eingangsänderungen reagiert
- Ein Prozeß ist auf diese Eingänge = Signale *sensitiv*
- Er ist nur aktiv, wenn sich diese Signale ändern

Syntax von Prozessen

Syntax

process [(sig1 [, sig2, ...])]
    declaration of
        types, subtypes
        constants, files
        variables, functions
    definition of
        functions,
        attributes
begin
    sequential statements
end process;

process
    declaration of
        types, subtypes
        constants, files
        variables, functions
    definition of
        functions,
        attributes
begin
    sequential statements
    wait *expression*;
    sequential statements
end process;

Beispiele:

architecture example **of** and2 **is**
begin
p: **process** (A, B)
variable x: **integer;**
begin
x := A **and** B;
end process;
end example;

Eine weitere Möglichkeit der Beschreibung besteht in der Erzeugung eines Prozesses mit Sensitivity-Liste und anschließender Abfrage über ein if-Statement:

Beispiel
process (clk, rst)
if (rst = '1') **then**
reset_actions
elsif (clk'event and clk = '1') **then**
normal_processing
end if;
end process;

Aktivierung über wait-Anweisung

```
ARCHITECTURE sequential_2 OF latch IS

BEGIN

  q_assignment : PROCESS
  BEGIN

    IF clk = '1' THEN
      q <= d ;
    END IF ;

    WAIT ON d, clk ;      -- entspricht "sensitivity-list"

  END PROCESS q_assignment ;

END sequential_2 ;
```

Es wird solange gewartet, bis eine Änderung auf dem Signal d oder clk stattfindet.

Es ist NICHT möglich zeitliches Verhalten in eine synthetisierbare VHDL-Beschreibung einzubauen.

Konstrukte wie beispielsweise „wait for 10 ns“ können nicht auf eine Technologie abgebildet werden!

Aktivierung über Sensitivity-Liste

```
ENTITY mult IS

  PORT (a, b : IN integer := 0; y : OUT intege

END mult

--------------------------------------------------------------------

ARCHITECTURE number_one OF mult

BEGIN

  PROCESS (a, b
    VARIABLE v1, v2 : integer :=
  BEGIN

    v1 := 3 * a + 7 * b    -- Variablenzuweisun
    v2 := a * b + 5 * v1   -- Variablenzuweisun
    y  <= v1 + v2 ;        -- Signalzuweisung (Ports sind interne Signa

  END PROCESS

END number_one
```

Die Prozeß wird nur dann durchlaufen, falls eine Änderung auf den Signalen der Sensitivity-Liste (hier "a" und "b") stattfindet.

Sequentielle Anweisungen

Innerhalb eines Prozesses werden alle Anweisungen sequentiell ausgeführt. Innerhalb der Sprache VHDL sind folgende sequentielle Anweisungen definiert:

- Signalzuweisung
- Variablenzuweisung
- Report
- Wait
- if - elsif - else - end if
- case - when - end case
- loop - end loop
- while - end loop
- for - end loop

case

Syntax z.B. für Endliche Automaten

```
case expression is
   {when choice =>
sequence_of_statements}
   [when others =>
sequence_of_statements]
end case;
case state is
   when s1 => state <= s2
   when s2 => state <= s3
   ...
```

if - elsif - else - endif

Syntax

```
if condition then
   sequence_of_statements
elsif condition then
   sequence_of_statements}
[else
   sequence_of_statements]
end if;
```

Report
Um eine Ausgabe zu erzeugen, kann das Schlüsselwort *report* genutzt werden.
Syntax
report "Message";

Sequentielle Anweisungen

loop
Syntax

loop
sequence_of_statements
end loop;

Beispiele:

```
FUNCTION mult (a,b : int_vector (1 TO 8)) RETURN int_vector IS
  VARIABLE count : integer ;
  VARIABLE result : int_vector (1 TO 8);
 BEGIN
  count := 0;
  LOOP
   count := count + 1;
   result(count) := a(count)*b(count);
   EXIT WHEN count = 8;
  END LOOP;
  RETURN result;
 END mult ;
```

while
Syntax

while *expression* **loop**
sequence_of_statements
end loop;

```
FUNCTION mult (a,b : int_vector(1 TO 8)) RETURN int_vector IS
  VARIABLE count : integer ;
  VARIABLE result : int_vector (1 TO 8);
 BEGIN
  count := 0;
  WHILE count < 8 LOOP
   count := count + 1;
   result(count) := a(count)*b(count);
  END LOOP;
  RETURN result;
 END mult ;
```

Sequentielle Anweisungen

for

Syntax

for *item* **in** *index_low* **to** *index_high* **loop**
sequence_of_statements
end loop;

Beispiel:

```
FUNCTION mult   (a,b : int_vector(1 TO 8)) RETURN int_vector IS
   VARIABLE result : int_vector (1 TO 8);
 BEGIN
   FOR count IN 1 TO 8 LOOP
     result(count) := a(count)*b(count);
   END LOOP;
   RETURN result;
 END mult;
```

Synthesewerkzeuge des digitalen Designflows

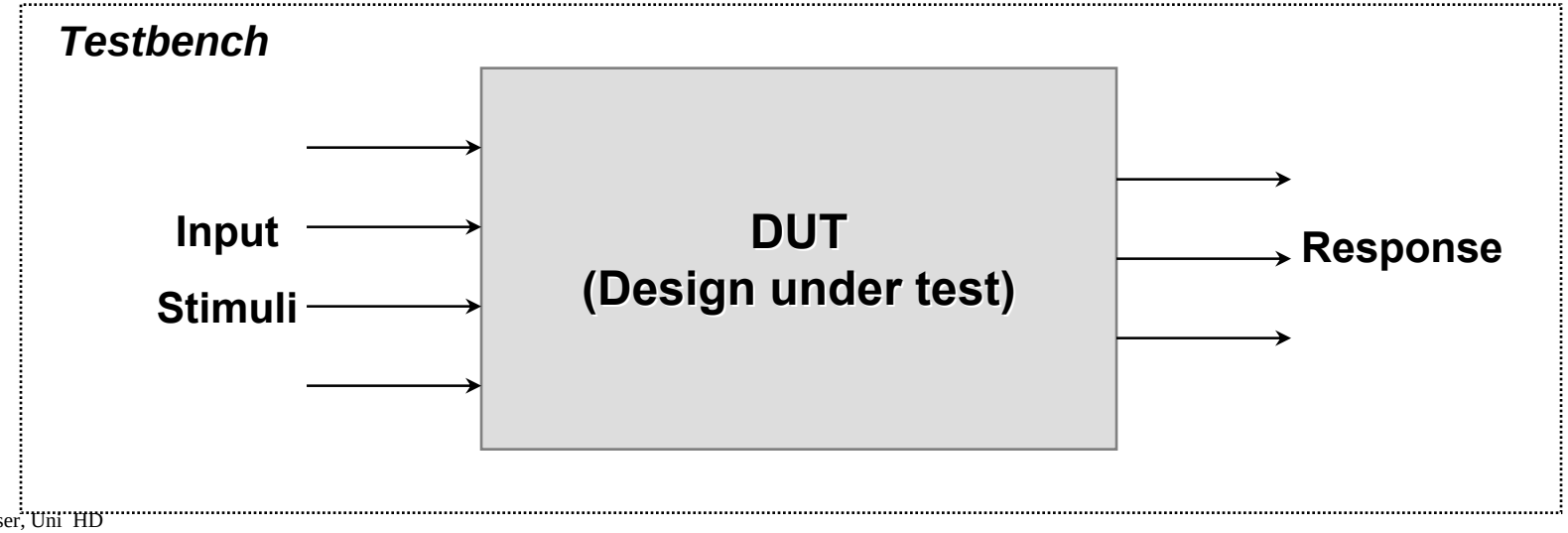
Falk Lesser, Volker Lindenstruth
Institut für Hochenergiephysik

Compilieren des VHDL-Modells

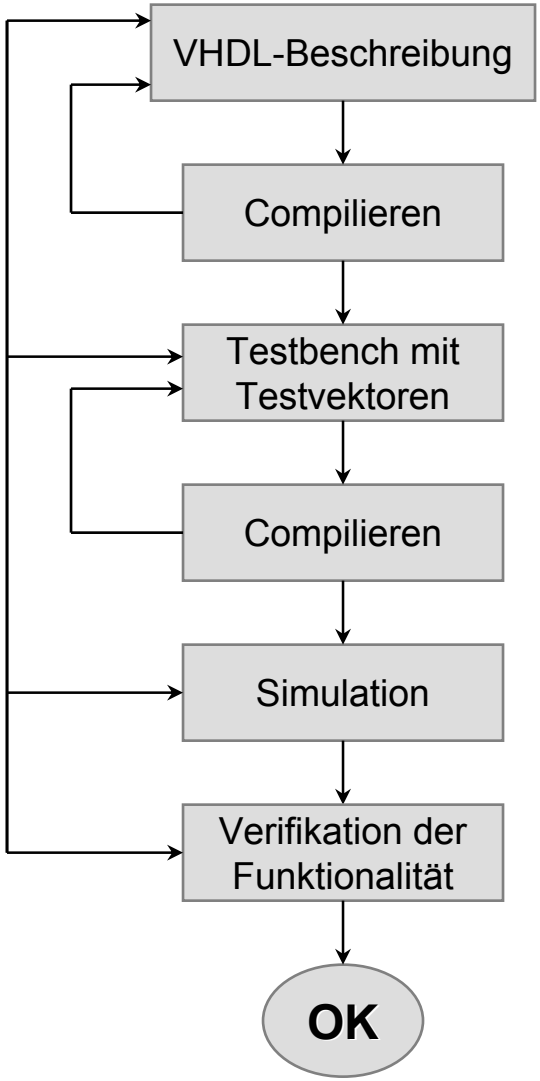
- Bevor das Modell simuliert werden kann, muß es mit Hilfe des VHDL-Compilers übersetzt werden.
- Zum Starten des Compilers ist an der Eingabeaufforderung der Befehl: ***vhdlan [option] dateiname.vhd*** einzugeben.
- Optionen:
  - -*spc* bzw. -*spc_elab* prüfen bereits während des Compilierens die Syntetisierbarkeit der VHDL-Beschreibung.
- Der Compiler überprüft den Sourcecode auf syntaktische Korrektheit.
- Der Compiler legt die erzeugten Simulationsdateien (*dateiname.sim* und *dateiname.mra*) im Arbeitsverzeichnis (WORK) ab.

Testen und Simulation

- Der Test bzw. die Simulation einer HDL-Beschreibung erfolgt in einer Testumgebung (***Testbench***)
- Eine Testbench enthält folgende Module:
  - Eine HDL-Beschreibung, in der die zu untersuchende Beschreibung eingebunden ist
  - Einen Stimulus-Generator bzw. einen Satz von Testvektoren, mit der die Beschreibung (Schaltung) getestet wird.



Simulationsablauf



VHDL-Beschreibung des entworfenen Designs

Die Testbench enthält die VHDL-Beschreibung des DUT als Instanz, sowie die Testvektoren

Die Funktionalität des Entwurfs wird im Waveformviewer überprüft. Dafür werden die entsprechenden Ports und internen Signale mit dem Hierarchie-Browser ausgewählt

Verläuft die Simulation erfolgreich, kann noch keine konkrete Aussage über das endgültige Verhalten gegeben werden, da die Gatterlaufzeiten der Zieltechnologie noch unberücksichtigt sind.

Aufbau einer Testbench

Beispiel:

```
entity OR2_tb is
end OR2_tb;

architecture RTL of OR2_tb is
signal x,y,z        : std_logic;

component OR2
port (    a,b               : in std_logic;
          c                 : out std_logic);
end component;

begin

instance1: OR2
prot map( x => a, y => b; z => c)


stimulus_generation: Process(A, B)
begin
x <= '0'; y <= '0'; wait for 10 ns
x <= '0'; y <= '1'; wait for 10 ns
x <= '1'; y <= '0'; wait for 10 ns
x <= '1'; y <= '1'; wait for 10 ns
wait;

end process;

configuration OR2_tb_CFG of OR2_tb is
for RTL
          for Instance1
                OR2 use entity WORK.OR2(RTL)
          end for;
end for;
end configuration;
```

Die entity einer Testbench bleibt leer, da nach Außen keine Ports notwendig sind.

Das Modul, dessen Funktionalität überprüft werden soll wird als Komponente in die Testbench eingebunden.

Das Modul, dessen Funktionalität überprüft werden soll wird als Komponente in die Testbench eingebunden.

Die Testvektoren werden in einem Prozeß eingebunden. Über die *wait*-Anweisung wird der zeitliche Verlauf der Simulation festgelegt.

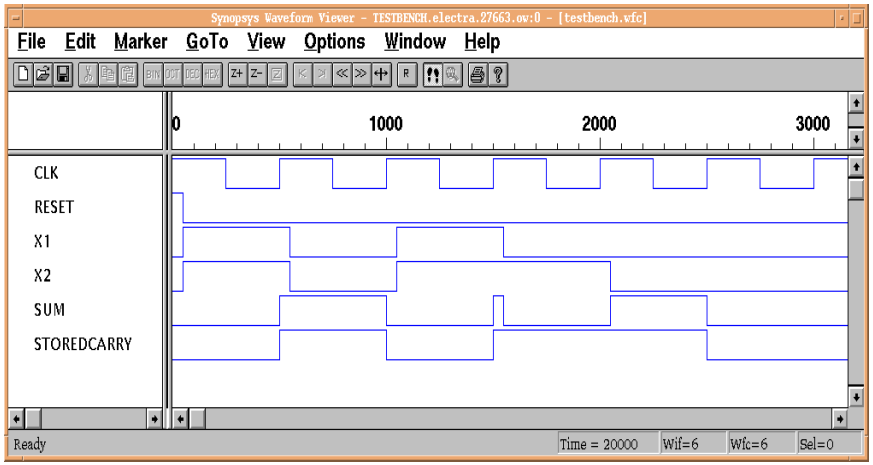
In der Konfiguration wird der Instanz das kompilierte Modul aus dem Arbeitsverzeichnis zugewiesen.

Simulation digitaler Schaltungen

- Simulation auf hoher Abstraktionsebene ermöglicht eine frühzeitige Verifikation des Entwurfs, während die spätere Simulation auf Gatterebene (basierend auf den gleichen Simulationswerkzeugen) eine hohe Entwurfssicherheit garantiert.
- In der Simulation wird die Funktion und das Zeitverhalten der entworfenen Schaltung mit der Spezifikation verglichen.
- Nach der Simulation erfolgt die Synthese, wodurch die technologieunabhängige Beschreibung auf eine Zieltechnologie abgebildet wird.

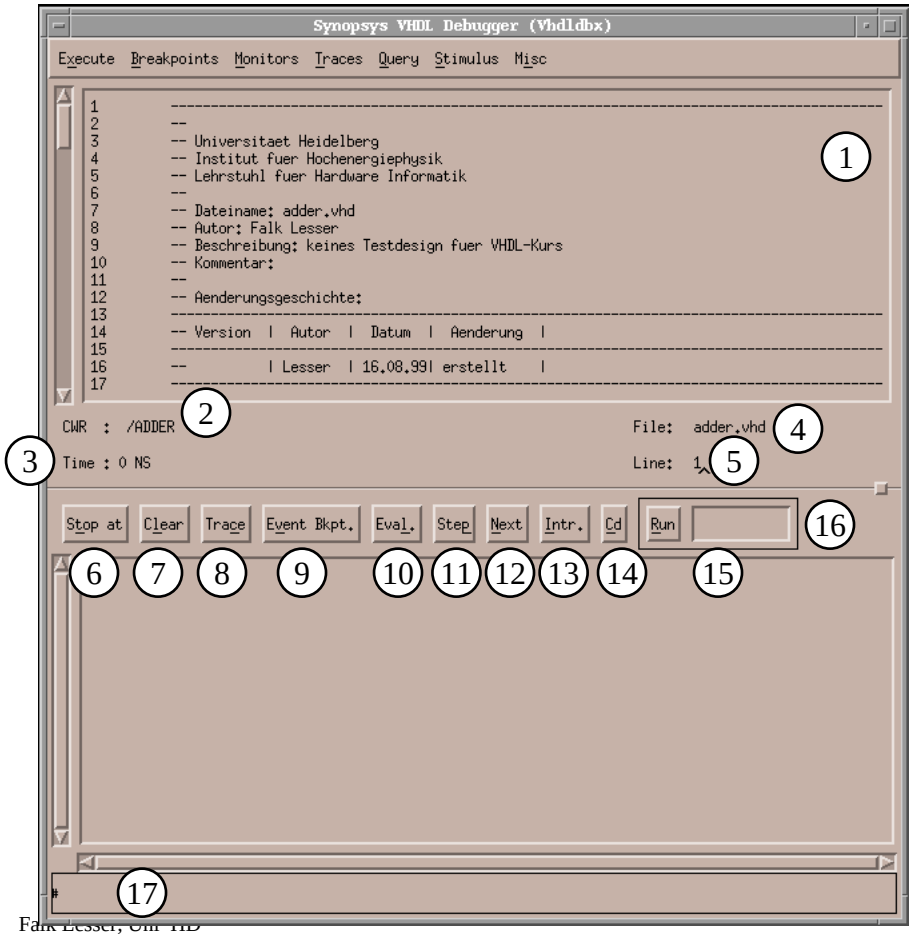
Beispiel:

- Simulation erfolgt in Testumgebung
- Generierung der Stimuli
- VHDL-Modul ohne Schnittstelle nach Außen
- Anhand der erzeugten Timing-digramme wird die Funktionalität überprüft.



Der VHDL-Simulator/Debugger

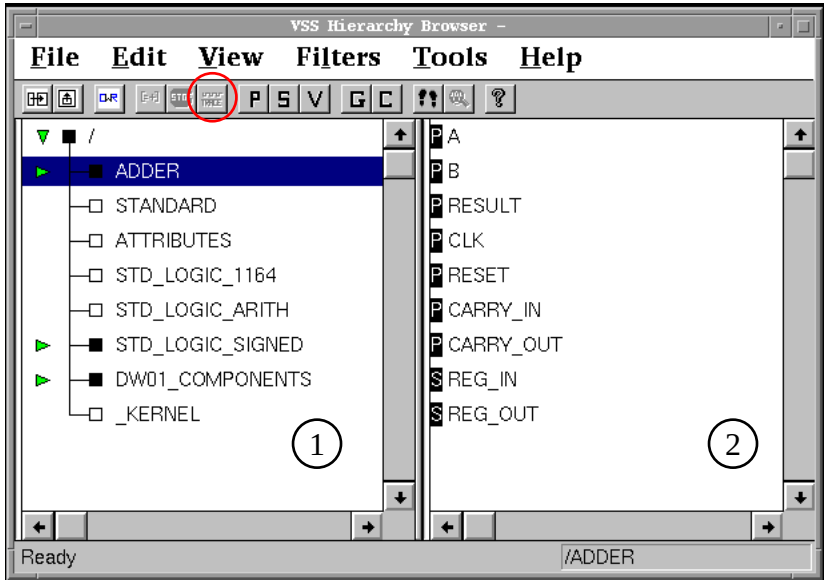
- Wurden alle VHDL-Dateien fehlerfrei compiliert, so kann der Simulator mit dem Befehl: ***vhdldbx*** aufgerufen werden.



1. Code-Fenster, Anzeige des eingelesenen VHDL-Codes
2. aktuelle Hierarchieebene
3. abgelaufene Simulationszeit
4. aktuelle VHDL-Datei
5. aktuelle Position im VHDL-Code
6. einsetzen eines Stop-Monitors in aktuelle Codezeile
7. löschen des Traces in der aktuellen Code-Zeile
8. Anzeige des selektierten Signals (Variablen) im Waveformviewer
9. Setzt einen Unterbrechungspunkt an der aktuellen Code-Position
10. Ausgabe des Wertes eines selektierten Ausdrucks
11. Einzelschrittausführung
12. Einzelschrittausführung innerhalb der Hierarchie
13. Unterbrechung der aktuellen Simulation
14. Wechsel der Hierarchieebene
15. Start der Simulation
16. Begrenzung der Simulationszeit
17. Kommando-Eingabefenster

Der Hierachy-Browser

- Der Hierachy-Browser zeigt sämtliche Ebenen eines Designs in einer Baumstruktur an und ermöglicht, über alle Hierarchieebenen hinweg, die Auswahl der zu analysierenden Signale.



1. Darstellung der Hierarchieebenen des zu Simulierenden Designs. Durch einen Mausklick auf die grünen Pfeile wird die nächste Hierarchieebene eingeblendet.
   Mit ***Trace*** werden die markierten Signale in den ***Synopsys-Waveform Viewer*** übernommen.

2. Darstellung der Signale innerhalb der ausgewählten Hierarchieebene.

   Die Buchstaben vor den Signalnamen geben die Funktion innerhalb der aktuellen Hierarchie an:

   P port

   S signal

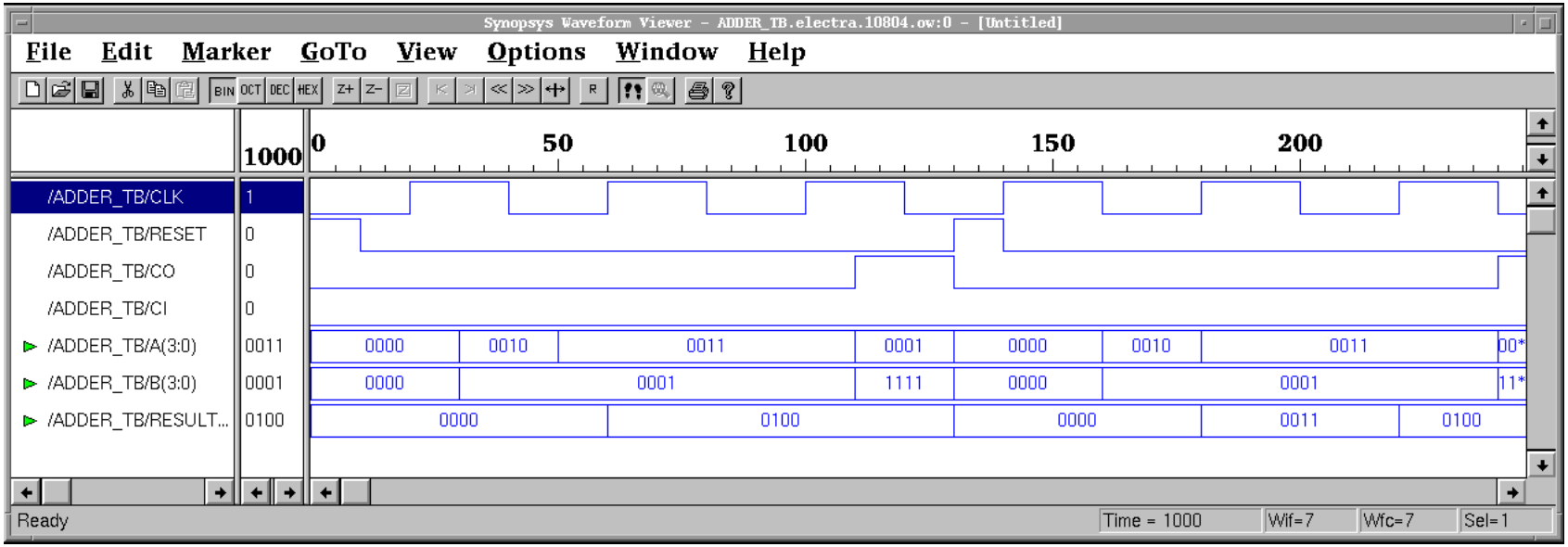
   V variable

   G generic

   C constant

Der Synopsys Waveform Viewer

- Mit Hilfe des Waveform Viewers werden die ausgewählten Signale über das Spektrum der Simulationszeit graphisch dargestellt.



- Im linken Abschnitt sind die ausgewählten Signale aufgeführt. Im rechten Teil befinden sich die Waveforms.

Einige VHDL-Konventionen

- Alle Dateien tragen die Dateiendung ".vhd"
- Jede VHDL-Datei beinhaltet genaue eine Entity, Architecture und Configuration.
- Die VHDL-Dateien tragen die Namen der zugehörigen Entities
- Für die Architekturen sind vorzugsweise die Bezeichnungen "RTL" für Beschreibungen auf Register-Transfer-Ebene, "behav" für Verhaltensbeschreibungen und "structural" für Strukturbeschreibungen zu verwenden.
- Die Konfiguration trägt den Namen der Entity mit dem Zusatz "_CFG"
- Alle Takt-Signale tragen die Bezeichnung "clk"
- Invertierte oder low-aktive Signale werden durch den Zusatz "_n" gekennzeichnet
- Signale vom Typ *buffer* werden nicht verwendet

Einige VHDL-Konventionen

- Um die Simulations- und Syntheseergebnisse möglichst einfach nachvollziehen zu können, ist es wichtig, alle ausgeführten Schritte exakt zu dokumentieren; dies geschieht am einfachsten mit Hilfe von Skript-Dateien, über die der Entwurf gesteuert wird.
- Grundsätzlich gilt:

  Synthetisierbarer VHDL-Code ist **„einfach“** und enthält keine verschachtelten Schleifenkonstrukte und umständlichen Spagetticode.

  Die zu implementierende Architektur sollte **vor** der Beschreibung in VHDL bereits bekannt sein.

  Think Hardware !!!!!!!

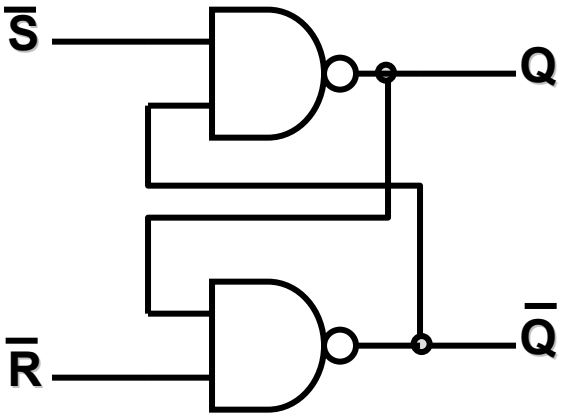
Zusammenfassung

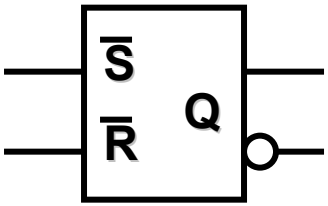
- Eine VHDL-Beschreibung ist die textuelle Beschreibung von Hardwarekomponenten
- Das Syntheseergebnis ist bei einer **einfachen** Beschreibung oft optimal
- Vor der Synthese wird die Schaltung getestet. Dadurch kann die Funktionalität der Schaltung überprüft werden
- Für den Test der Schaltung wird das Modell in eine Testbench eingebettet

Inhalt des dritten Tages

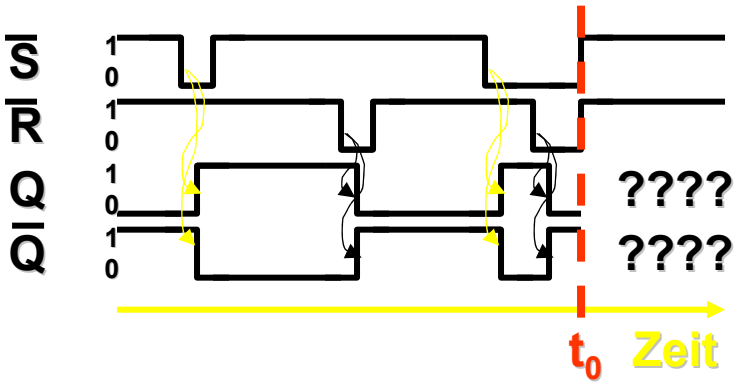
- Einführung in die Technologie synchroner Logik
- Synthesegerechte Beschreibung in VHDL
- VHDL-Beschreibungen elementarer Grundelemente
- Einführung in die Synthesetools
- Technologie von FPGAs
- Übungen

Das S/R Latch



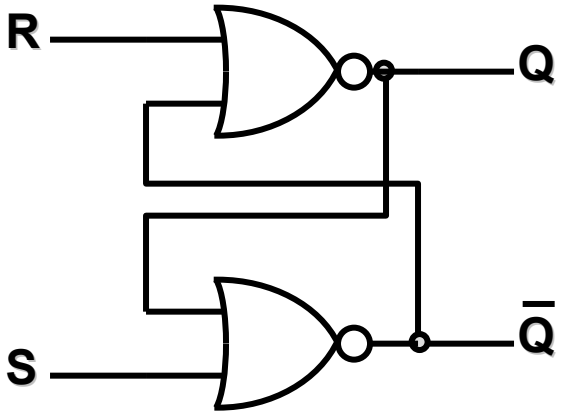


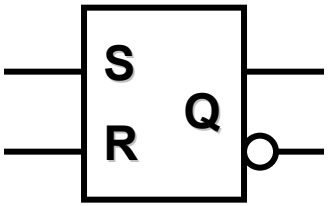
S	R	Q	/Q
1	1	Q	/Q
1	0	0	1
0	1	1	0
0	0	**1**	**1**



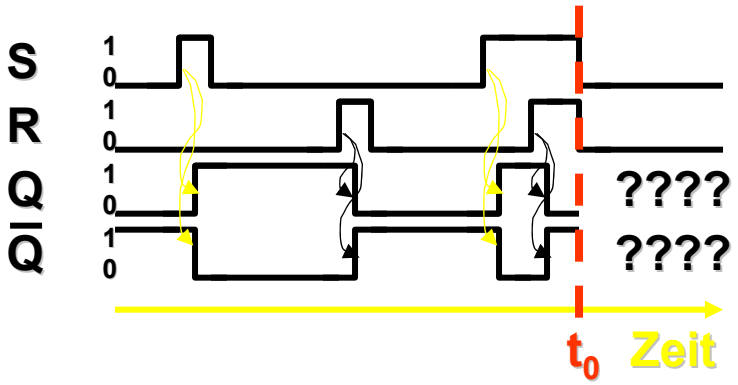
➔ **Schaltnetz hat Erinnerung des letzten Set/Reset Pulses**

Das S/R Latch



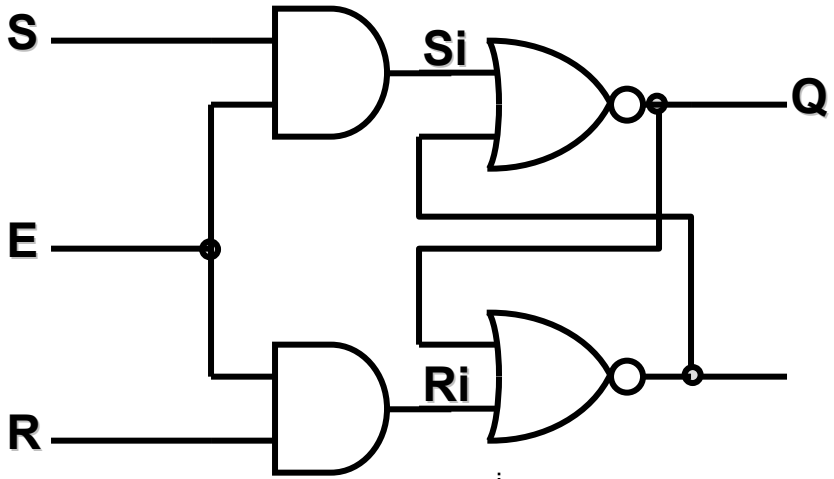


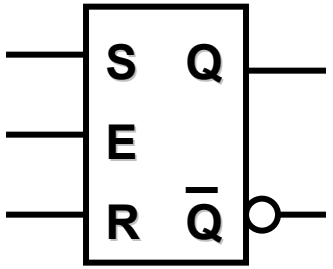
S	R	Q	/Q
0	0	Q	/Q
0	1	0	1
1	0	1	0
1	1	**0**	**0**

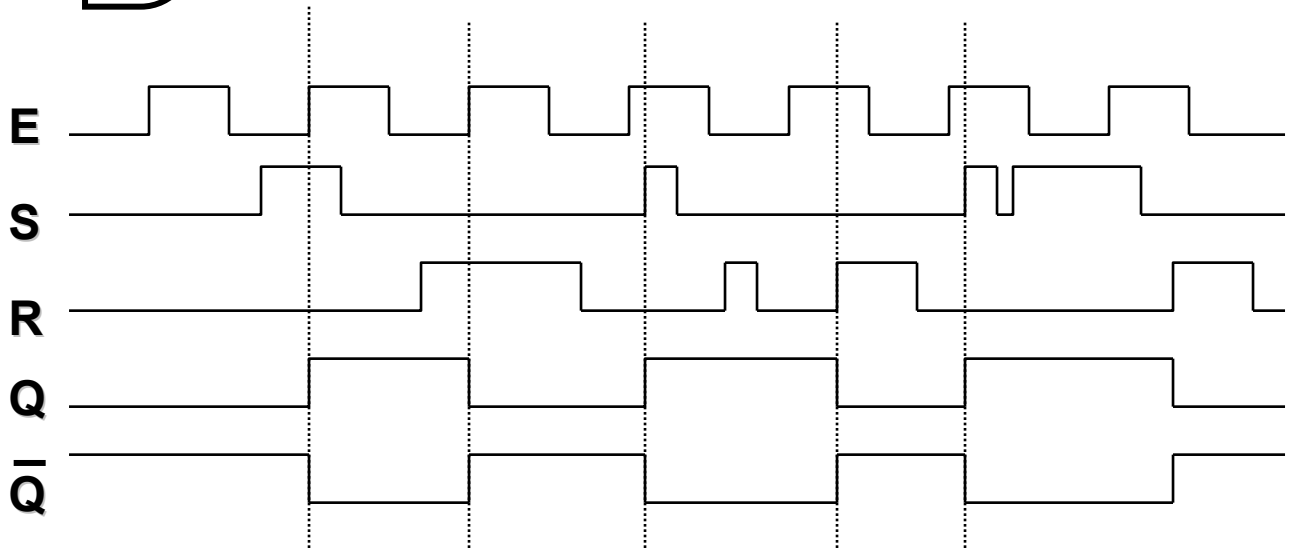


➔ **Schaltnetz hat Erinnerung des letzten Set/Reset Pulses**

S/R Latch mit Enable

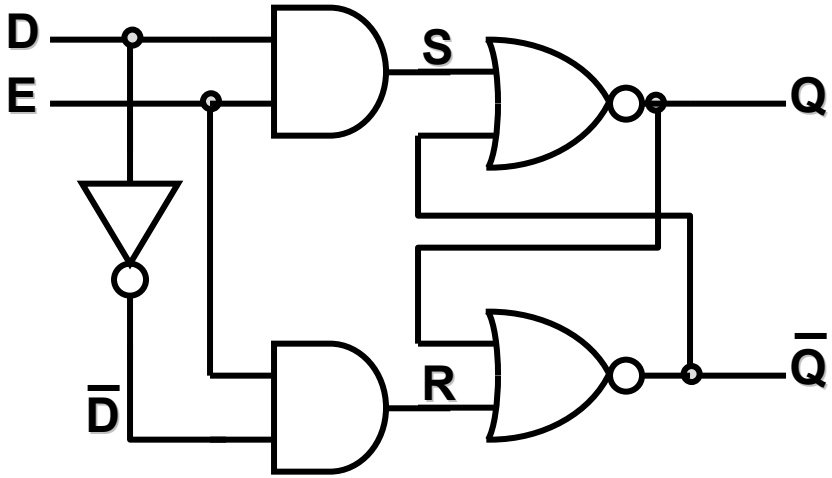


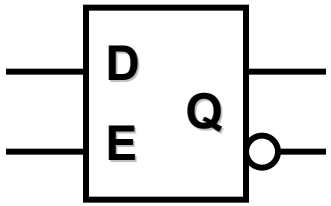




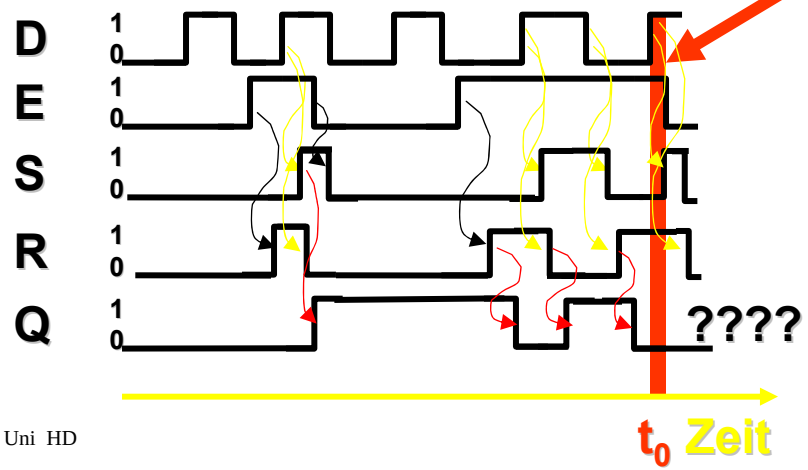
Ist dieses Latch nun frei von metastabilen Zuständen ??

Das D-Latch





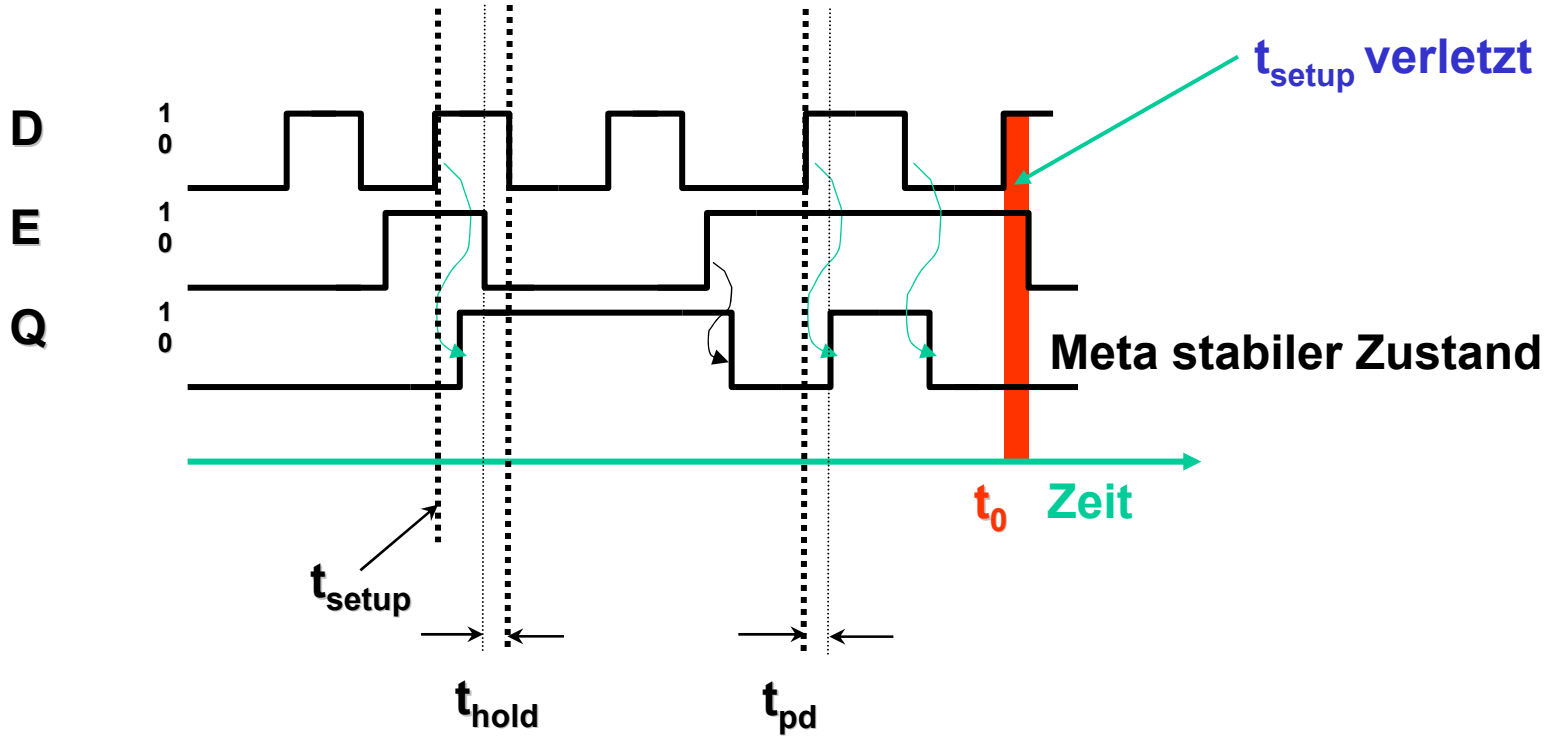
E	*D*	*Q*	*/Q*
1	0	0	1
1	1	1	0
0	X	Q	/Q



- E, D und $\overline{D}$ temporär aktiv -> S und R aktiv
- E aktiv/inaktiv Übergang -> gleichzeitiger S,R aktiv/inaktiv Übergang

-> Metastabiler Zustand wie S/R FlipFlop

Setup und Hold Times



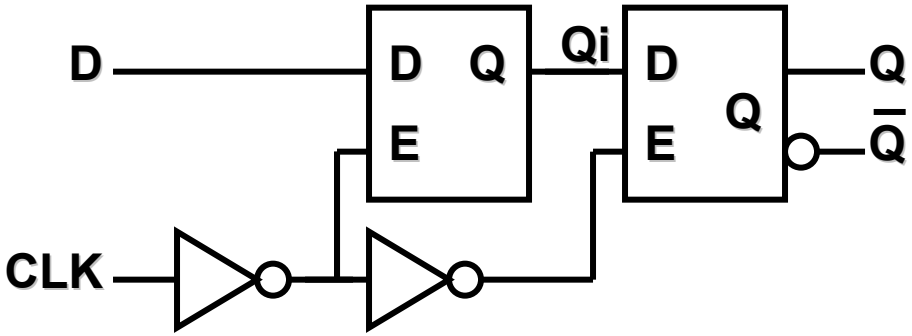
Def.:

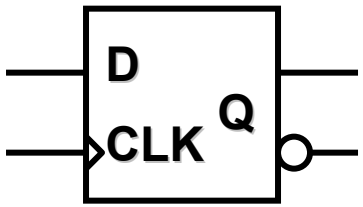
t_{setup}: Zeit, die ein Eingang stabil sein muß bevor ein anderes Signal seinen Zustand wechselt

t_{hold} : Zeitdifferenz zwischen Deaktivierung von Data und Enable.

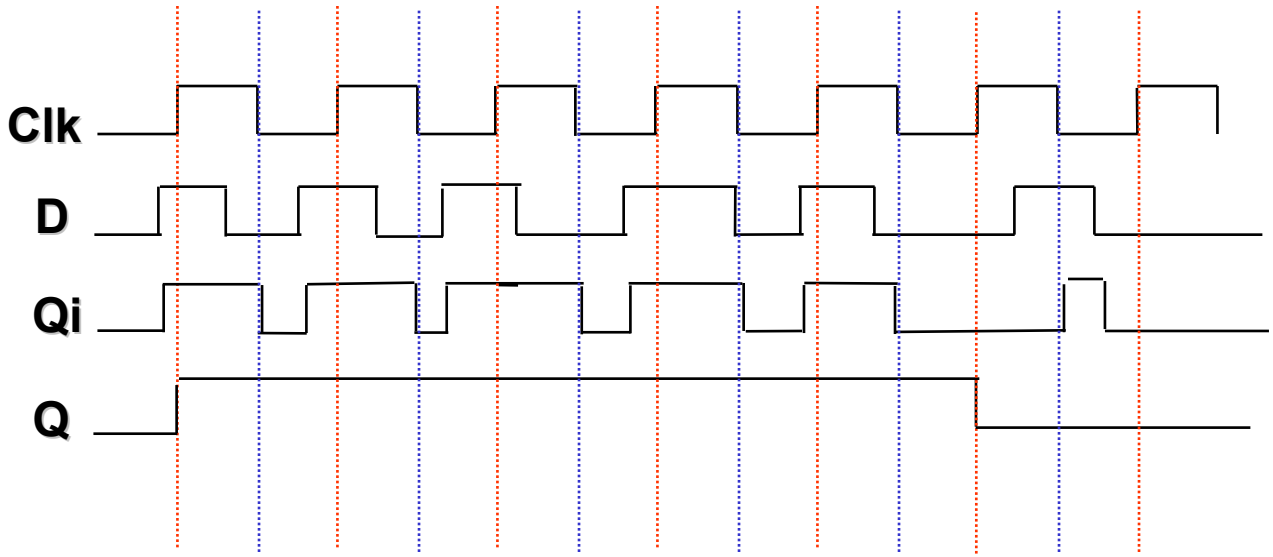
$t_{pd:}$: Zeit, die zwischen einer Änderung des Eingangssignals und einer damit verbundenen Änderung des Ausgangssignals vergeht.

Flankengesteuertes D-Flip Flop



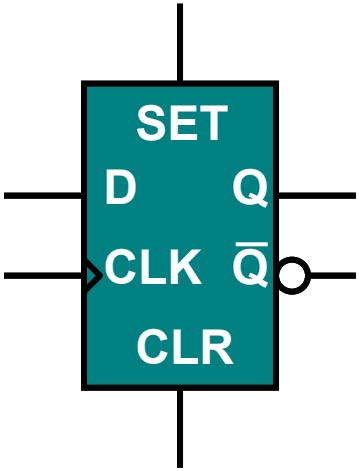


D	*CLK*	*Q*	*/Q*
0	↑	0	1
1	↑	1	0
X	0	Q	/Q
X	1	Q	/Q



Laufzeiteffekte sind in diesem Diagramm vernachlässigt

Genereller Fall

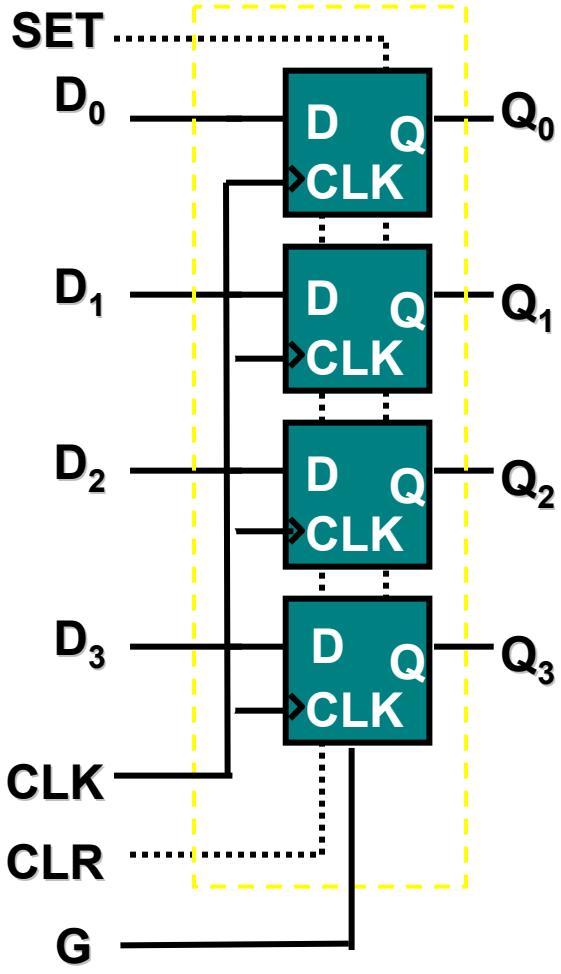


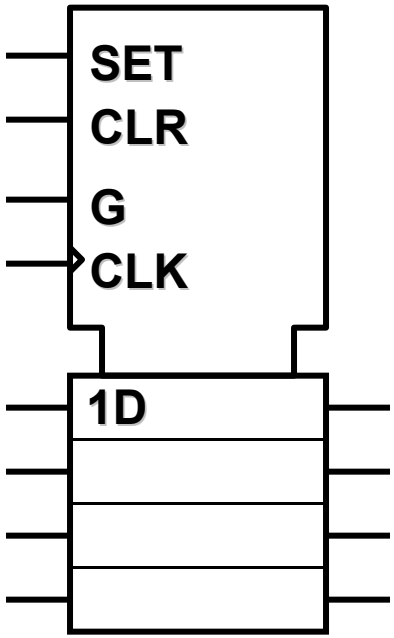
Unterscheidungsmerkmale:

• Typ	Latch / FlipFlop
• Takt	rising / falling edge (Flip Flop)
• Enable	high / low active (Latch)
• SET/CLR	high / low active
• SET/CLR	synchron / asynchron (FlipFlop)

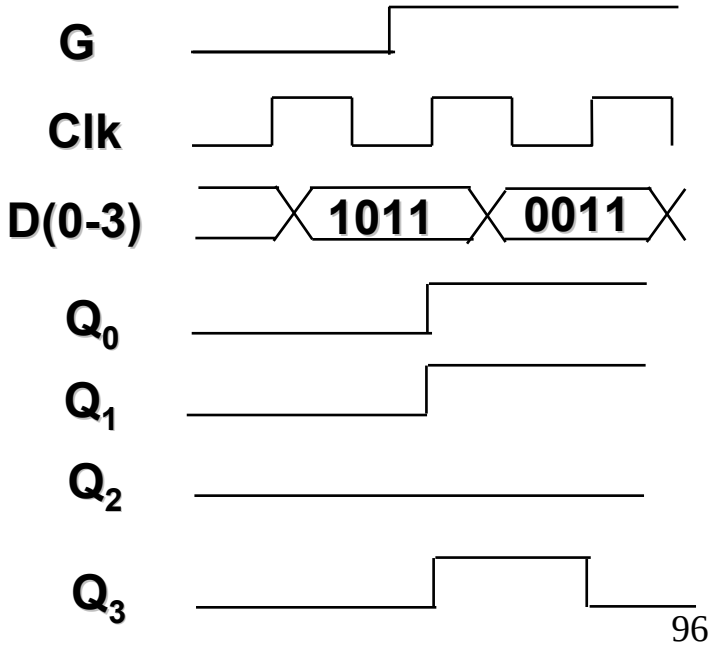
Def.: high active: Signal muß 1 sein, damit aktiviert
rising edge: Signal muß Übergang von 0 nach 1 haben
synchron: Signal wird nur bei bestimmter Takt Flanke gelesen
(Beispiel D Eingang bei FlipFlop)
asynchron: Signal wird immer gelesen

Das Register

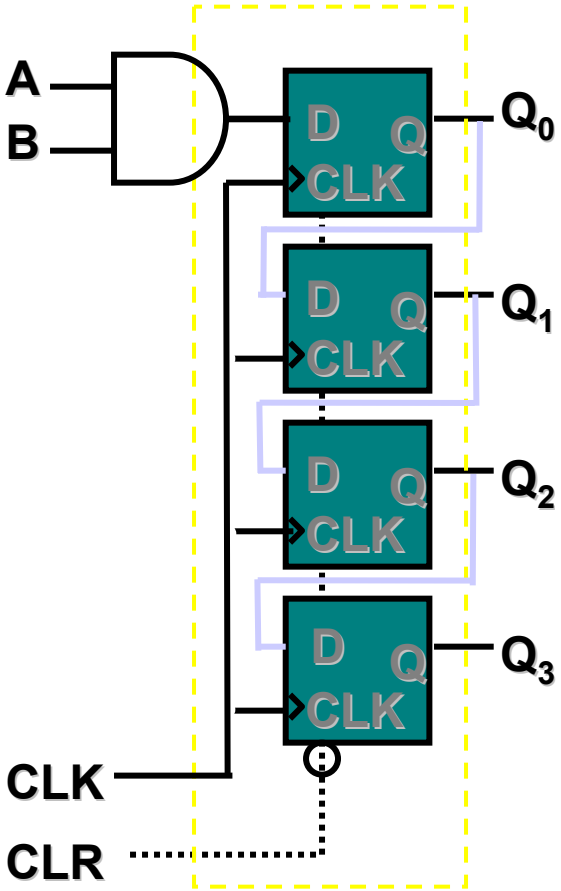


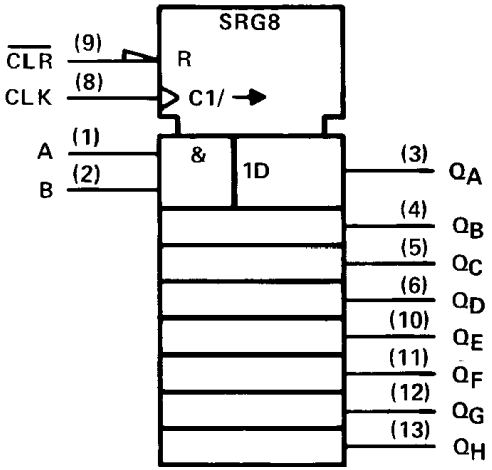


G	*CLK*	D_i	Q_i
1	↑	0	0
1	↑	1	1
1/0	0	X	Q
1/0	1	X	Q
0	↑	1	Q
0	↑	0	Q



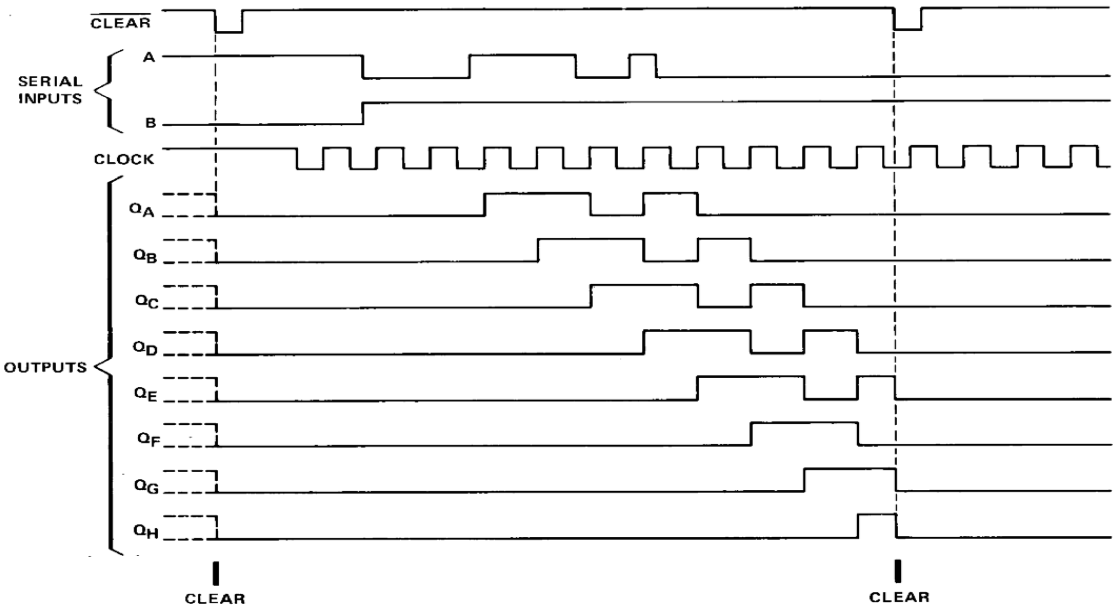
Das Shift Register



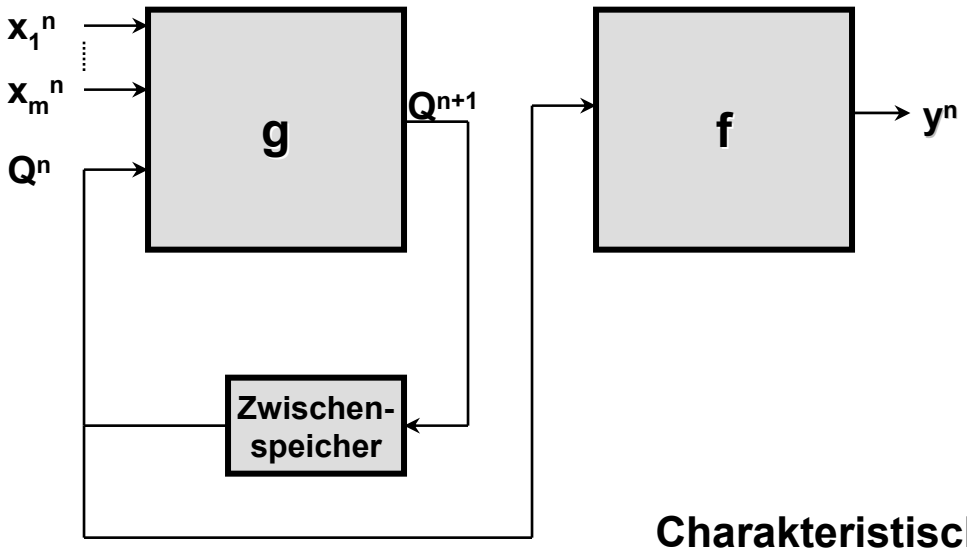


Beispiel 74164

INPUTS				OUTPUTS			
$\overline{\text{CLEAR}}$	CLOCK	A	B	Q_A	Q_B	...	Q_H
L	X	X	X	L	L		L
H	L	X	X	Q_{A0}	Q_{B0}		Q_{H0}
H	↑	H	H	H	Q_{An}		Q_{Gn}
H	↑	L	X	L	Q_{An}		Q_{Gn}
H	↑	X	L	L	Q_{An}		Q_{Gn}



1-Bit Speicherelement



Übergangsfunktion:

$$Q^{n+1} = g(x_1, \ldots, x_m, Q)^n$$

Charakteristische Funktion:

$$Q^{n+1} = (g_1 Q \vee g_2 \bar{Q})^n$$

wobei die Funktionen g1 und g2 nur von den Eingangsvariablen abhängen

Charakteristische Funktionen:

1. S/R Latch:

$$Q^{n+1} = (\overline{R}^n Q^n + S^n)$$

2. D Latch:

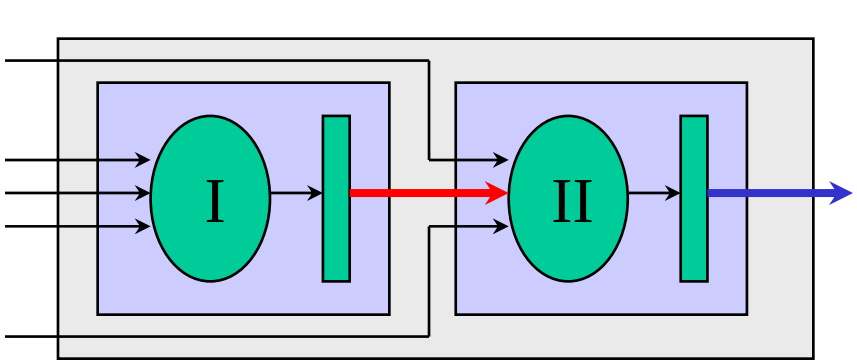
$$Q^{n+1} = D^n$$

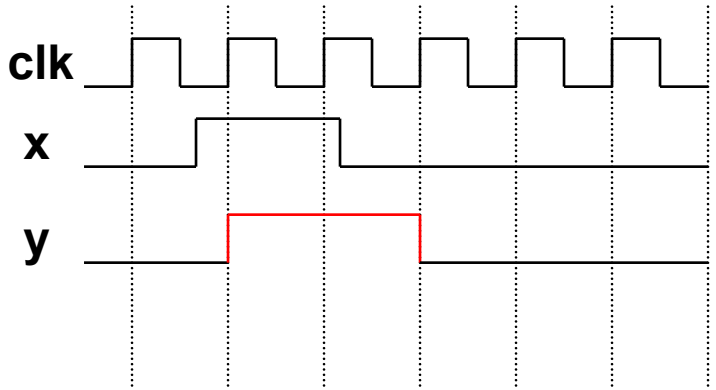
Synthesegerechte Beschreibung in VHDL

Falk Lesser, Volker Lindenstruth
Institut für Hochenergiephysik

Anforderungen an eine Schaltung

- Portierbarkeit auf eine andere Zieltechnologie
- Voraussetzung ist ein synchroner Systementwurf
  - Synchrone Schaltungen bestehen aus einer Verknüpfung von kombinatorischer Logik und Speicherelementen (Flip-Flops).
  - Alle Register werden synchron mit einem Taktsignal getriggert.
  - Alle Zustandsänderungen treten nur mit steigender Taktflanke auf, d.h. mit steigender Taktflanke übergeben die Flip-Flops den Signalpegel am Dateneingang an den Datenausgang.





Vorteile synchrone Logik

- Deterministisches Verhalten
- Höchstmaß an Portabilität
- Keine Probleme aufgrund unterschiedlicher Signallaufzeiten, da Signale zu definierten Zeitpunkten von den Speicherelementen übernommen werden.
- Robustheit gegenüber Temperatur- und Spannungsschwankungen
- Einfache Kommunikation zwischen verschiedenen Modulen der Gesamtschaltung durch ein standardisiertes synchrones Verhalten der Gesamtschaltung.
- Asynchrone Interfaces benötigen zusätzliche Signale (Handshake-Protokoll)

VHDL-Signaltypen

- Für alle Port-Deklarationen der verschiedenen Entities wird der Signaltyp **std_logic** verwendet.
- Dieser Datentyp ist im IEEE std_logic_1164 deklariert.
- Eingebunden wird das Package über die Kommandos:

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

Für den Datentyp std_logic sind neun Signalwerte definiert:

- 'U' noch nicht initialisiert
- 'X' treibend unbekannt
- '0' treibend logisch 0
- '1' treibend logisch 1
- 'Z' hochohmig
- 'W' schwach unbekannt
- 'L' schwach logisch 0
- 'H' schwach logisch 1
- '-' don't care

Wissenswertes zu Schaltnetzen

- VHDL bietet mehrere Möglichkeiten kombinatorische Logik zu beschreiben

Beispiel:

```
architecture RTL of OR_Beispiel is
begin
        X1 <= A OR B;

        Beisp_OR2: Process(A, B)
        begin
                X2 <= A OR B;
        end process;

        Beisp_OR3: Process(A, B)
        begin
                if A= '1' OR B = '1' then
                        X3 <= '1';
                else
                        X3 <= '0';
                end if;
        end process;
end RTL;
```

Für ein Signal wird nur dann kombinatorische Logik erzeugt, wenn dem Signal bei jeder Prozeßaktivierung ein Wert zugewiesen wird. Andernfalls werden vom Synthesewerkzeug Speicherelemente (Latches) eingefügt.

Verzögerungszeiten werden vom Synthesewerkzeug ignoriert und sollten deshalb **nicht** Bestandteil einer VHDL-Beschreibung sein.

Beispiel:

```
Beisp_OR4: Process (A,B)
begin
        X4 <= A OR B after 10 ns;
end;
```

Codingstyle 2

- Logikoperatoren (and, or, ..) werden direkt auf die entsprechenden Gatter abgebildet
- Aufgrund der verschiedenen Optimierungsstrategien ist es wichtig frühzeitig den Entwurf zu überprüfen.

Beispiel:

```
Summe1 <= A1 + A2 + A3 + A4          -- Kaskade aus drei Addierern
Summe2 <= (A1 + A2) + (A3 + A4)      -- Parallele Addition von (A1 + A2) und (A3 + A4)
```

- Teilausdrücke werden mehrfach genutzt, wenn die Reihenfolge **und** Position übereinstimmt.

Beispiel:

```
Summe1 <= A + B + C
Summe2 <= A + B + C      -- Summe von A + B wird nur einmal gebildet
Summe1 <= A + B + C
Summe2 <= B + A + D      -- Summe von A + B wird zweimal gebildet  (falsche Reihenfolge)
Summe1 <= A + B + C
Summe2 <= D + A + B      -- Summe von A + B wird zweimal gebildet (falsche Position)
```

Arithmetische Funktionen

- Arithmetische Operationen können in VHDL auf verschiedene Weise realisiert werden:
  - Elemente aus der Synopsys DesignWare Bibliothek (Standardbausteine wie Addierer, Speicher, ... 8051 die auf eine generischen Bauteilebibliothek aufbauen)
  - Konvertierung des *std_logic_vectors* in Signaltypen signed bzw. unsigned und anschließender Verknüpfung über Operatoren. Dadurch wird verhindert, daß grundsätzlich arithmetische Einheiten für die Verarbeitung von pos. und neg. Zahlen eingesetzt werden. (Konvertierungsfunktionen aus Package *std_logic_arith*)
  - Direktes verwenden der Verknüpfungsperatoren (+, -, *, ...). Dabei werden Standardbauelemente verwendet. Nicht benötigte Funktionalität wird vom Synthesewerkzeug eliminiert.

Codingstyle 3

- Beispiele zur Realisierung arithmetischer Verknüpfungen

Beispiel:

```
architecture RTL of OR_Beispiel is

Component dw01_add
generic (width : integer);
port (     A                    : in  std_logic_vector (width-1 downto 0);
           B                    : in  std_logic_vector (width-1 downto 0);
           CI                   : in  std_logic;
           CO                   : out std_logic;
           SUM                  : out std_logic_vector(width -1 downto 0));
end component;


begin

I1: dw01_add
    generic map  (width => 64)
    port map (A => signal_A, B => signal_B, SUM => add_result, CI => ci, CO => open);

          SUM_1 <= add_result;

          SUM_2 <=  X + Y;

end RTL;
```

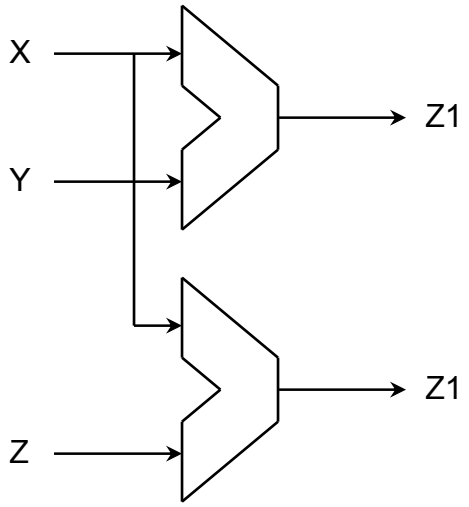
"harte" Instanz eines Addierers

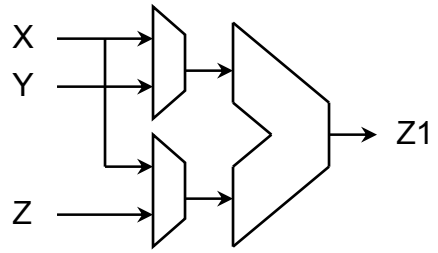
Instanz über arithmetischen Operator

Instanz versus Operator

- Bei "harten" Instanzen können arithmetische Einheiten mehrfach verwendet werden (kompaktes Design).
- Bei der Operatorinstanziierung wird für jeden Operator eine arithmetische Einheit instanziiert.
- Die VHDL-Beschreibung ist leichter lesbar und über Generics leicht konfigurierbar.

```
Beisp_add: Process (X,Y,Z)
begin
if cool = '1' then
      Z1 <= X + Y;
else
      Z1 <= Z + X;
end;
```






```
Beisp_add: Process (X,Y,Z)
component ADDER
begin
Instanz
MUX1; MUX2
end;
```

Multiplexer

- Multiplexer werden mit ***if then else***- oder ***case***-Konstrukten beschrieben.
- **if then else if** -Strukturen werden vom Synthesewerkzeug zu Prioritäts-decodern synthetisiert.
- Bei ***case***-Konstrukten werden alle Zuweisungen gleichberechtigt behandelt.

Beispiel:

```
Mux1: process (a)
begin
    if a = '00' then
        b <= '11';
    elsif a = '01' then
        b <= '01';
    else
        b <= '00'
    end if;
end;
```

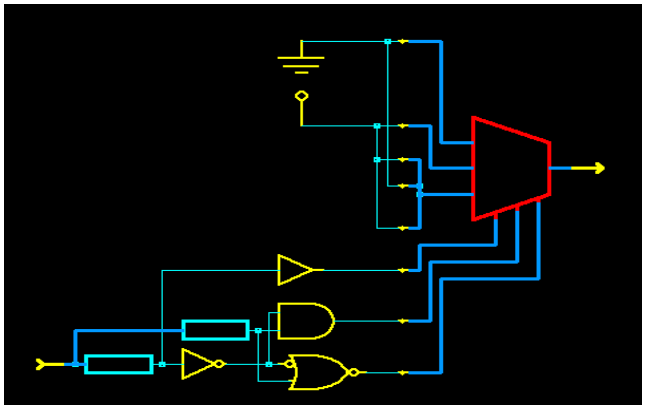
Hinweis: *if then else* und *case*-Anweisungen sind **sequentielle** Konstrukte, die in einen Prozeß einbettet werden müssen.

Bei Verwendung von case-Anweisungen und geschachtelten if-Strukturen müssen alle Signale vollständig definiert werden! Andernfalls werden vom Synthesewerkzeug Latches eingefügt, um die aktuellen Werte zu halten.

Beispiele zu Multiplexern

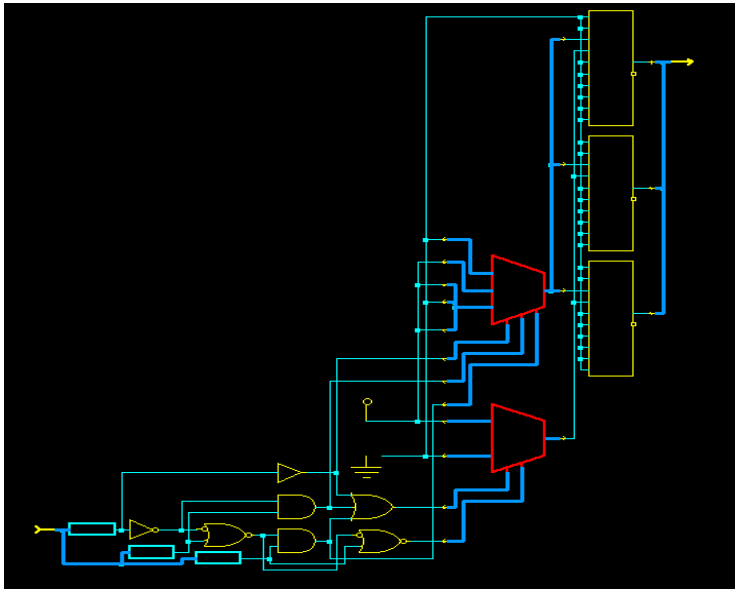
```
mux1 : process(a
begin

        if a = "00" the
                temp_b <= "000
        elsif a = "01"the
                temp_b <= "111
        else
                temp_b <= "101
        end if

end process
```

Richtig



```
mux2 : process(a)
begin

        if a = "00" ther
                temp_b <= "000"
        elsif a = "01"ther
                temp_b <= "111"
        elsif a = "10" ther
                temp_b <= "101"
        end if;

end process;
```

Falsch



Flip-Flops

- Flip-Flops sind Speicherelemente, die logische Werte (0,1) speichern können
- Der Übernahmezeitpunkt wird durch den Flip-Flop Typ bestimmt
- Das taktflankengesteuerten D-FF wird transparent, falls am 'CLK'-Port ein Wechsel von logisch '0' auf logisch '1' erfolgt. Sonst erfolgt keine Signal-zuweisung, so daß der letzte Wert gespeichert wird.

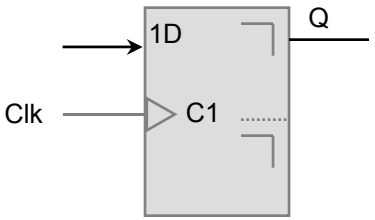
Beispiel:

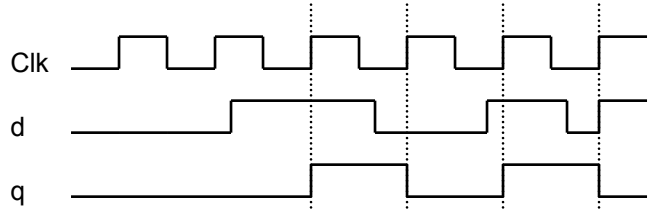
Leitet einen sequentiellen *Process* ein

```
architecture sequentiell of D-FF is
begin
    process (d)
    begin
        if reset = ‘1‘ then
                q <= ‘0‘;
        elsif clk‘event and clk = ‘1‘
then
                q <= d;
        end if;
    end process;
end sequentiell;
```

Sensitivity-List: *Process* wird immer genau dann durchlaufen, wenn eine aufgeführte Variable ihren Wert ändert.

Alle aufgeführten Anweisungen werden sequentiell durchlaufen



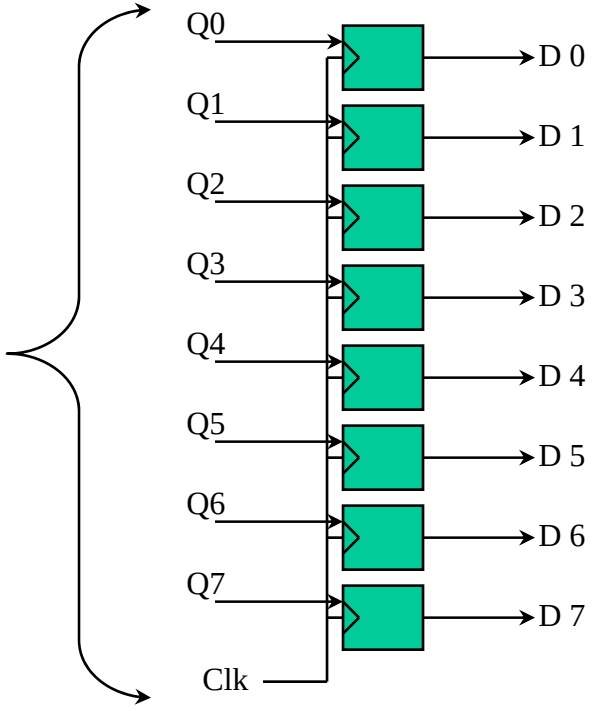


Register

- Register sind Gruppen aus Flip-Flops. Die Syntax zur Beschreibung eines Registers ist die gleiche, nur wird als Signaltyp nun anstatt std_logic der Logiktyp std_logic_vektor verwendet.
- Alternativ, kann ein Register auch aus Instanzen von D-FFs aufgebaut werden. Das Syntheseergebnis ist aber gleich.

Beispiel:

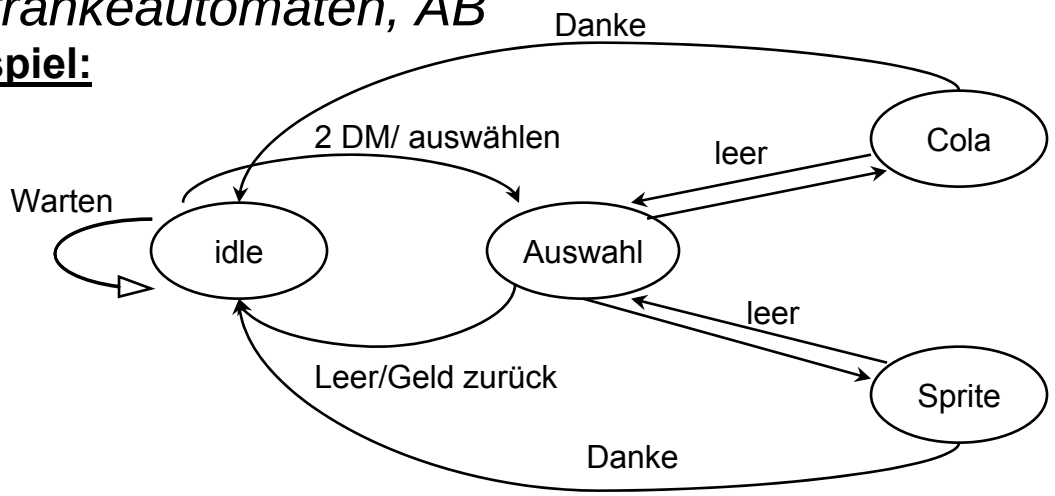
```
architecture sequentiell of Register is
signal uno, duo :        std_logic_vector (7 downto 0);
begin
    process (duo)
    begin
        if reset = '1' then
                uno <= '00000000';
        elsif clk'event and clk = '1' then
                uno <= duo;
        end if;
    end process;
end sequentiell;
```



Endliche Automat (FSM)

Endliche Automaten sind Maschinen ,die auf eine Eingabe hin bestimmte Ausgaben erzeugen. Beispiel: Bankautomaten, Getränkeautomaten, AB

Beispiel:



- Ein endlicher Automat besteht aus Speicherelementen für den aktuellen Zustand und Logikelementen zur Festlegung des Folgezustandes und der Ausgabewerte.
- Jedem Zustand der State Machine wird ein Binärcode zugeordnet
- Unterteilung der FSM in Register und kombinatorische Logik spiegelt sich exakt in der VHDL-Beschreibung wieder.

VHDL-Beschreibung einer FSM

```
architecture RTL of FSM is
type state_typ is (idle, state1, state2, state3);
signal current_state, next_state : state_typ;
attribute ENUM_ENCODING : string;
attribute ENUM_ENCODING of state_type : type is "00 01 10 11";
attribute STATE_VECTOR : string;
attribute STATE_VECTOR OF RTL is: architectur "current_state";

begin

FSM_body : process (current_state)
begin
        case current_state is
        when idle =>
        Ausgangsport1 <= '0'; Ausgangsport2 <= '0'; .....
        next_state <= state1;

        when state1 =>
        Ausgangsport1 <= '0'; Ausgangsport2 <= '1'; .....
        next_state <= state2;

        when others =>
        Ausgangsport1 <= '0'; Ausgangsport2 <= '0'; .....
        next_state <= idle;
```

Für die Zustände der FSM wird ein neuer Datentyp deklariert.

Mit dem *STATE_VECTOR* Attribut ist der DC in der Lage, den Zustandsvektor zu identifizieren. *ENUM_ENCODING* erlaubt die feste Zuordnung binärer Zustandsvektoren zu den angegebenen Zuständen.

In diesem Prozeß werden in Abhängigkeit vom aktuellen Zustand der Folgezustand und die Ausgabewerte berechnet.

VHDL-Beschreibung einer FSM

```
Reg: process (clk, reset)
begin;
   if reset = '1' then
         current_state <= idle;
   if clk'event and clk = '1' then
         current_state <= next_state;
   end if;
end process;

end RTL;

configuration FSM_CFG of FSM is
for RTL
end for;
```

Prozeß zur Speicherung des aktuellen Zustandes.

Register verfügt - wie alle Register - über einen asynchronen Reset-Eingang.

Zusätzlich können/werden hier die Ausgangsregister beschrieben

Werden in der Beschreibung der Architektur Submodule verwendet, müssen diese in Konfiguration den Instanzen die VHDL-Beschreibungen zugewiesen werden.

Synthese digitaler Schaltungen

- Die Synthese einer digitalen Schaltung entspricht der Abbildung auf eine Zieltechnologie.
- Verhaltensbeschreibung (VHDL) wird nahezu direkt in Struktur aus kombinatorischen Logikelementen und Registern umgesetzt.
- Festlegung welche Signale gespeichert werden müssen
- Auswahl der Registertypen (in Abhängigkeit von der Triggerbedingung)
- Das Verhalten der nicht zu speichernden Signale wird als kombinatorische Logik implementiert.
- Ergebnisse der RTL-Synthese werden in einer generischen Netzliste abgelegt, die auf eine technologieunabhängigen Bibliothek aufbaut.
- Anschließend wird die Netzliste auf eine technologiespezifische Bibliothek abgebildet (FPGA, AMS035, ...)

Optimierungsziele:

- Einhaltung der festgelegten Design-Regeln (Design Rule Constraints)
  - Diese werden von dem Synthesewerkzeug automatisch beachtet
- Schnelle, kompakte und verlustarme schaltungstechnische Realisierung
  - Direkte Einflußnahme des Designers über Synthese-Constraints

Synthese mit dem FPGA-Analyzer

- Konnte die einwandfreie Funktionalität mit dem Simulator überprüft werden, kann mit der Synthese (Technologie-Mapping) begonnen werden. Das verwendete Synthesewerkzeug wird mit Befehl: ***fpga_analyzer*** aufgerufen.

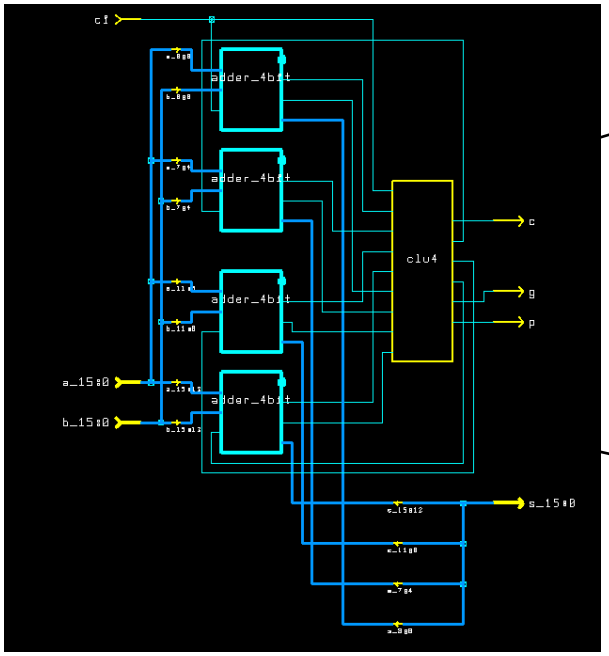
Charakteristika:

- Graphische Benutzeroberfläche zur Anzeige der synthetisierten Netzliste
- Benutzte Kommandos können ebenso im ***Comand Window*** in textueller Form abgesetzt werden (*dc_shell*).
- Nicht alle Befehle sind über Pulldown-Menüs erreichbar.
- Aufruf der Online-Dokumentation über ***sold***.

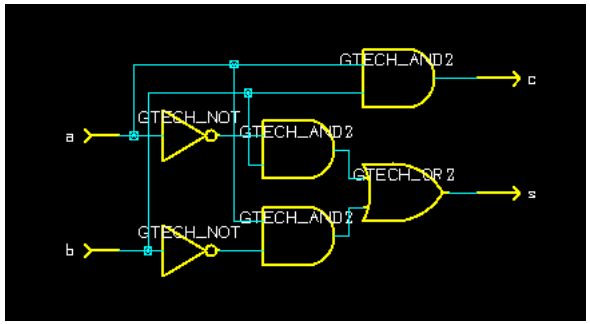
Synthese digitaler Schaltungen

Beispiel: **16 Bit Addierer**

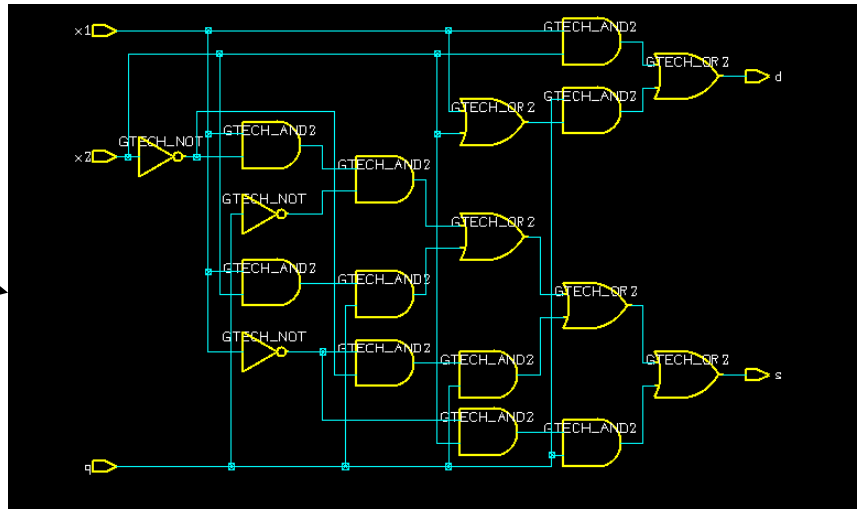
Mit Hilfe des Syntheseprogramms wird die VHDL-Beschreibung in eine Netzliste aus Standardbauelementen überführt.



Darstellung des 16-Bit Addierers in hierarchischer Form



Schaltnetz des Halbaddierers

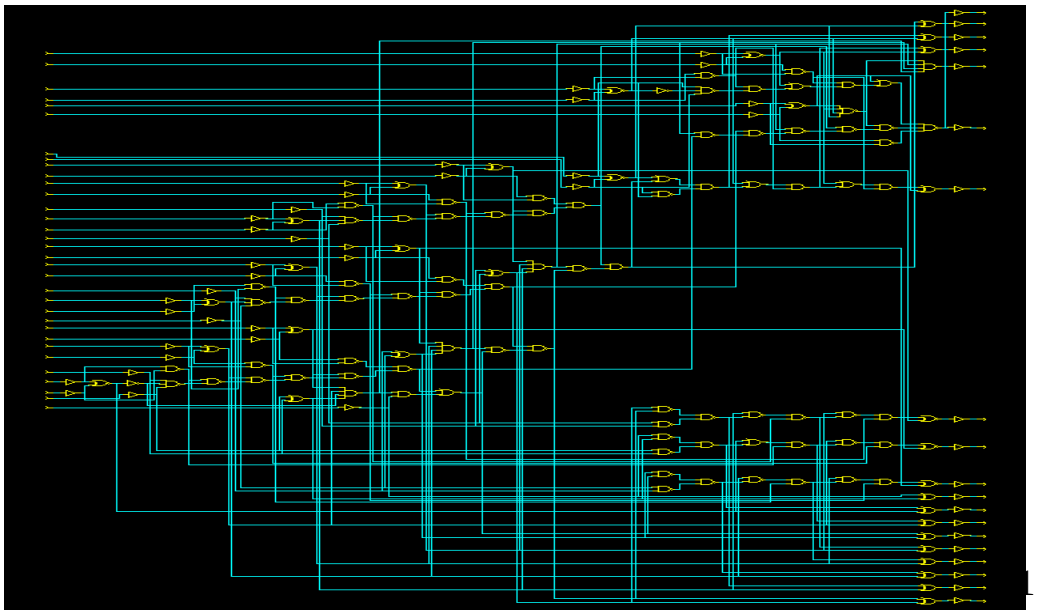


Schaltnetz des Volladdierers

Technologiemapping

- In einem zweiten Schritt wird die technologieunabhängige Netzliste auf eine Bauteilebibliothek eines Halbleiterherstellers abgebildet
- Hierbei optimiert das Synthesetool die bisherige Darstellungs-form bzgl. Geschwindigkeit, Fläche und weiteren Design-Constrains.

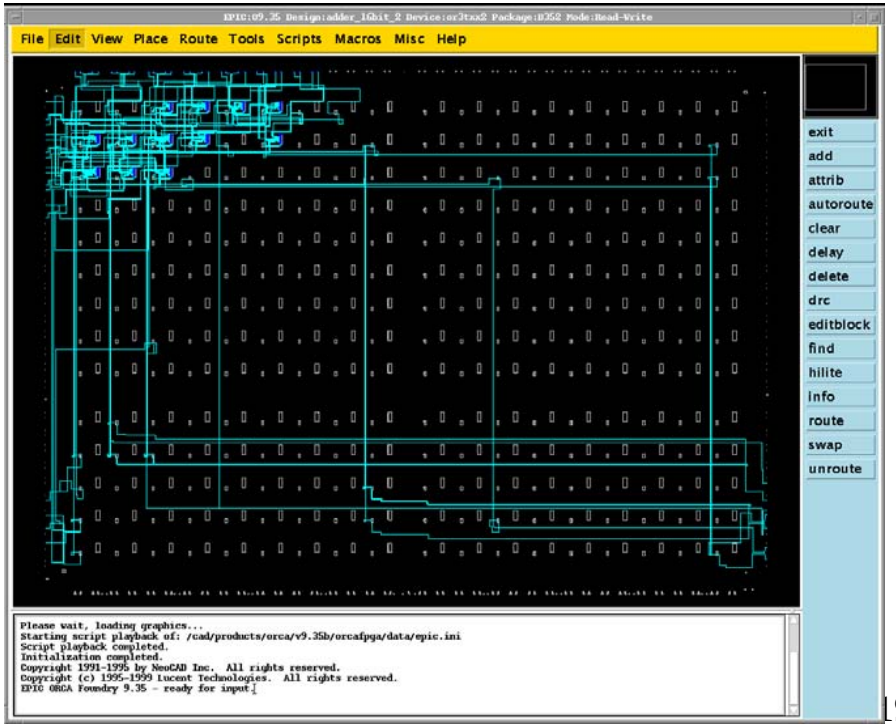
Die synthetisierte Schaltung. Aufgebaut ist Sie aus den Bausteinen der Zieltechnologie



Place and Route bei FPGAs

- Die vom Synthesetool erzeugte Netzliste kann nun dazu verwendet werden, die Schaltung zu Plazieren und Verdrahten
- Nach dem Einfügen von I/O-Buffern erfolgt das Plazieren und Verdrahten der Logikbausteine (PFUs und LUTs).
- Es kann ein automatisierter Ablauf gewählt werden aber auch eine Verdrahtung von Hand ist möglich.

- Nach erfolgtem Plazieren und Verdrahten wird das Layout erstellt
- Die komplette Auswahl und Verdrahtung wird in ein Konfigurations-Bit-File gespeichert.
- Nach jedem Power-Up wird das FPGA neu durch das Bit-File konfiguriert.

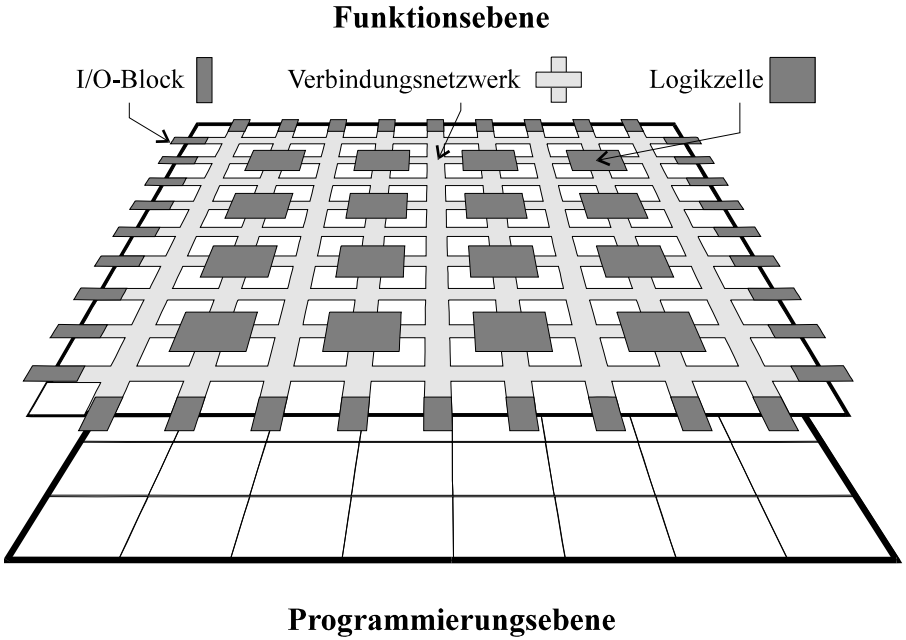


Was ist ein FPGA

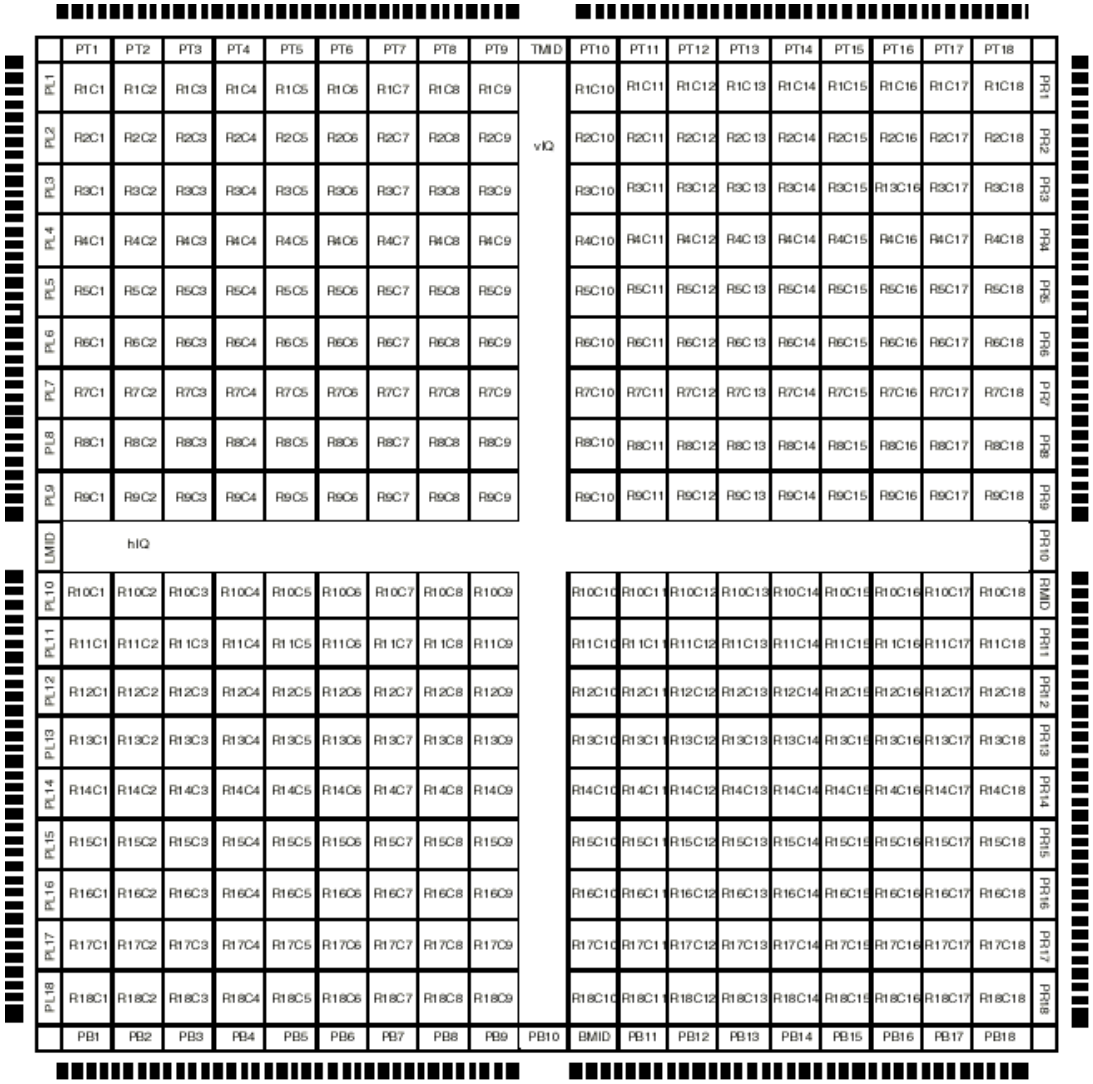
- FPGA = Field Programmable Gate Array
- Wiederverwendbare programmierbare Logikbausteine
- Fuse-, Antifuse- RAM-Technologie
- Vergleichbar mit dem Laden und Ausführen eines neuen Programms durch einen Mikroprozessor.

Bestandteile:

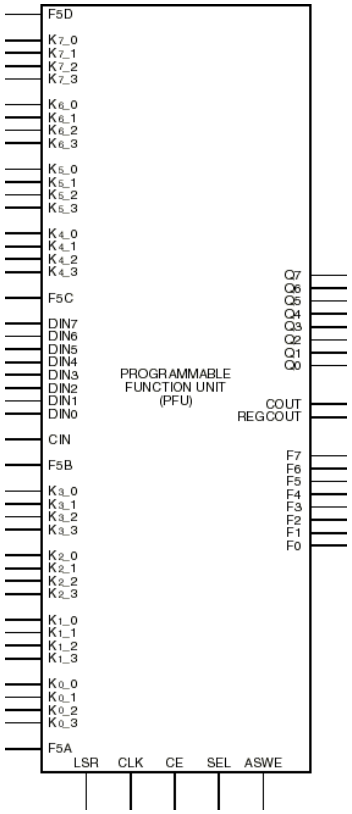
- Programmierbare Logikzellen
- Programmierbare I/O-Zellen
- Verbindungsnetzwerk



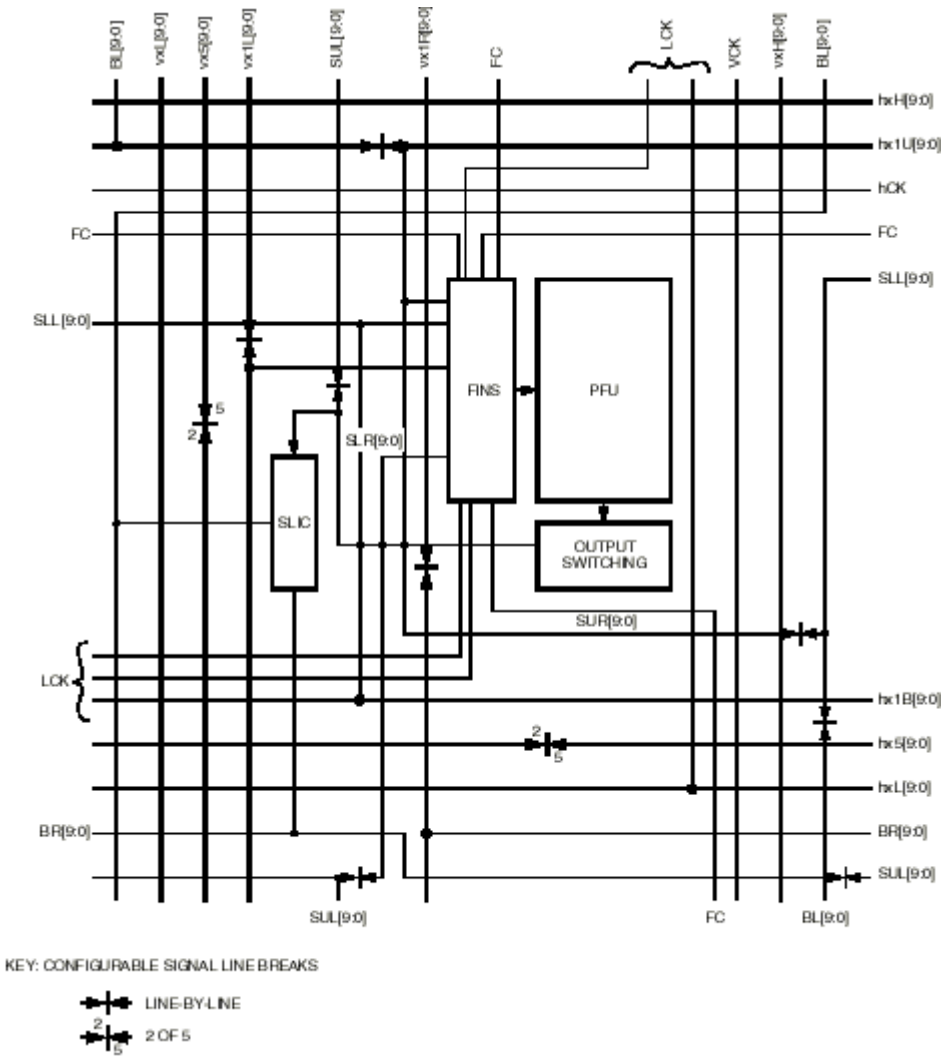
Programmierbare Schaltwerke



Das FPGA besteht aus einer Matrix von Schaltwerken (PFU), die frei programmiert werden können. Sie sind durch eine Matrix von frei programmierbaren Signalleitungen verbunden.



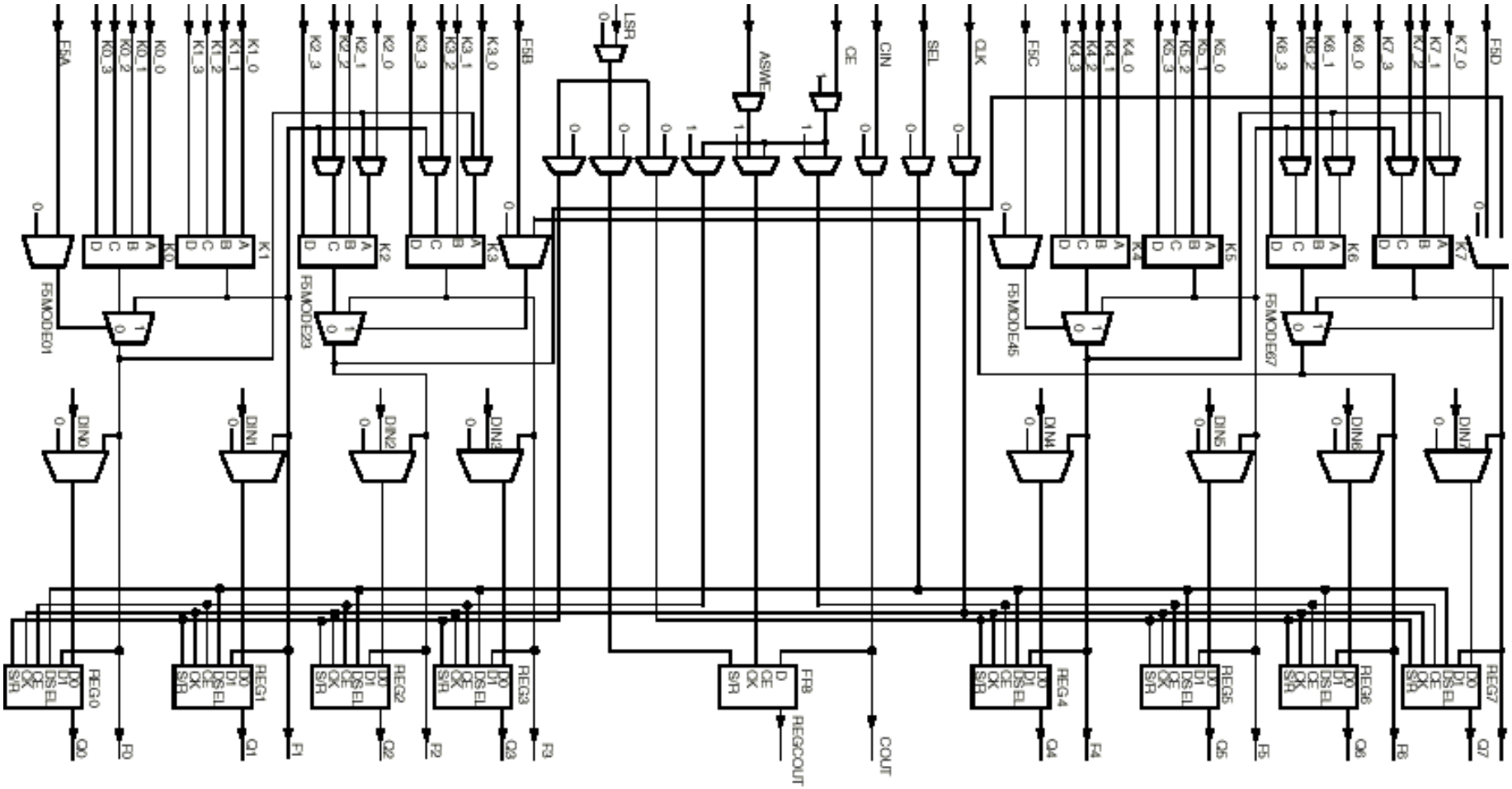
Detail eines internen FPGA Blocks



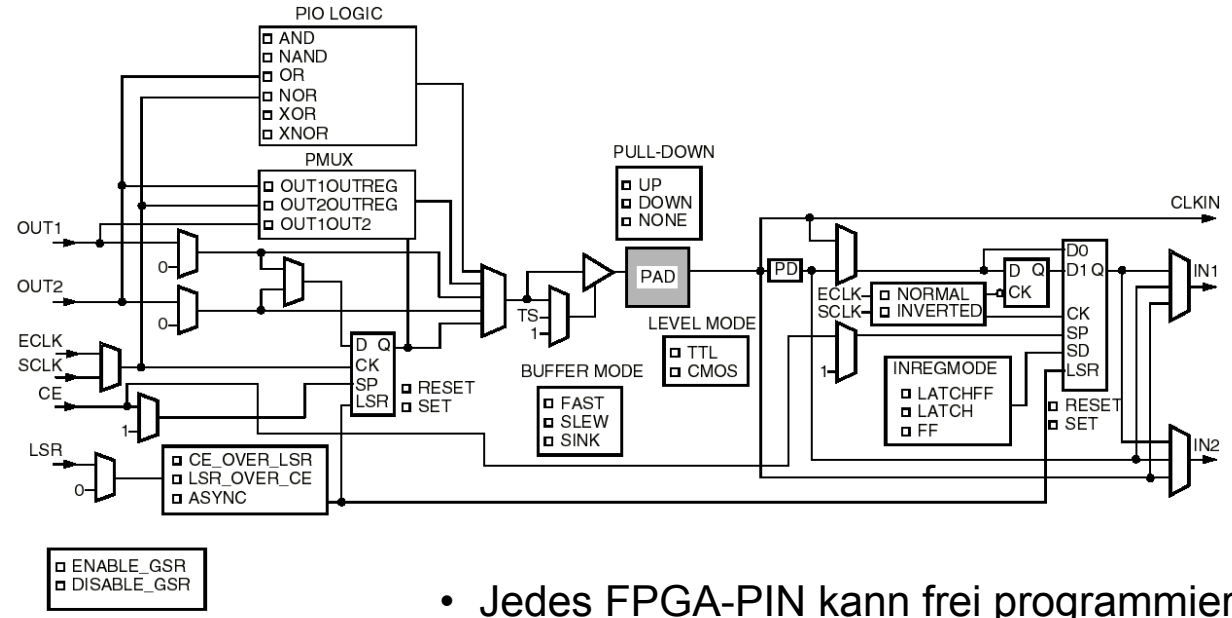
Bestandteile:

- Orca3 = SRAM-FPGA
- Programmable Function Units
- Maskenprogrammierbare Logik (Schnittstelle zum PCI-Bus)
- Feldprogrammierbare Logik
- Supplemental Logic and Interconnect Cell
- Jede PFU beinhaltet acht Flip-Flops und und acht LUTs

Das innere einer PFU



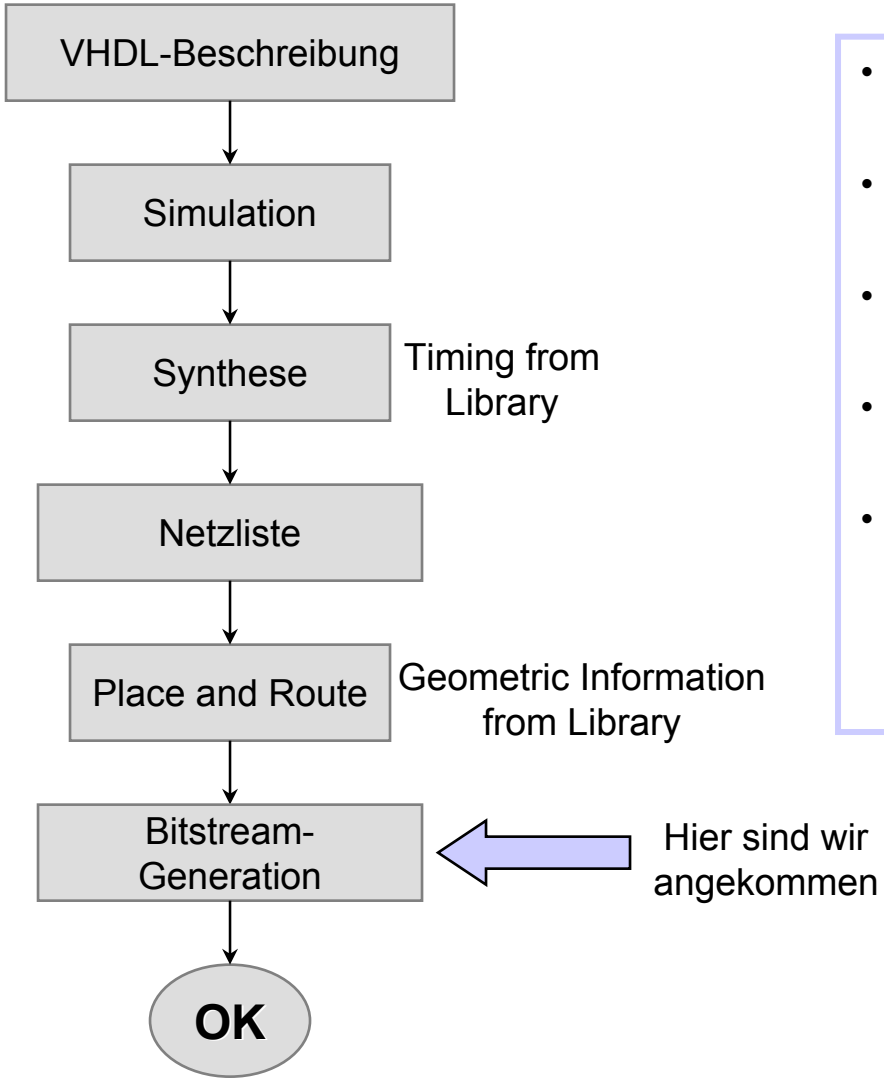
FPGA Ein/Ausgabe



Key: PD = programmable delay.

- Jedes FPGA-PIN kann frei programmiert werden
- Es kann Input, Output oder Bidirektionales Pin verwendet werden
- Zusätzlich sind verschiedene Technologievarianten möglich
  - TTL, CMOS
  - mit/ohne PULL Up bzw. PULLDOWN Widerstand
  - Treiberleistung (6 mA, 12 mA)

Zusammenfassung



- VHDL ist die Beschreibungssprache zum Entwurf der Hardware
- Die Simulation dient der Verifikation der Funktionalität
- Die Synthese entspricht der Abbildung auf eine Zieltechnologie
- Die Netzliste ist eine Verschaltung von Bausteinen der Zieltechnologie
- Beim Place und Route werden die benötigten Bausteine plaziert und entsprechend der Netzliste verdrahtet. Dabei versucht das Synthesewerkzeug eine möglichst kompakte und schnelle Schaltung zu realisieren.

Inhalt des vierten Tages

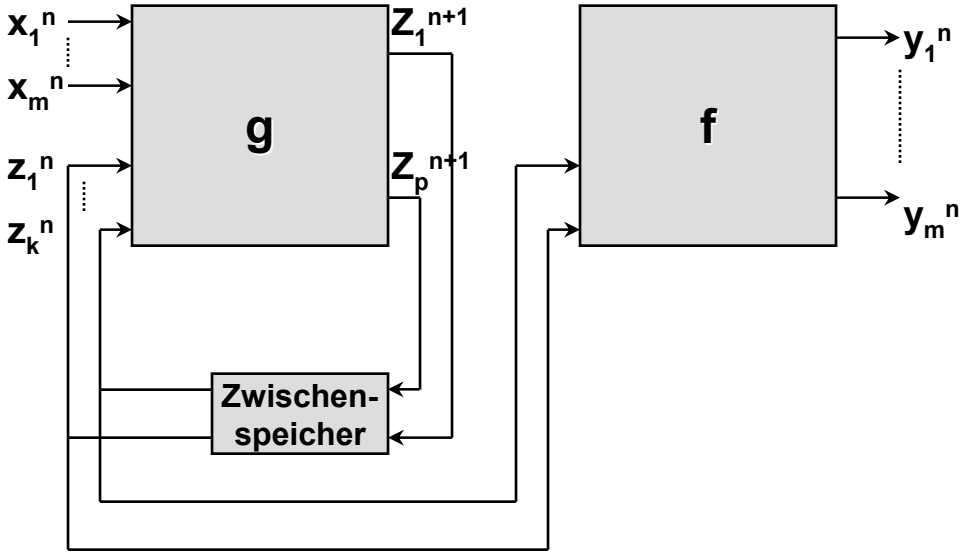
- Endliche Automaten
- VHDL-Modellierung einer FSM (Finite State Machine)
- Erweiterte Synthesestrategien
- Synthesewerkzeuge des Orca Designflows
- Übungen

Grundlagen der Automatentheorie

1. Moore:

Ausgabefunktion: $\underline{Y}^n = f(\underline{Z})^n$

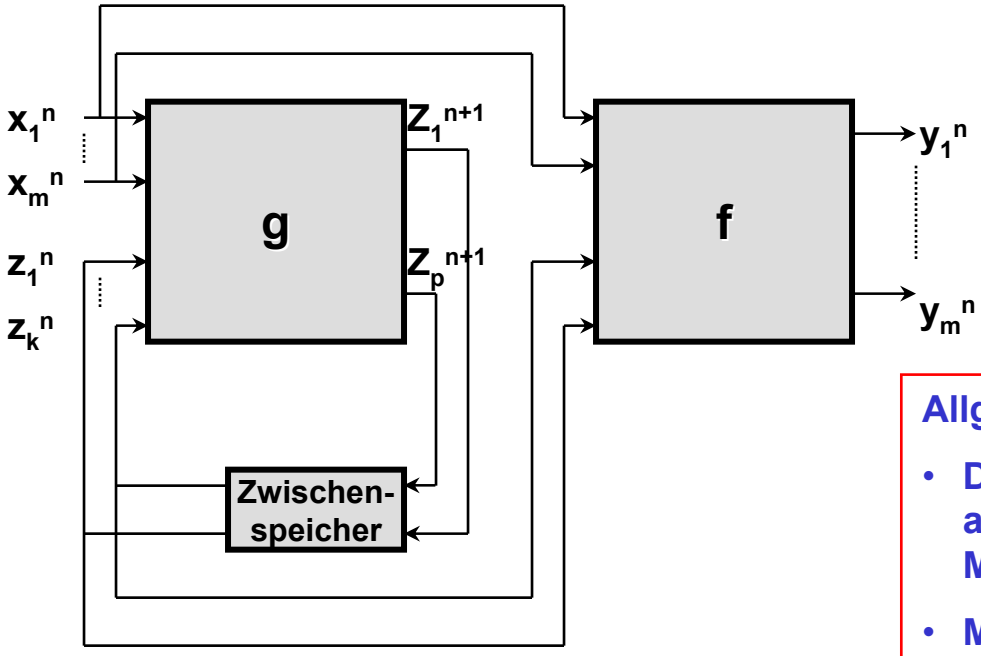
Übergangsfunktion: $\underline{Z}^n = g(\underline{X}, \underline{Z})^n$



Grundlagen der Automatentheorie

Automaten-theoretische Beschreibung:

2. Mealy:



Eingabevektor: $\underline{X} = (x_1, x_2, \ldots, x_n)$

Ausgabevektor: $\underline{Y} = (y_1, y_2, \ldots, y_m)$

Zustandsvektor: $\underline{Z} = (z_1, z_2, \ldots, z_p)$

Ausgabefunktion: $\underline{Y}^n = f(\underline{X}, \underline{Z})^n$

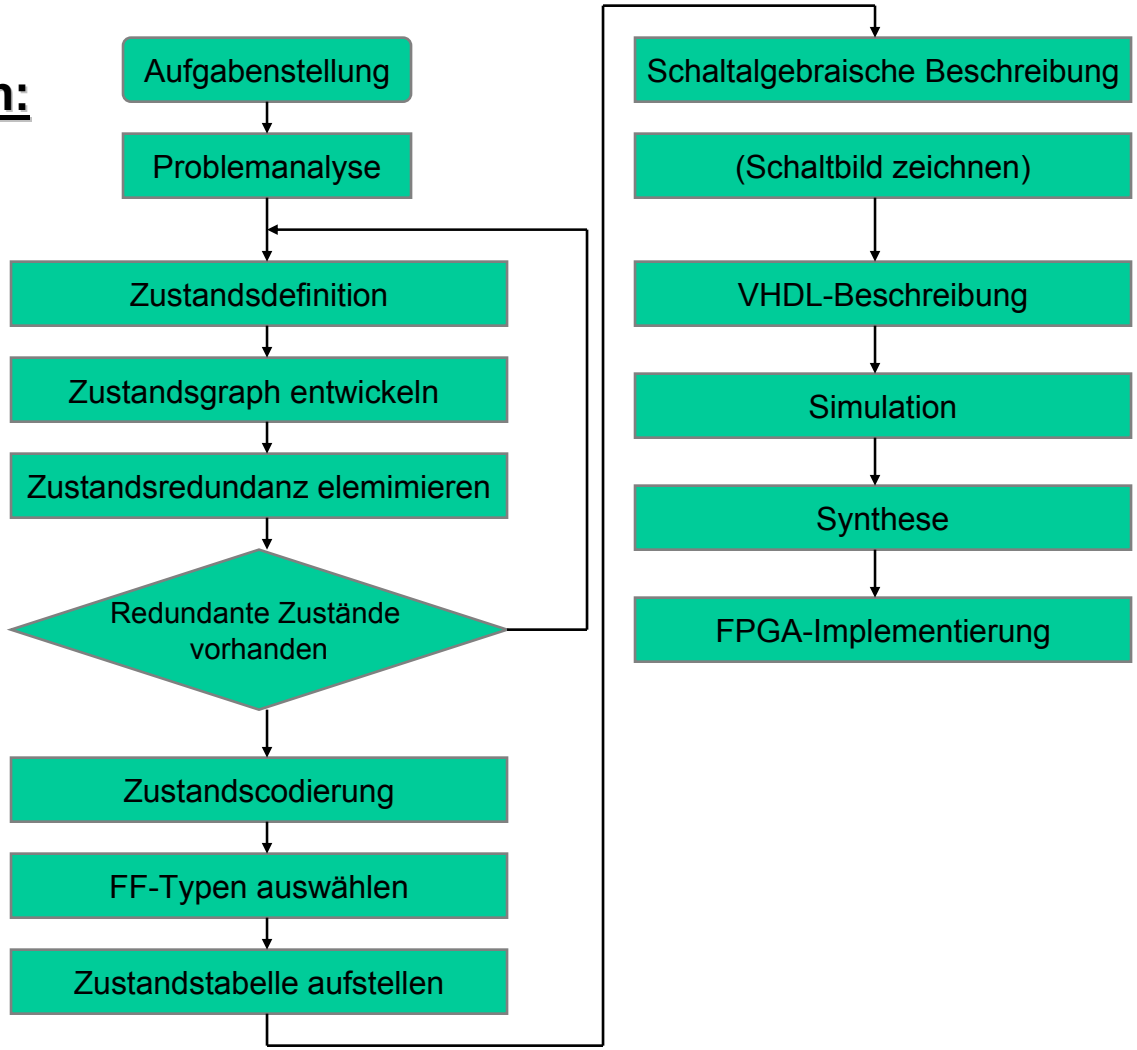
Übergangsfunktion: $\underline{Z}^n = g(\underline{X}, \underline{Z})^n$

Allgemein gilt:

- Die Ausgabe eines Mealy-Automat ist abhängig vom Zustand und Eingabe, der Moore-Automat jedoch nur vom Zustand.
- Moore- und Mealy Automat sind ineinander überführbar.

Entwurf von Schaltwerken

Verfahren:



Darstellungsformen

1. Zustandstabellen:

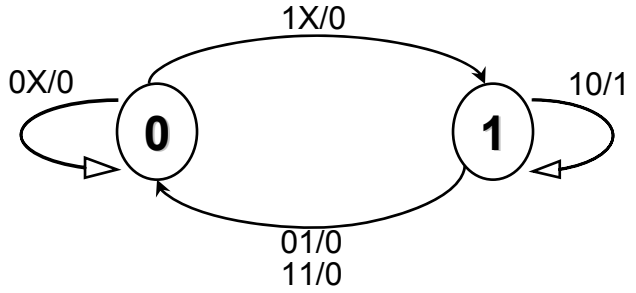
Z^n	X_1^n	X_2^n	Z^{n+1}	Y^n
0	0	0	0	0
0	0	1	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	0
1	0	1	0	0
1	1	0	1	1
1	1	1	0	1

Vollständige Zustandstabelle mit allen Eingangsvariablen, Zuständen, Folgezuständen und Ausgangsvariablen

X_1^n	X_2^n	Z^{n+1}	Y^n
0	0	Z^n	0
0	1	0	0
1	0	1	Z^n
1	1	$\leftarrow^n$	Z^n

Vereinfachte Zustandstabelle mit allen Eingangsvariablen, Folgezuständen und Ausgangsvariablen. Zustände auf Vorzustand bezogen

2. Zustandsübergangsgraph:



X = don`t care

- Knoten: Zustände des Schaltwerks
- Kanten: Zustandsübergänge
- Kantenmarkierung: Eingabe/Ausgabe (Mealy-Automat)

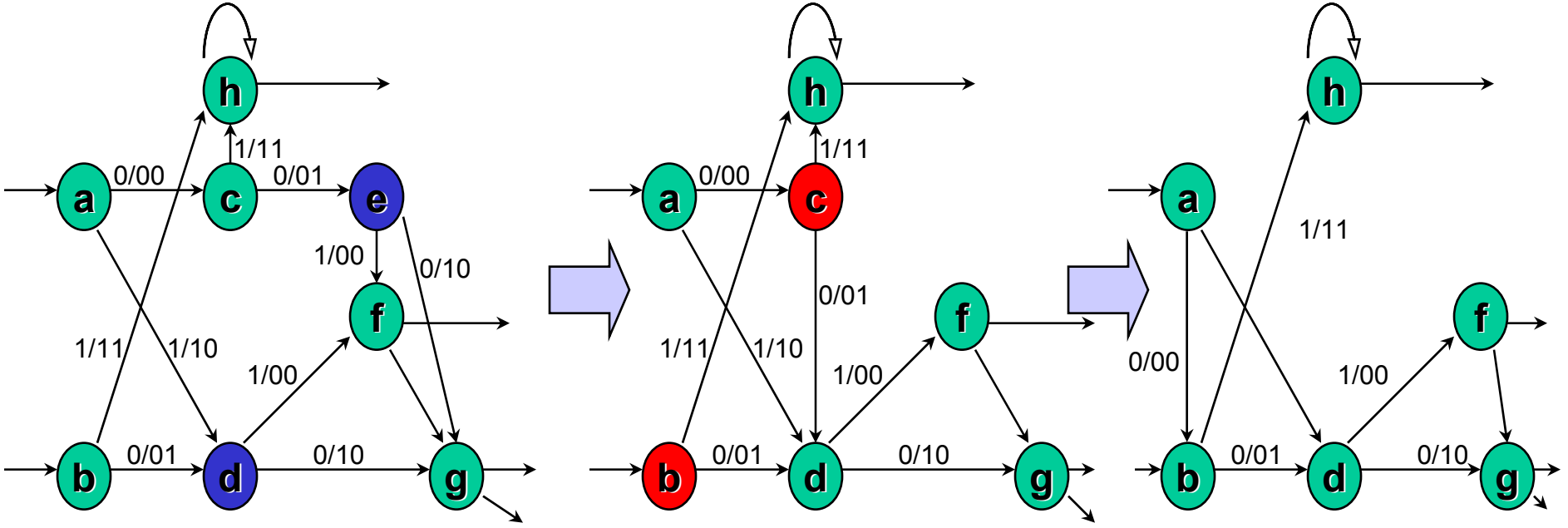
Es wird immer genau ***ein*** Zustand angenommen

Moore-Automat analog mit Ausgabe als Beschriftung der Zustände

Einsparung redundanter Zustände

Zwei Zustände eines Schaltwerks heißen äquivalent, wenn sie bei gleichem Eingangsvektor stets den selben Ausgangsvektor erzeugen und einen äquivalenten Folgezustand annehmen.

Äquivalente Zustände können durch einen einzigen Zustand ersetzt werden.



Zustand „e“ und „d“ sind äquivalent

Zustand „e“ eliminiert

Zustand „b“ und „c“ sind äquivalent

Zustand „c“ eliminiert

Schaltwerksentwurf

1. Aufgabenstellung:

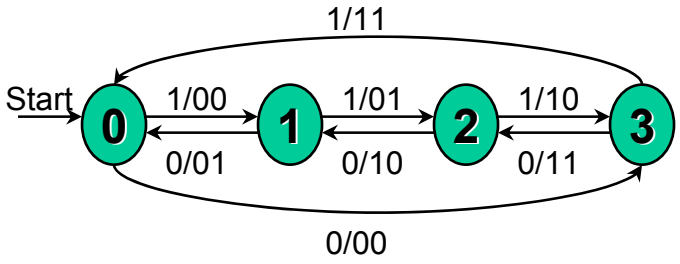
Modulo-4 Vorwärts/Rückwärtszähler

Der Zähler soll von 0 bis 3 zählen können. Dabei soll durch einen Steuereingang x die Zahlenfolge 0,1,2,3 (vorwärtszählen) für x=1 und beim rückwärtszählen die Zahlenfolge 3,2,1,0 für x=0 durchlaufen´und ausgegeben werden. Am Ausgang ist der Zählerstand anzugeben (Ausgabevektor y_0, y_1). Der Zähler ist als Ringzähler zu realisieren.

2. Zustandsdefinition:

Vier Zustände erforderlich: 0, 1, 2, 3

3. Entwicklung des Zustandsgraphen:

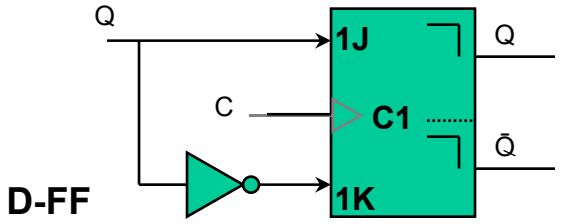


4. Zustandsredundanzen eliminieren:

5. Zustandscodierung:

00, 01, 10, 11

6. Wahl des FF-Typs:



D^n	Q^{n+1}	$\overline{Q}^{n+1}$
0	0	1
1	1	0

7. Zustandstabelle:

	x	Z_1^n	Z_0^n	Z_1^{n+1}	Z_0^{n+1}	Y_1	Y_0
Vorwärts-zählen	1	0	0	0	1	0	0
	1	0	1	1	0	0	1
	1	1	0	1	1	1	0
	1	1	1	0	0	1	1
Rückwärts-zählen	0	1	1	1	0	1	1
	0	1	0	0	1	1	0
	0	0	1	0	0	0	1
	0	0	0	1	1	0	0

8. Schaltalgebraische Beschreibung der Ausgangs- und Übergangsfunktion:

Ausgangsfunktion:

$$Y_0 = Z_0^n$$

$$Y_1 = Z_1^n$$

Übergangs-funktion:

$$Z_0^{n+1} = (x \leftarrow_0 \leftarrow_1 + x \leftarrow_0 Z_1 + \boxtimes \leftarrow_0 Z_1 + \boxtimes \leftarrow_0 \leftarrow_1)^n$$

$$Z_1^{n+1} = (x \leftarrow_0 \leftarrow_1 + \boxtimes Z_0 Z_1 + x Z_0 \leftarrow_1 + x \leftarrow_0 Z_1)^n$$

Minimierung:

$$Z_0^{n+1} = \leftarrow_0^n$$

Z_1^{n+1} **nicht weiter minimierbar**

Koeffizientenvergleich für D-FF:

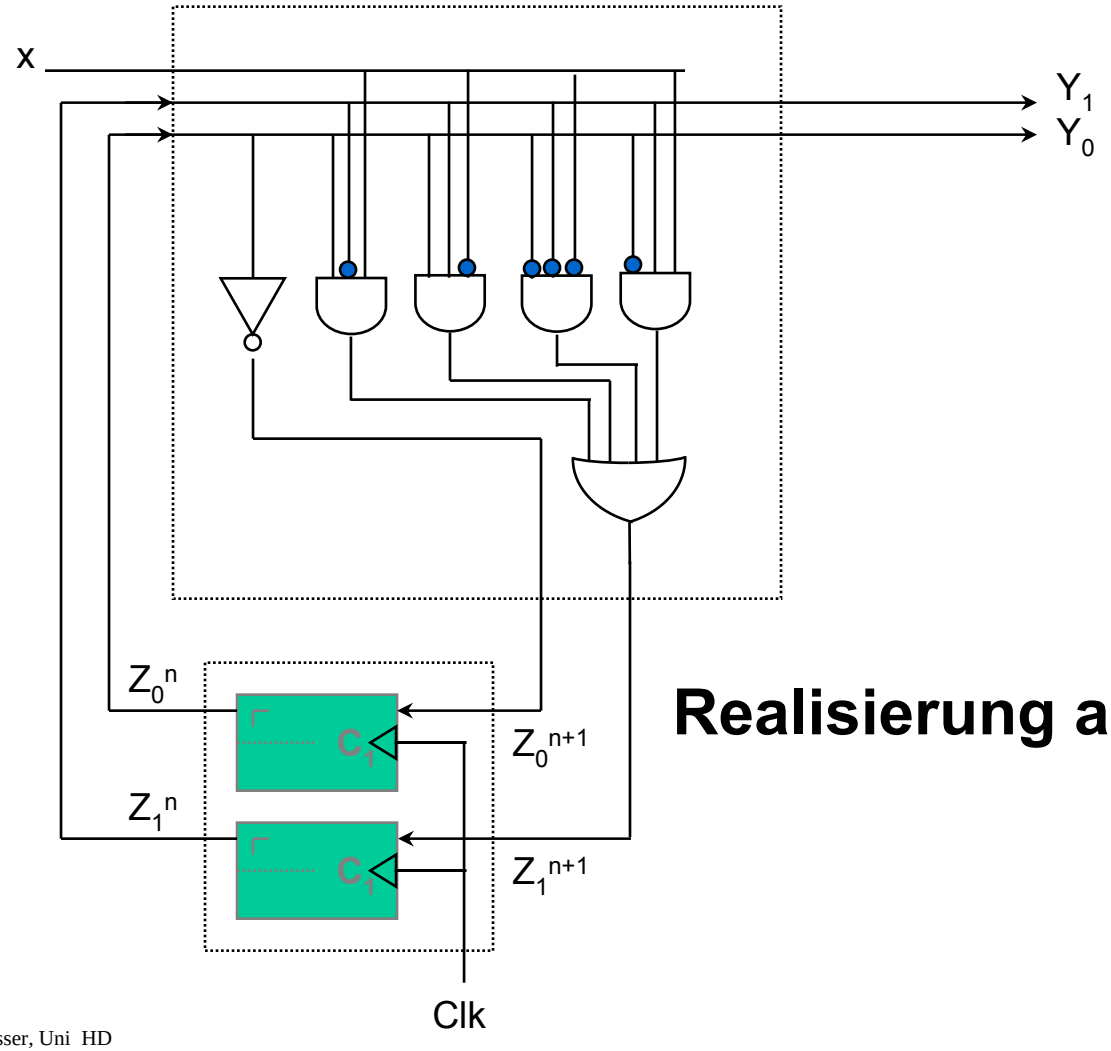
$$Q^{n+1} = D^n$$

Ansteuerungsgleichung:

$$Q_0^{n+1} = D_0^n = Z_0^{n+1} = \leftarrow_0^n$$

$$Q_1^{n+1} = D_1^n = Z_1^{n+1} = (x \leftarrow_0 \leftarrow_1 + \boxtimes Z_0 Z_1 + x Z_0 \leftarrow_1 + x \leftarrow_0 Z_1)^n$$

Schaltbild:

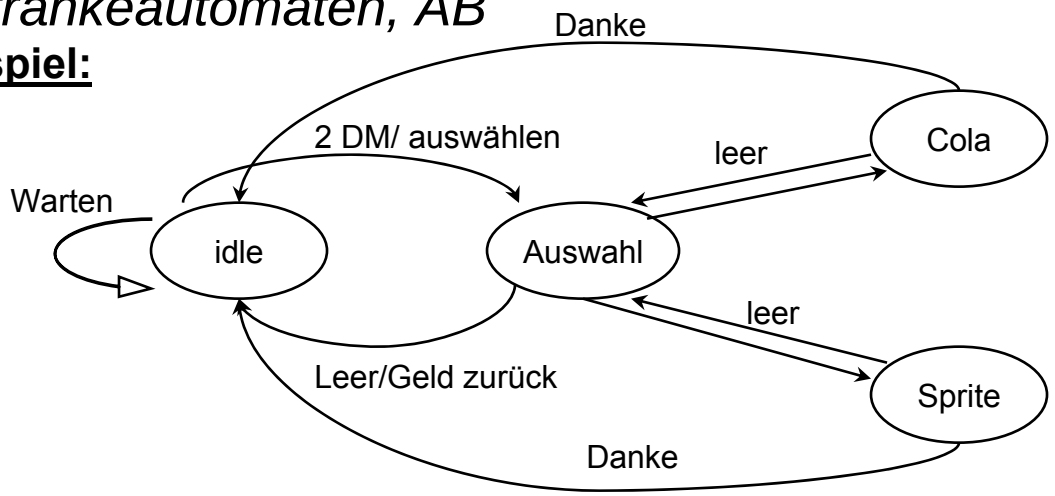


Realisierung als Moore-Automat

Endliche Automaten (FSM)

Endliche Automaten sind Maschinen , die auf eine Eingabe hin bestimmte Ausgaben erzeugen. Beispiel: Bankautomaten, Getränkeautomaten, AB

Beispiel:



- Ein endlicher Automat besteht aus Speicherelementen für den aktuellen Zustand und Logikelementen zur Festlegung des Folgezustandes und der Ausgabewerte.
- Jedem Zustand der State Machine wird ein Binärcode zugeordnet
- Unterteilung der FSM in Register und kombinatorische Logik spiegelt sich exakt in der VHDL-Beschreibung wieder.

VHDL-Beschreibung einer FSM

```
architecture RTL of FSM is
type state_typ is (idle, state1, state2, state3);
signal current_state, next_state : state_typ;
attribute ENUM_ENCODING : string;
attribute ENUM_ENCODING of state_type : type is "00 01 10 11";
attribute STATE_VECTOR : string;
attribute STATE_VECTOR OF RTL is: architectur "current_state";

begin

FSM_body : process (current_state)
begin
        case current_state is
        when idle =>
        Ausgangsport1 <= '0'; Ausgangsport2 <= '0'; .....
        next_state <= state1;

        when state1 =>
        Ausgangsport1 <= '0'; Ausgangsport2 <= '1'; .....
        next_state <= state2;

        when others =>
        Ausgangsport1 <= '0'; Ausgangsport2 <= '0'; .....
        next_state <= idle;
```

Für die Zustände der FSM wird ein neuer Datentyp deklariert.

Mit dem *STATE_VECTOR* Attribut ist der DC in der Lage, den Zustandsvektor zu identifizieren. *ENUM_ENCODING* erlaubt die feste Zuordnung binärer Zustandsvektoren zu den angegebenen Zuständen.

In diesem Prozeß werden in Abhängigkeit vom aktuellen Zustand der Folgezustand und die Ausgabewerte berechnet.

VHDL-Beschreibung einer FSM

```
Reg: process (clk, reset)
begin;
   if reset = '1' then
         current_state <= idle;
   if clk'event and clk = '1' then
         current_state <= next_state;
   end if;
end process;

end RTL;

configuration FSM_CFG of FSM is
for RTL
end for;
```

Prozeß zur Speicherung des aktuellen Zustandes.

Register verfügt - wie alle Register - über einen asynchronen Reset-Eingang.

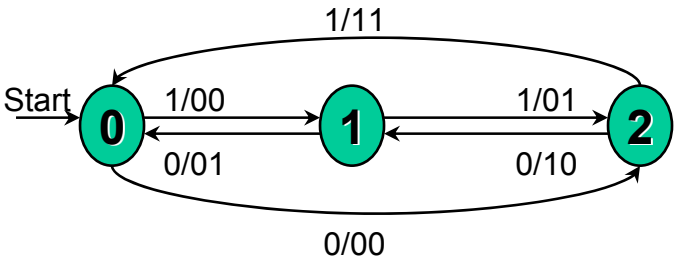
Zusätzlich können/werden hier die Ausgangsregister beschrieben

Werden in der Beschreibung der Architektur Submodule verwendet, müssen diese in Konfiguration den Instanzen die VHDL-Beschreibungen zugewiesen werden.

Zustandscodierung

- Jedem möglichen Zustand der State Machine ist ein bestimmter Binärcode zugeordnet (state encoding)
- n Register kodieren maximal 2^n Zustände
- One Hot Methode: Jedem Zustand wird exakt ein Register zugeordnet
- n Register kodieren n Zustände
- keine kombinatorische Logik zur Dekodierung des aktuellen Zustandes
- schnellste Implementierungsvariante

Beispiel



- Der Automat besitzt drei Zustände
- Die Ausgabedaten hängen vom Zustand ab => Moore Automat
- Pfeile zeigen die Übergänge von gegenwärtigen zum nächsten Zustand an
- Die Beschriftungen kennzeichnen die Übergangs-bedingung und die zu generierenden Ausgangsdaten (Eingabe/ Ausgabe)

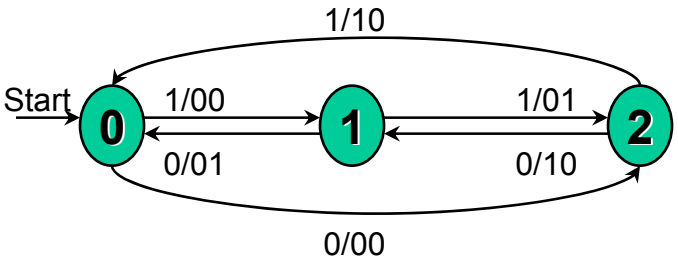
Ports der Entity:

- Input (1 Bit breit), Start (1 Bit breit)
- Output (2 Bit breit)
- Zusätzlich clk und reset, da FSMs immer über Zustandsspeicher verfügen.

Configuration:

- Die Configuration der FSM ist leer, da keine Komponenten instanziiert werden müssen

Beispiel



Das Zustandsdiagramm läßt sich eindeutig auf eine VHDL-Beschreibung abbilden

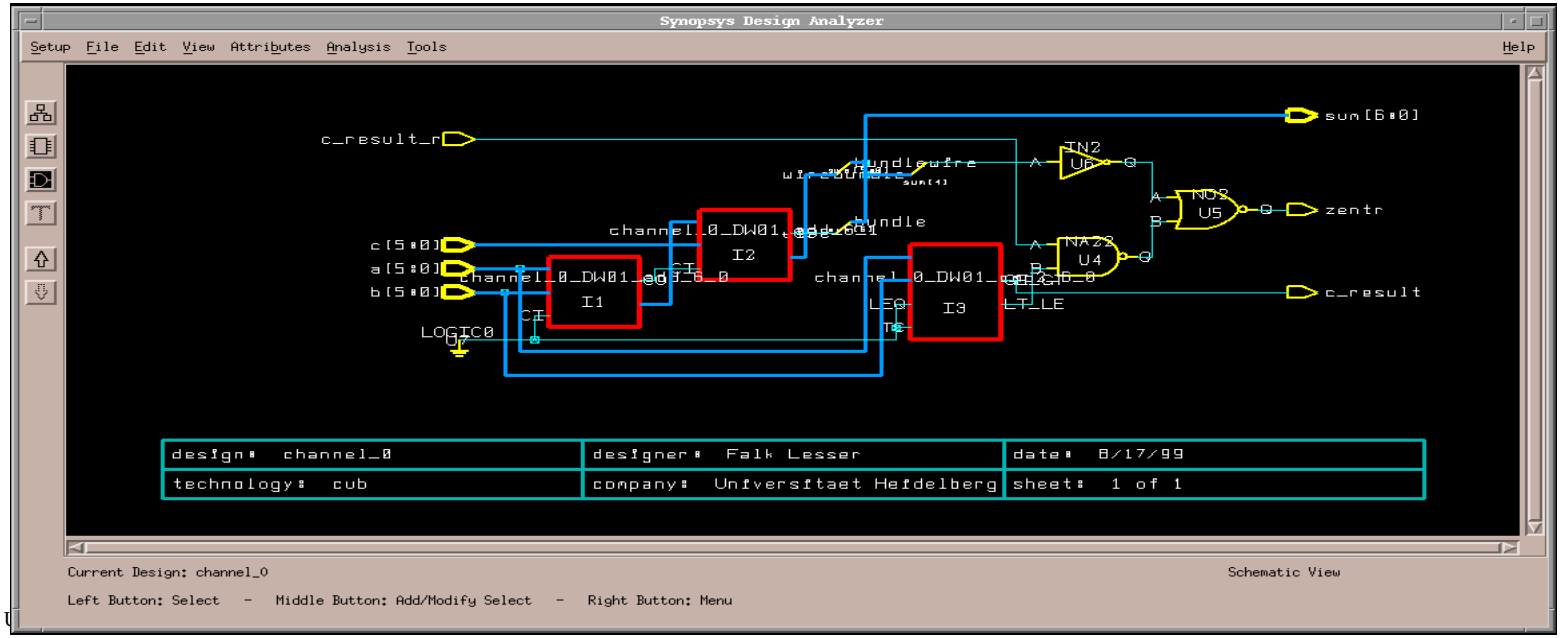
```
architecture behav of fsm is

type state_type is (state_0, state_1, state_2)
signal current_state, next_state                    : state_type;

begin

state_decoder : process (current_state, eingabe, start)
begin
    case current_state is
            when state_0 =>
            if start = '1' then
                    if eingabe = '0' then
                            ausgabe = "00";
                            next_state <= state_2;
                    elsif eingabe = '1' then
                            ausgabe = "00";
                            next_state <= state_1;
                    end if;
            else
            next_state <= state_0;
            end if;

            when state_1 =>
                    if eingabe = '0' then
                            ausgabe = "01";
                            next_state <= state_0;
                    elsif eingabe = '1' then
                            ausgabe = "01";
                            next_state <= state_2;
                    end if;

            when state_2 =>
                    if eingabe = '0' then
                            ausgabe = "10";
                            next_state <= state_1;
                    elsif eingabe = '1' then
                            ausgabe = "10";
                            next_state <= state_0;
                    end if;
            end case;

end process;
end behav;
```

Erweiterte Synthesestrategien und physikalischer Test

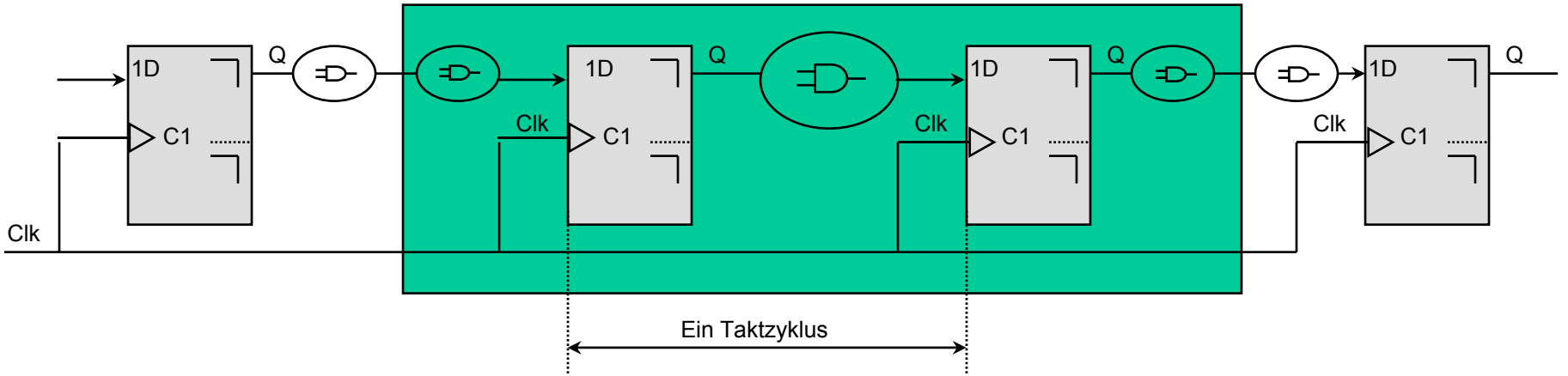
Volker Lindenstruth, Falk Lesser
Institut für Hochenergiephysik

Design Objekts

Design:	Die gesamte Schaltung (VHDL-Beschreibung)
Cell:	Eine Instanz innerhalb eines Designs bzw. ein Baustein aus einer Bauteilebibliothek.
Reference:	Referenz zu einer Instanz.
Port:	Die Inputs und Outputs eines Designs.
Pin:	Die Inputs und Outputs einer Instanz.
Net:	Die Verbindungsleitungen für Ports und/oder Pins.
Clock:	Eine Zeitreferenz. Beschreibt den Signalverlauf zur Zeitanalyse.



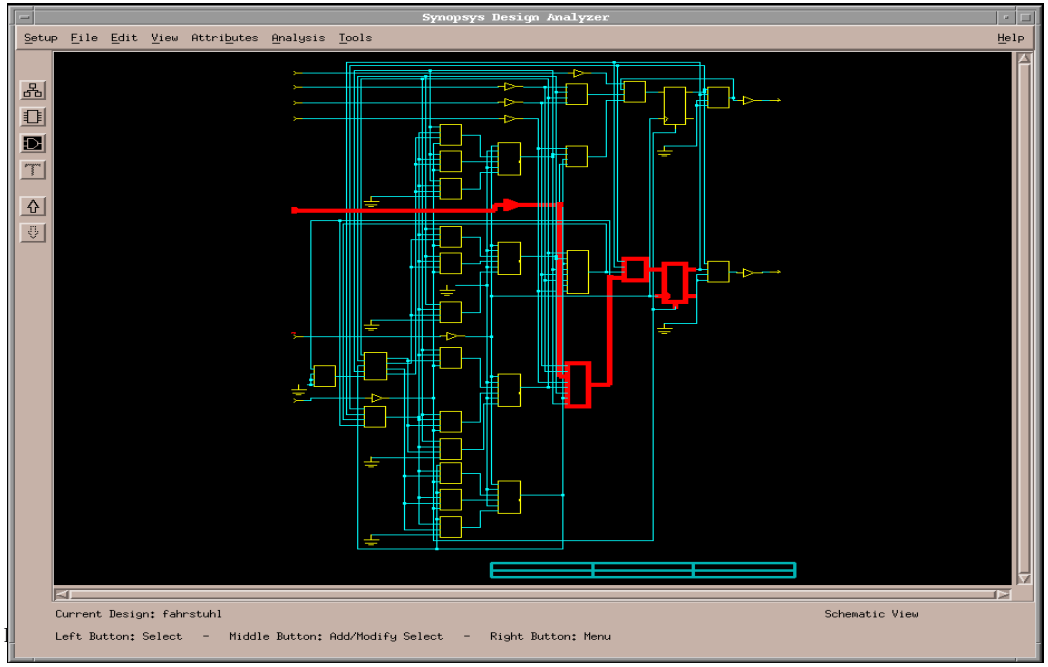
Timing Constraints



- Timing Constraints sind ausschließlich in synchronen Designs sinnvoll.
- Synchron: Alle Start- und Endpunkte eines kombinatorischen Pfades beginnen und enden an einem Register. Diese werden von der gleichen Clock angesteuert.
- Nach Eingabe eines Timing-Constraints versucht das Synthesewerkzeug das Funktionale Verhalten der Schaltung unter Einhaltung der Zeitvorgabe aus Bauteilen der Zieltechnologie aufzubauen.
- Der kritische Pfad wird aus den Verzögerungszeiten der Bauteile bestimmt.
  - Startpunkte sind: Input Ports und Clock Pins von Flip-Flops oder Registern
  - Endpunkte sind: Output Ports und Dateninputs sequentieller Module

Timing Report

- Nach erfolgter Synthese kann der kritische Pfad vom Design Compiler berechnet werden.
- Der kritische Pfad gibt den längsten Signalverlauf an.
- Aus Ihm kann die maximale Taktfrequenz für das Gesamtsystem bestimmt werden.
- Er zeigt den Start- und Endpunkt des kritischen Pfades.
- Durch geschickte Wahl bestimmter Design-Constraints kann das Syntheseergebnis u.U. stark beeinflußt werden.



Report Output

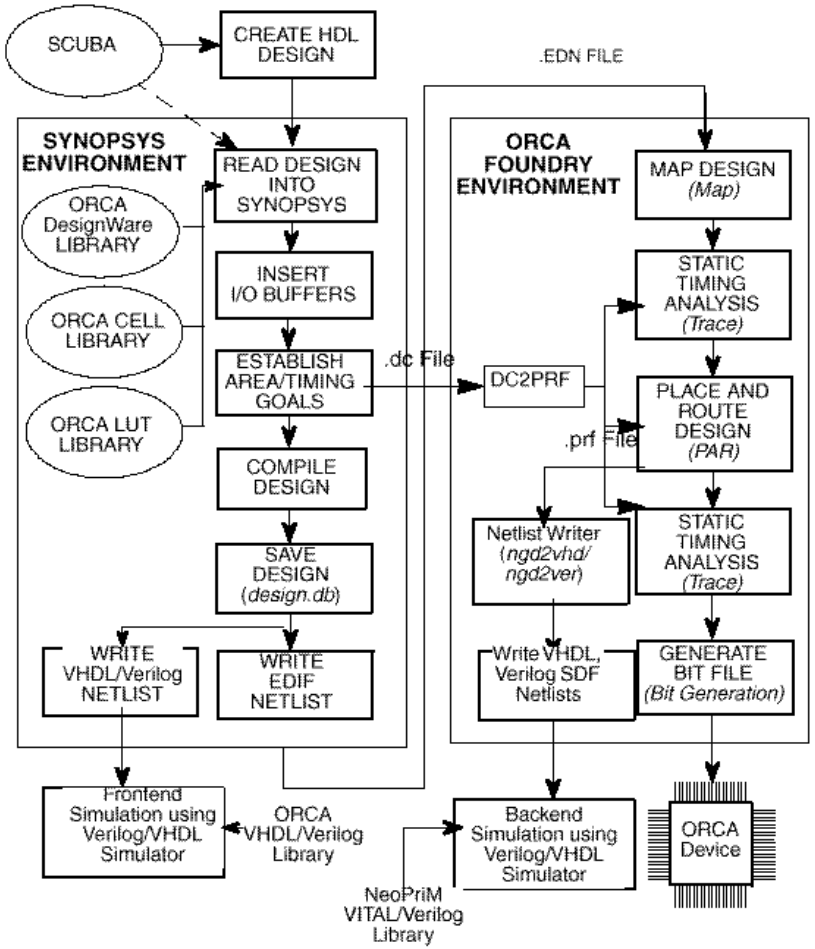
```
design_analyzer> Warning: Design 'fahrstuhl' has '1' unresolved references. For more detailed i
Information: Updating design information... (UID-85)

****************************************
Report : timing
         -path full
         -delay max
         -max_paths 1
Design : fahrstuhl
Version: 1998.08
Date   : Fri Sep  3 08:26:43 1999
****************************************

Operating Conditions: nom_25_5   Library: OR3LUT-5.lib
Wire Loading Model Mode: top

Design          Wire Loading Model      Library
------------------------------------------------
fahrstuhl             .10               OR3LUT-5.lib


  Startpoint: etage_2 (input port)
  Endpoint: current_state_reg_0
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Point                                                   Incr       Path
  ---------------------------------------------------------------------------
  clock (input port clock) (rise edge)                    0.00       0.00
  input external delay                                    0.00       0.00 r
  etage_2 (in)                                            0.00       0.00 r
  U89/O (IBT)                                             4.38       4.38 r
  lt_94/lt/lt/a_2 (fahrstuhl_OR3_LT_TC_4_0)               0.00       4.38 r
  lt_94/lt/lt/FSUB4_0/BO (FSUB4)                          4.10       8.48 r
  lt_94/lt/lt/U2/Z (ORCALUT4)                             3.41      11.90 r
  lt_94/lt/lt/altb (fahrstuhl_OR3_LT_TC_4_0)              0.00      11.90 r
  U21/Z (ORCALUT4)                                        2.85      14.74 r
  current_state_reg_0/D (FD1S3DX)                         0.00      14.74 r
  data arrival time                                                 14.74

  clock clk (rise edge)                                  50.00      50.00
  clock network delay (ideal)                             0.00      50.00
  current_state_reg_0/CK (FD1S3DX)                        0.00      50.00 r
  library setup time                                     -0.32      49.68
  data required time                                                49.68
  ---------------------------------------------------------------------------
  data required time                                                49.68
  data arrival time                                                -14.74
  ---------------------------------------------------------------------------
  slack (MET)                                                       34.94


1
design_analyzer>
```

Show Next Previous Cancel

Zweiter Simulationsdurchlauf

- Nach der Synthese kann die Schaltung ein weiteres Mal getestet.
- Diesmal werden die Verzögerungszeiten und elektrischen Parameter der Logikzellen mitberücksichtigt
- Das tatsächliche Verhalten kann jetzt „hardwarenah“ simuliert werden
- Laufzeiten des Verbindungsnetzwerkes bleiben dabei unberücksichtigt. Diese können der Simulation durch Definition von Setup-Zeiten hinzugefügt werden.

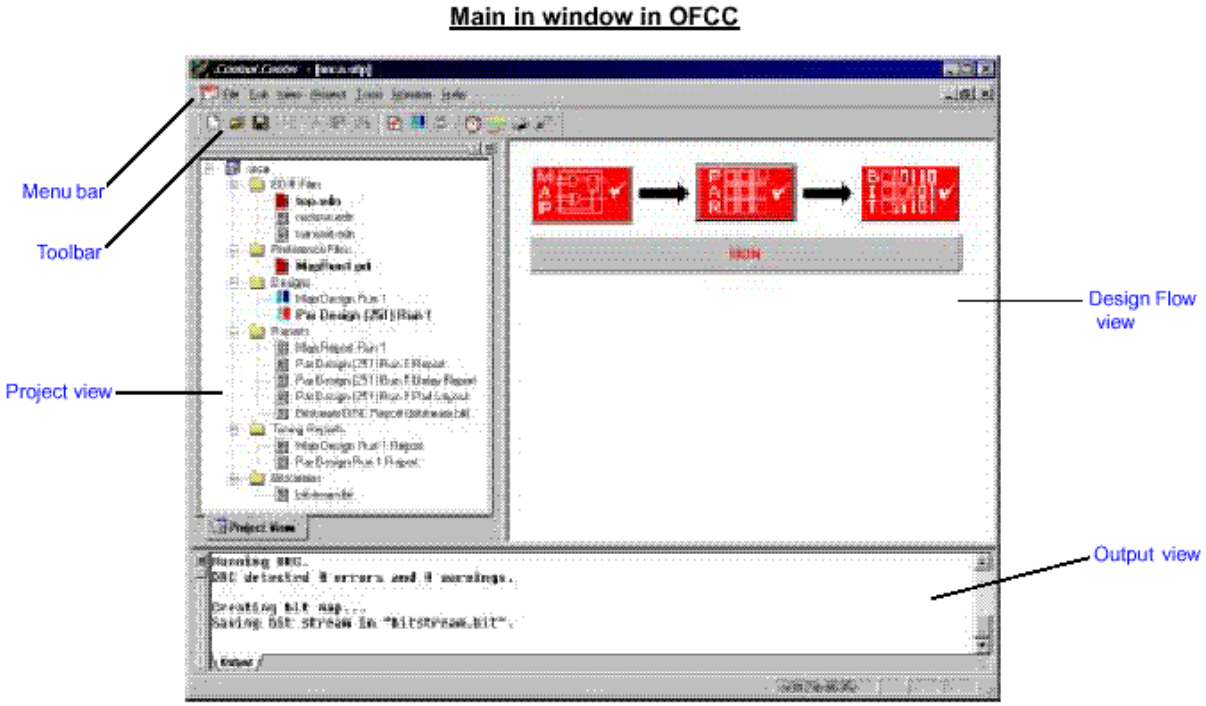
Orca-Designflow



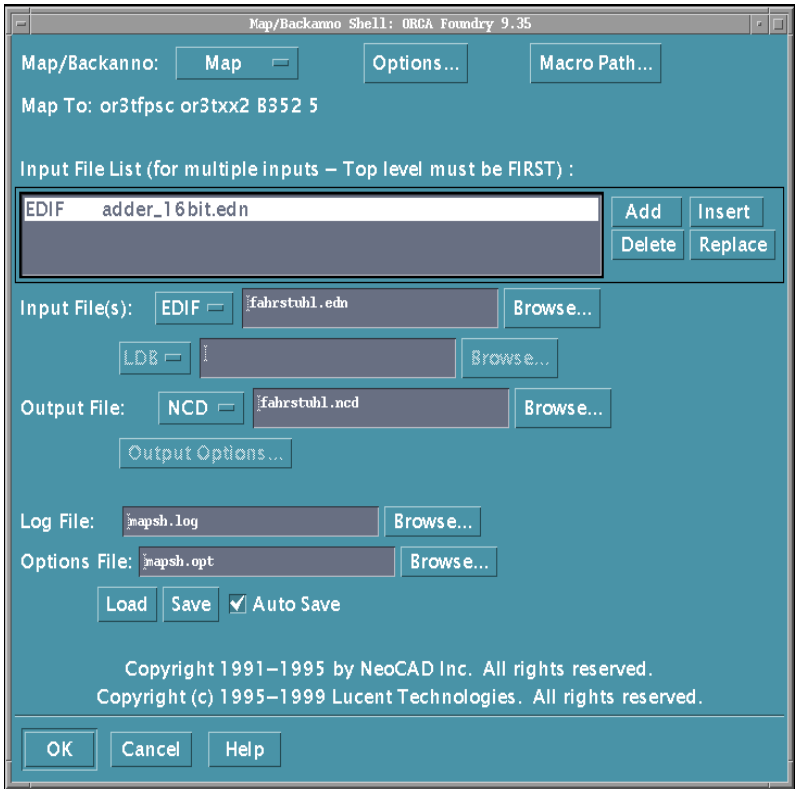
- Der Designflow beinhaltet die Elemente
  - **Mapping (mapsh)**
  - **Place and Route (parsh)**
  - **Erzeugung eines Bitstreams**
- Diese und weitere Synthesetools sind im Orca Foundrary Control Center zusammengefast.
- Dazwischen liegen Analyseschritte um das Syntheseergebnis zu verifizieren. Hierfür werden von den einzelnen Synthesetools Reports erstellt.
- Anschließend erfolgt das Programmieren des FPGA.
- Der Funktionale Test wird direkt am Logik-baustein durchgeführt.

Orca Foundary Control Center (ofcc)

- Integration aller benötigten Synthesewerkzeuge des Orca-Designflows
- Jeder Schritt der Synthese wird ausführlich dokumentiert (Reports)
- Iteratives Vorgehen möglich



Mapshell



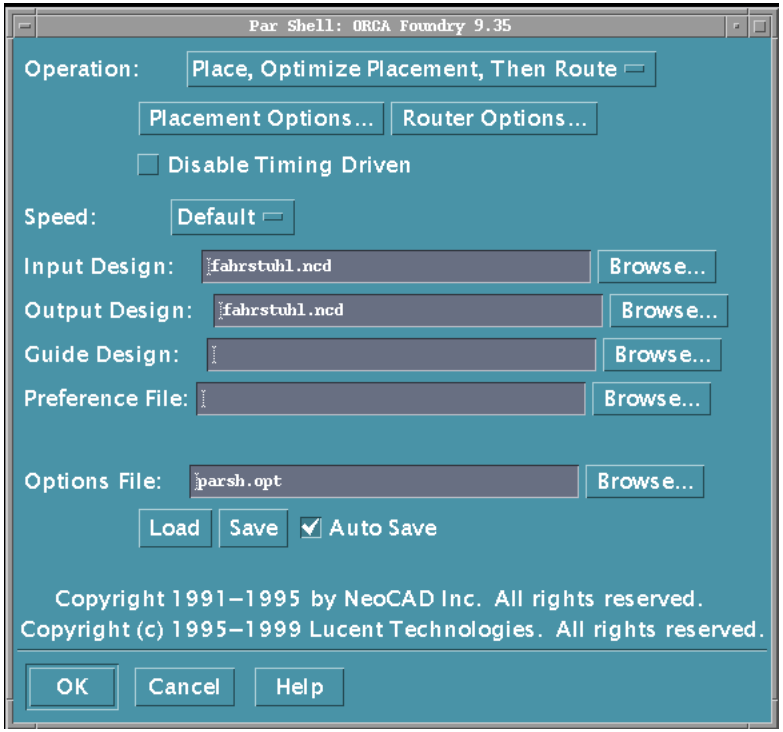
- Die Mapshell erzeugt aus der EDIF 200 Netzliste eine physikalische Beschreibung (*.ncd) der verwendeten Komponenten der Zieltechnologie (Orca-Lib).
- **Aufruf des Programms: $> mapsh &**

Optionen:

- Architecture : or3tfpsc
- Device: or3txx2
- Package: B252
- Speed: 5

- Das erzeugte *.ncd-File dient als Input für das Place and Route Tool.

Place and Route



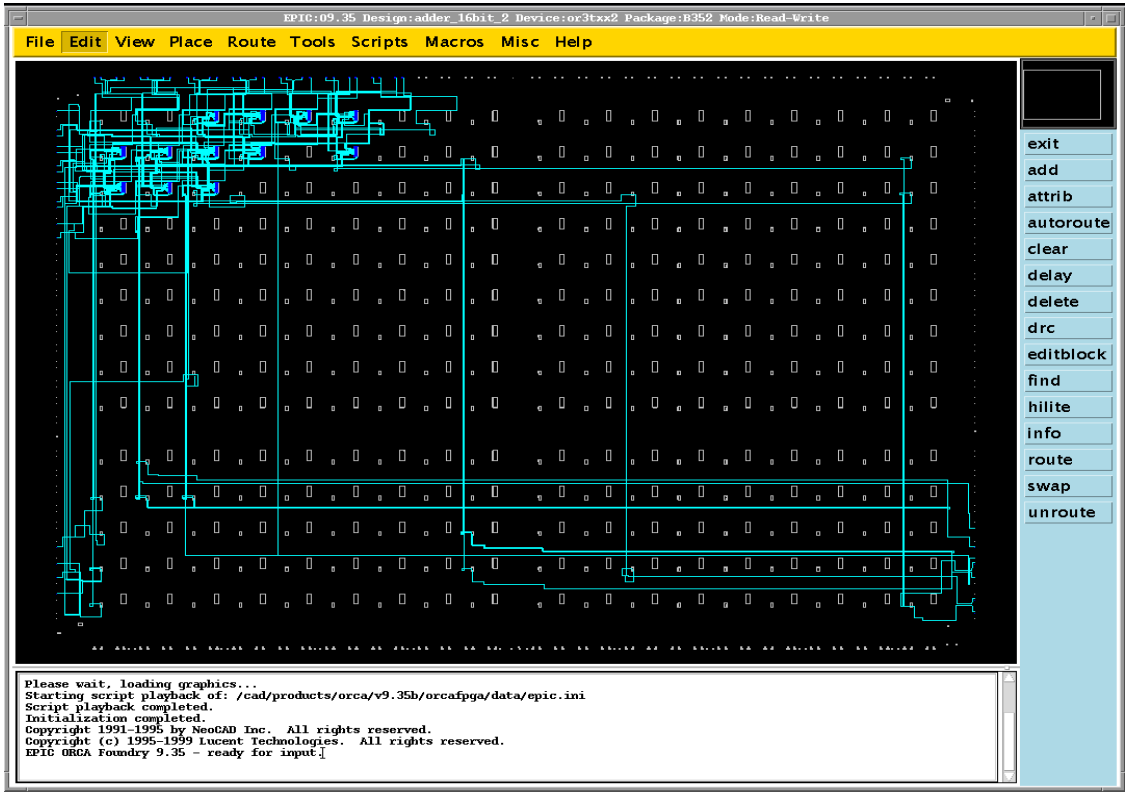
- Mit Parshell werden die Logikzellen Plaziert und Verdrahtet.
- **Aufruf des Programms: $> parsh &**

Optionen:

- Es sind die Defaultwerte zu verwenden (*cost-based*)
- Preference File: <filename>.prf

Editor für Programmierbare ICs

- Über ein GUI können interne Signale (z.B. interne Register) auf Ouput-Pins gelegt werden. Auf diese Weise kann das funktionale Verhalten der Schaltung nach "Außen" und "Innen" getestet werden.



- Mit epic können die Logikzellen von Hand Plaziert und Verdrahtet werden
- **Aufruf : $> epic &**

Optionen:

- keine besonderen Optionen zu beachten
- Mehr Infos gibt's bei der Übung

Statische Timinganalyse (trcesh)

- Die Traceshell ist die graphische Benutzer-oberfläche zur Durchführung der statischen Timinganalyse.
- Statische Timinganalyse bedeutet, daß anhand der Verzögerungszeiten der verwendeten Logikbausteinen ein kritischer Pfad berechnet werden kann.
- **Aufruf des Programms: $> trcesh &**

Optionen:

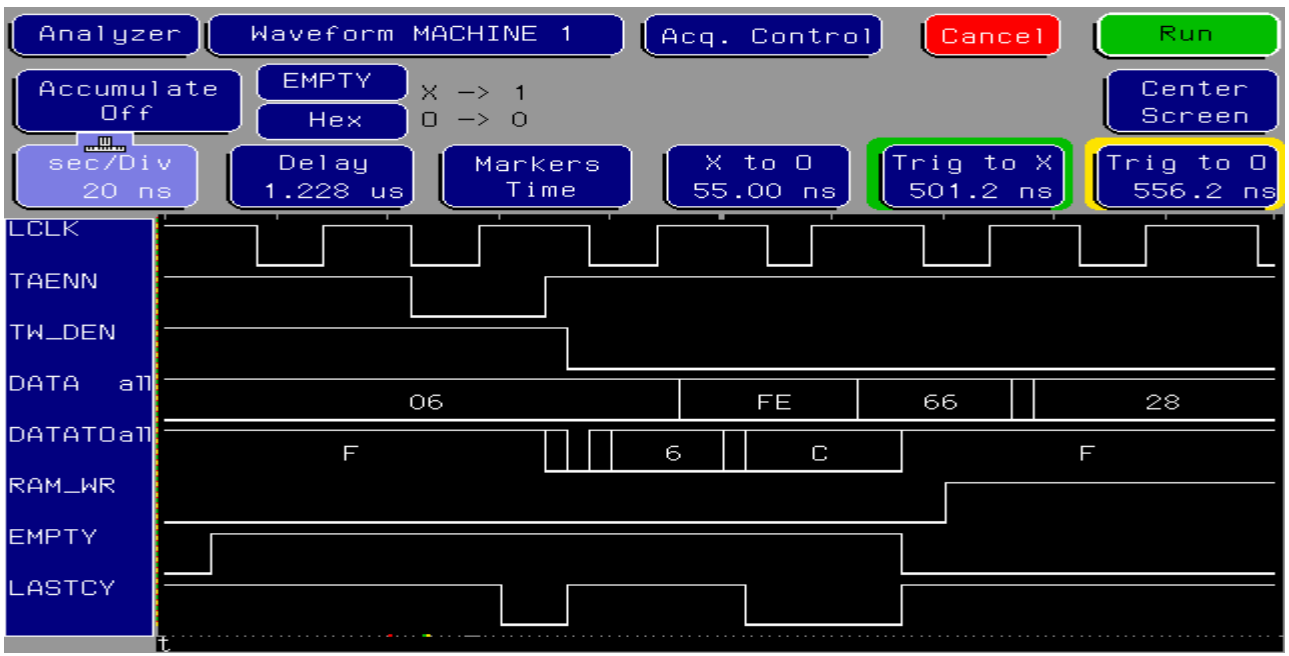
- Architecture : or3tfpsc
- Worst Case Path: 1
- Timing Report: Verbose
- Speed: 5
- Preference file: <filename>.prf
- Das erzeugte *.twr-File enthält die Verzögerungs-zeiten des kritischen Pfades.

Vorher mit dem folgendem Kommando ein Preferencefile erzeugen:

$> dc2prf -e file.edn -m file.mrp -n file.ncd -p file.prf -d compile.dc

Realer Test am FPGA

- Der Test der Schaltung wird direkt am FPGA durchgeführt.
- Die Ausgabedaten werden am Testboard abgegriffen und durch den Logik-Analysator dargestellt.
- Der dargestellte zeitliche Verlauf entspricht dem realen Verhalten!



Ansicht der Signalverläufe auf dem Logik-Analysator

Zusammenfassung

- FSMs sind das zentrale Element für das Design schneller Hardware.
- Design-Constraints sind Bedingungen, die das Synthesewerkzeug zwingen gewünschte Eigenschaften zu erfüllen.
- Der Syntheseablauf ist weitgehend automatisiert.

Inhalt des fünften Tages

- Tips und Tricks
- Fragen
- Selbständige Entwurfs- und Syntheseaufgabe

Tips und Tricks

- Die Architektur der Schaltung **vor** der Implementierung entwerfen
- Komplexe Entwürfe möglichst modular gestalten
- Die Gesamtaufgabe in sinnvolle Teilaufgaben aufteilen
- Datenpfade und Steuereinheiten voneinander trennen
- Arithmetische Einheiten sind die komplexesten Einheiten, deshalb sollten diese Elemente mehrfach verwendet werden.
- Möglichst ein global gültiges Taktsignal verwenden
- Signalwege innerhalb von Modulen kurz halten. Ausgabewerte enden in Registern.
- PIN-Count bedenken
- Sourcecode ausreichend dokumentieren !

Tips und Tricks

Einige wichtige Synthesekommandos zur Erstellung von Syntheseskripts für den Synopsys FPGA-Analyzer

- alias : Def. von Aliasen
- compile -map_effort : Mappen auf Tech.
- create_clock : Takt erzeugen
- find : Designobjekte suchen
- foreach : Schleifenanweisung
- history : Liste der letzten Befehle
- include : Skript einbinden
- insert_pads : Pads einfügen
- remove_* : Attribute entfernen
- report_* : Ausgabe
- set_clock_latency : Latenzzeit definieren
- set_dont_touch_* : Objekt isolieren
- set_input_delay : Latenzzeit
- set_max_area : Area Constrain
- set_operating_cnditions : Umgebungsbed.
- set_output_delay : Latenzzeit
- set_pad_type : Padtyp
- set_wire_load : Wire-Model
- ungroup : Design flach
- uniquify : auflösen
- while : Schleifenbefehl

Tips und Tricks

Einbinden von Komponenten aus definierten Bauteilebibliotheken:

8 Bit Counter mir synchronen Set und asynchronen Reset

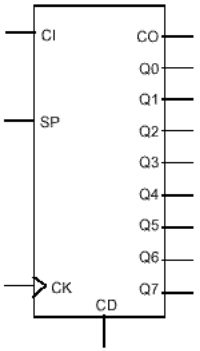
Beispiel Orca-Designelements:

```
architecture structural of test_counter is

component CU8P3DX
port (      CI, SP, CK, CD                          : in std_logic;
            CO, Q0, Q1, Q2, Q3, Q4, Q5, Q6, Q7 : out std_logic);
end component;

begin

counter_instance: CU8P3DX
port map (  CI_sig, SP_sig, CK_sig, CD_sig, CO_sig, Q0_sig,
                        Q1_sig, Q2_sig, Q3_sig, Q4_sig, Q5_sig, Q6_sig,
                        Q7_sig)

end structural;
```

Truth Table:

INPUTS				OUTPUTS	
CI	SP	CK	CD	CO	Q[0:7]
X	X	X	1	0	0
0	X	X	0	0	Q[0:7]
1	0	X	0	*	Q[0:7]
1	1	↑	0	*	count+1

X = Don't care
* When Q[0:7] is 11111111, CO will be 1; otherwise, CO will be 0
When GSR=0, Q[0:7]=0 (SP=CK=CD=X)



CU8P3DX

INPUTS: CI,SP,CK,CD
OUTPUTS: CO,Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7
PINORDER: CI SP CK CD CO Q0 Q1 Q2 Q3 Q4 Q5 Q6 Q7
MINIMUM CELL AREA: 0.5

Reihenfolge der Ports

8 Bit Addierer mit Carry-In und Carry-Out

Beispiel Synopsys-Designwarebibliothek:

```
architecture structural of test_adder is

component DW01_add
generic (width:          integer)
port (       A, B        : in  std_logic_vector(width-1 downto 0);
             CI          : in  std_logic;
             SUM         : out std_logic_vector(width-1 downto 0);
             CO          : out std_logic);
end component;

begin

adder_instance: DW01_add
generic map (witdth => 8)
port map (   A => A_sig, B => B_sig, CI => CI_sig, CO => CO_sig,
             SUM => SUM_sig);
end structural;
```

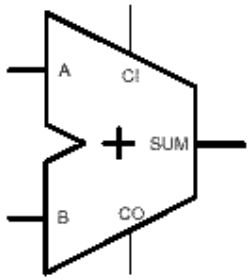


Table 1 – Pin Description

Pin Name	Size	Type	Function
A	width	Input	Input data
B	width	Input	Input data
CI	1	Input	Carry-in
SUM	width	Output	Sum of A + B
CO	1	Output	Carry-out

Table 2 – Parameter Description

Parameter	Function	Legal Range
width	Word length of A, B, and SUM	≥ 1

Fragen und Kritik

- Hardwaredesign ist cool!!!!!!

Selbstständige Entwurfsaufgabe

- Entwurf einer Digitaluhr mit Weckfunktion
- Vollständiger Designflow