

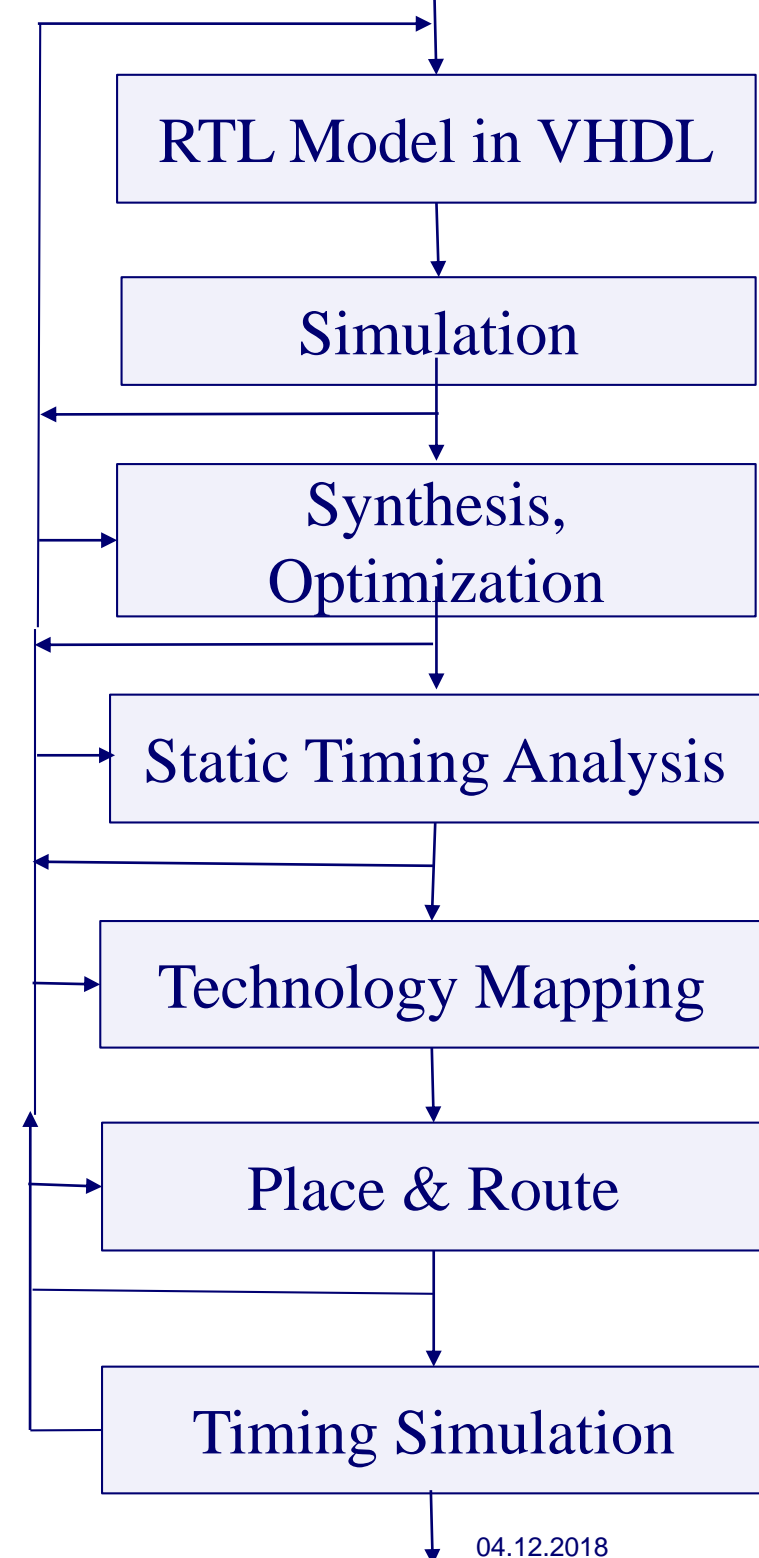
# Inhaltsübersicht

---

- **Warum Modellierung und Hardwarebeschreibungssprachen (HDLs) ?**
- **VHDL-Einführung**
- **Logikminimierung**
- **Physikalische Implementierung**
- **Datenpfadkomponenten**
- **Latches und Flipflops**
- **Entwurf synchroner Zustandsautomaten**
- **Entwurf von Synchronzählern und Schieberegistern**
- **Programmierbare Logik**
- **Digitale Halbleiterspeicher**

# Entwurfsprozess

- Die Simulation dient der **Validierung bzw. Verifikation**:
  - **Funktionale VHDL-Simulation**: Überprüfung der Entwurfsidee.
  - **Formale Verifikation**: Überprüfung von Schlüsseigenschaften
  - **Äquivalenzprüfung** zwischen verschiedenen Modellen
- Verifikation des Zeitverhaltens durch
  - **Statische Timing Analyse**
  - **Postlayout VHDL-Timing-Simulation**

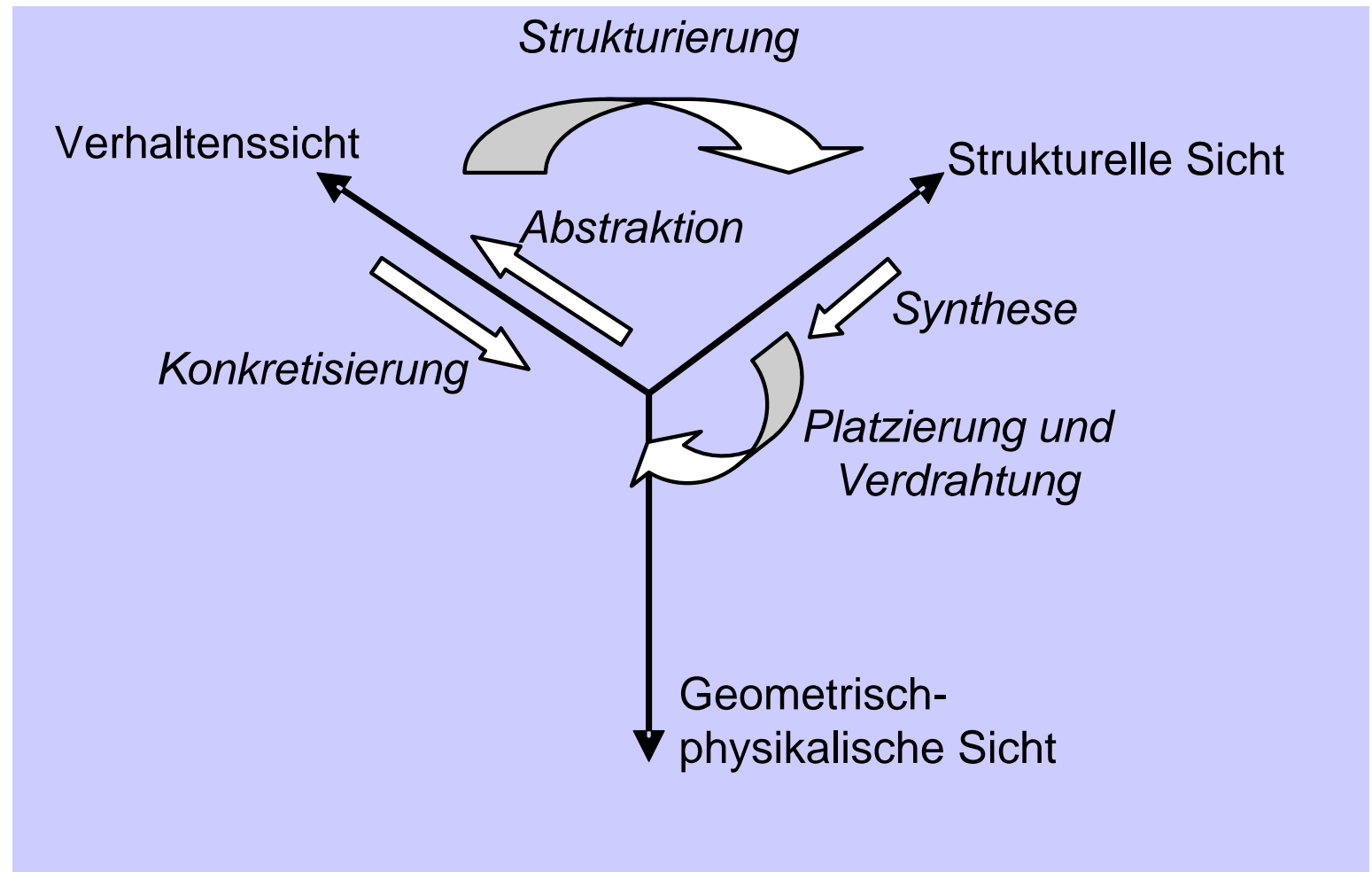


# Einsatz von Simulations- und Synthesewerkzeugen

- Werkzeuge zur **Simulation, Synthese, und Analyse**.
- Modelle: C, C++, SystemC, VHDL, Verilog, PSL, GDSII, Spice, ...

- **Domänen des Entwurfsprozesses:**

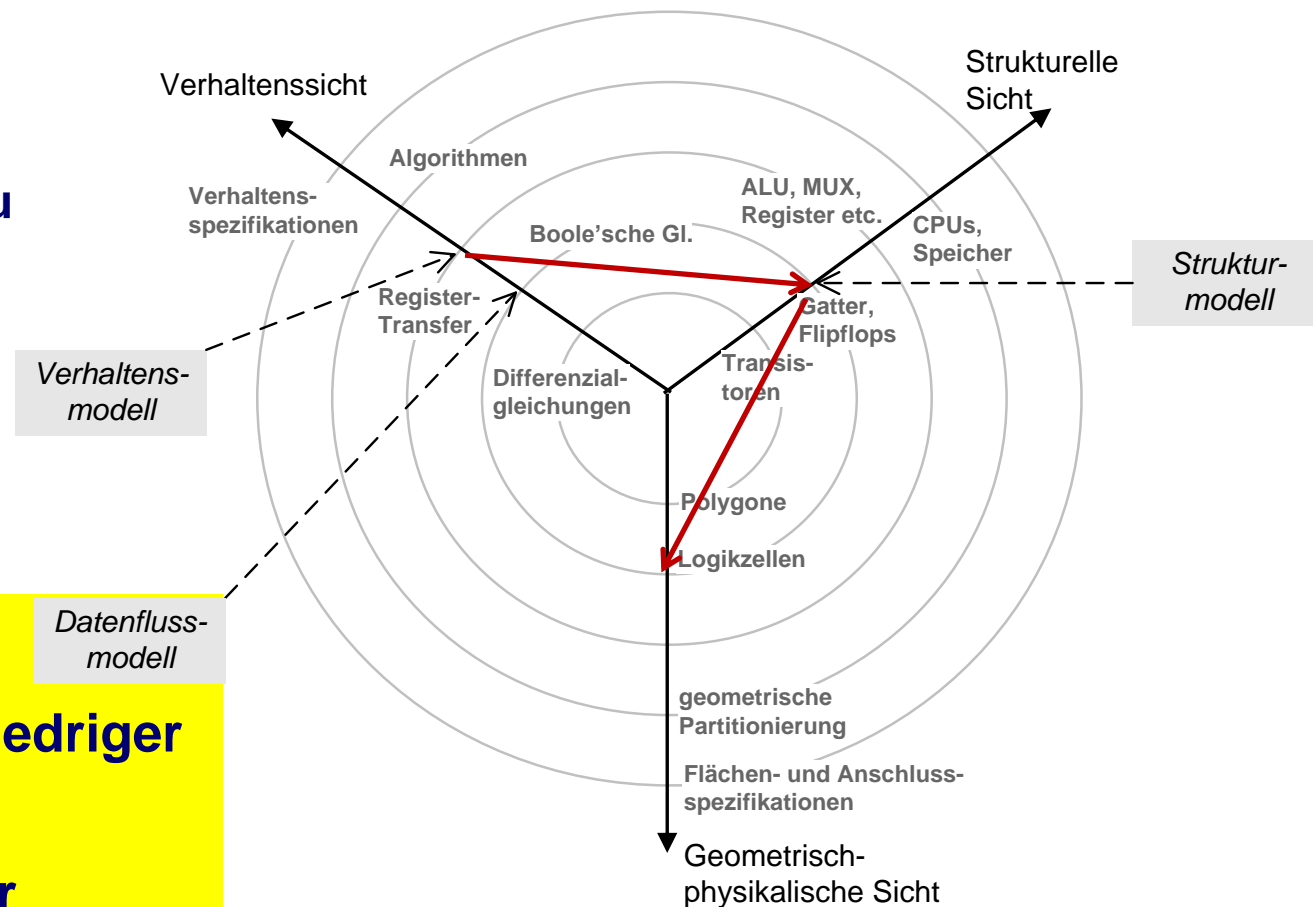
- Verhalten,
- Strukturierung,
- Geometrie.



Daniel D. Gajski, UC Irvine

# Schritte im Entwurfsprozess

- Der Entwurfsprozess erfolgt von außen nach innen. Dabei werden in verschiedenen Phasen die Domänen gewechselt.
- Beim Digitalentwurf sind verschiedene Entwurfsschritte zu unterscheiden:



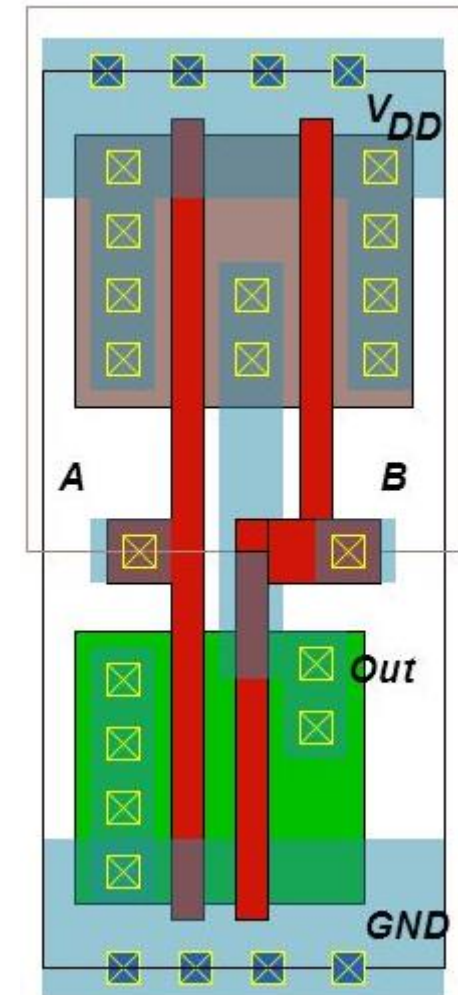
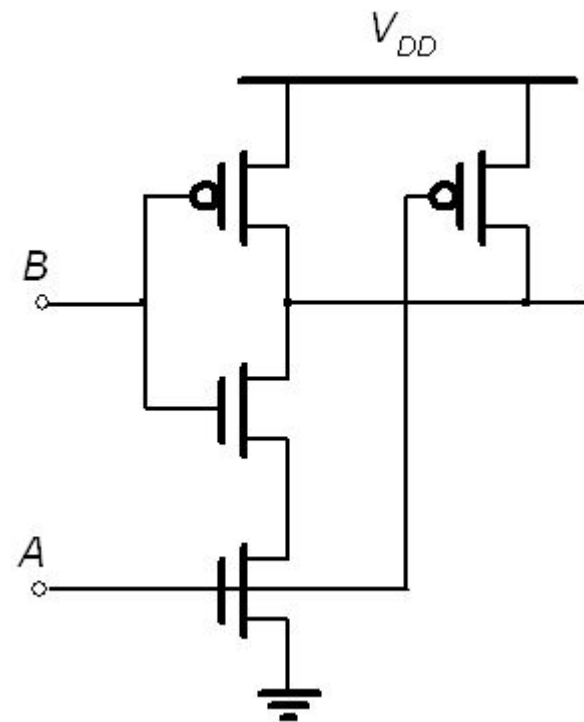
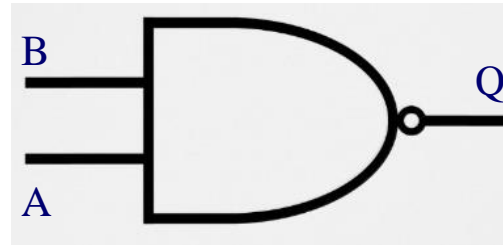
## • **Synthese:**

- Übergang von höherer zu niedriger Abstraktion
- Übergang vom Verhalten zur Struktur

• **Place and route:** Übergang von Struktur zur Geometrie

# Ein NAND Gate

```
architecture A1 of NAND is
begin
    Q <= not (A and B);
end A1;
```



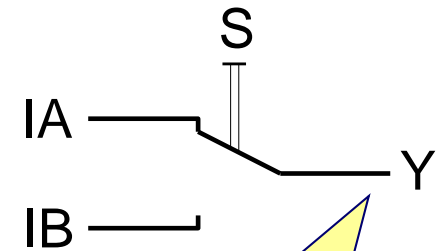
Verhalten

Struktur

Geometrie

# Modellierung mit Hardwarebeschreibungssprachen

- **BEISPIEL Multiplexer (MUX)**
- **Verschiedene Modellierungsstile:**
  - **Datenflussmodelle**
  - **Strukturmodelle**
  - **Verhaltensmodelle**



**Das Ausgangssignal Y erhält den Wert von IA UND wenn gleichzeitig  $S=0$  ist ODER den Wert von IB UND wenn gleichzeitig  $S=1$  ist.**

# Datenflussmodelle

- ***Datenflussmodell:*** Der Fluss der Daten durch die verarbeitenden Komponenten ist explizit.

```
architecture DATENFLUSS of MUX is  
  
begin  
    Y <= (IB and S) or (IA and (not S));  
end DATENFLUSS;
```

- Im Gajski-Diagramm findet sich diese Art der Modellierung auf der, von innen gesehen, zweiten Abstraktionsebene der Verhaltensachse.

# Strukturmodelle

- **Strukturmodell:** Komponenten und ihre Verbindungen sind explizit.

Bibliothek von Komponenten (**VHDL-Default-Bibliothek:** `./work`).

```
architecture STRUKTUR of MUX is
  signal NODE1, NODE2, NODE3 : bit;
begin
  U1: UND port map (IB, S, NODE1);
  U2: INVERTER port map (S, NODE2);
  U3: UND port map (NODE2, IA, NODE3);
  U4: ODER port map (NODE1, NODE3, Y);
end STRUKTUR;
```

Die Komponenten UND, INVERTER, ODER wurden zuvor kompiliert und in der work-Bibliothek abgelegt

- Im Gajski-Diagramm findet sich diese Art der Modellierung ebenfalls auf der, von innen gesehen, zweiten Abstraktionsebene allerdings auf der Strukturachse.



# Verhaltensmodelle

- **Verhaltensmodell** : Algorithmische Beschreibung; Der Kontrollfluss ist explizit.
- Abgeschlossene Hardware-Funktionsblöcke werden durch Prozesse abgebildet. Innerhalb von Prozessen sind u.a. Schleifen und bedingte Verzweigungen erlaubt.

```
architecture VERHALTEN of MUX is
begin
  P1: process (IA, IB, S)
  begin
    if S = '1' then
      Y <= IB;
    else
      Y <= IA;
    end if;
  end process P1;
end VERHALTEN;
```

- Im Gajski-Diagramm findet sich diese Art der Modellierung auf der, von innen gesehen, dritten Abstraktionsebene der Verhaltensachse.

# entity und architecture

- **entity**

- Deklaration aller Schnittstellen einer Entwurfseinheit nach außen;
- Parametrisierbar

- **architecture**

- beschreibt die Funktionalität der **entity**.
- Jeder **entity** muss mindestens eine **architecture** zugeordnet sein.
- Eine **entity** kann mehrere Architekturrealisierungen haben.

- **VHDL-Bezeichner:** VHDL ist nicht case-sensitiv. Erlaubt sind alle alphanumerischen Zeichen (ohne Umlaute), sowie der Unterstrich `_` als Sonderzeichen. Das erste Zeichen muss alphabetisch sein.

Vereinbarung: selbstdefinierte Bezeichner werden groß geschrieben!

# entity und architecture

```
entity MUX is  
port( IA, IB, S : in bit;      -- Eingangssignale  
      Y : out bit);           -- Ausgangssignale  
end MUX;
```

```
architecture DATENFLUSS of MUX is  
  
begin  
Y <= (IB and S) or (IA and (not S));  
end DATENFLUSS;
```

```
architecture STRUKTUR of MUX is  
signal NODE1, NODE2, NODE3 : bit;  
begin  
  U1: UND port map(IB, S, NODE1);  
  U2: INVERTER port map(S, NODE2);  
  U3: UND port map(NODE2, IA, NODE3);  
  U4: ODER port map(NODE1, NODE3, Y);  
end STRUKTUR;
```

```
architecture VERHALTEN of MUX is  
begin  
P1: process(IA, IB, S)  
begin  
  if S = '1' then  
    Y <= IB;  
  else  
    Y <= IA;  
  end if;  
end process P1;  
end VERHALTEN;
```

# Grundlegende Syntaxelemente

```
entity <ENTITY_NAME> is
    port( {{<PORT_NAME_i>} : <mode> <type_1>;}
        );
end <ENTITY_NAME>;
```

- **Einfachste Port-Datentypen:** `bit`, `bit_vector` (Wertevorrat: 0, 1)
- **Bitvektoren sind Busse / Signalbündel z.B.:**

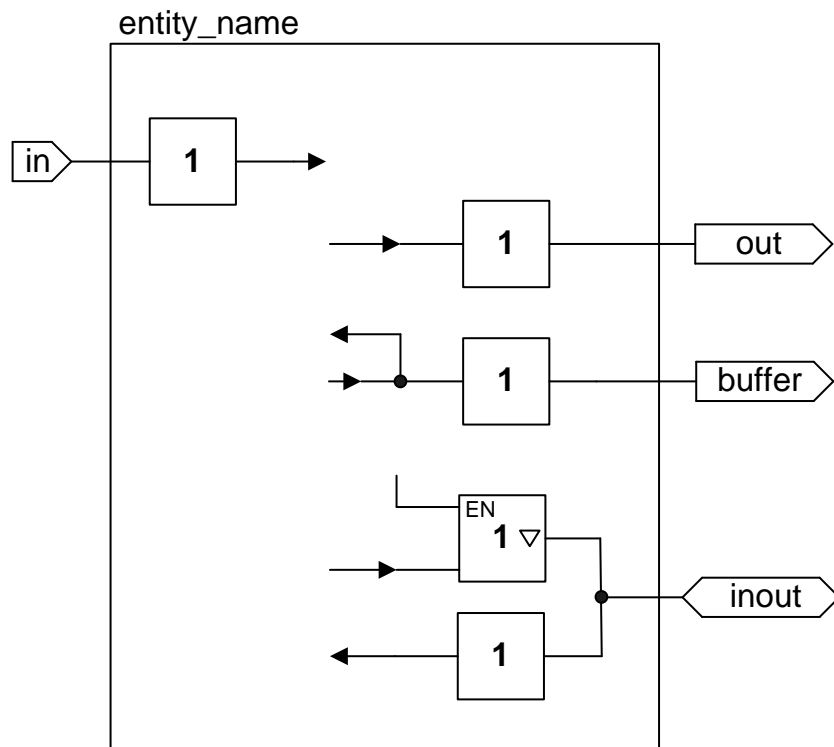
```
MY_BYTE: in bit_vector(7 downto 0); -- MSB hat Index 7
```

```
MY_BYTE: in bit_vector(0 to 7); -- MSB hat Index 0
```

## Weitere VHDL-Syntaxelemente:

- **Kommentare** beginnen an beliebiger Stelle einer Zeile mit zwei Minuszeichen „--“ und enden am Ende einer Zeile.
- Am Ende einer VHDL-Anweisungen steht ein **Semikolon** „;“.
- In Anweisungen: Schlüsselwörter (z.B. `with`, `select`, `when`) oder Beistrich „ , “.

# Port-Modi



port-Modus	Verwendung
<b>in</b>	Das Signal kann nur gelesen (rechte Seite einer Signalzuweisung) oder abgefragt werden. Die Signalquelle liegt extern.
<b>out</b>	Die Signalquelle liegt in der architecture (interne Quelle). Das Signal darf nur auf der linken Seite einer Signalzuweisung stehen.
<b>buffer</b>	Das Signal befindet sich auf der linken Seite einer Signalzuweisung (interne Quelle), es kann aber auch gelesen werden (rechte Seite der Signalzuweisung).
<b>inout</b>	Bidirektionales Signal: Die Quelle liegt zeitweise intern und zeitweise extern. Die Verwendung erfordert den speziellen Datentyp <code>std_logic</code> (vgl. Kap 9.6).

# Aufbau einer architecture

## Zwei Bestandteile:

- **Deklarationsteil:** *lokale Signale*, und Komponenten
- **Anweisungsteil:** begin-end-Rahmen mit VHDL-Anweisungen:
  - Nebenläufige Signalzuweisungen
  - Prozesse
  - Komponentenmodule aus einer Bibliothek

```
entity TEST is
port( A, B, C : in bit;      -- Eingangssignale
      X, Y : out bit);      -- Ausgangssignale
end TEST;

architecture ARCH1 of TEST is
signal TEMP: bit;           -- Deklaration eines lokalen Koppelsignals
begin
    TEMP <= A and B;         -- Zuweisung an das Koppelsignal
    Y <= TEMP and C;
    X <= TEMP;               -- Kopie als Ausgangssignal
end ARCH1;
```

# Nebenläufige Signalzuweisungen

- Verwende den Signalzuweisungsoperator `<=`
  - Als Signalwerte auf der rechten Seite kommen in Frage:
    - Bit-Konstanten '0' und '1',  
z.B. `Y <= '0'`;
    - `bit_vector`-Konstanten eines Busses  
z.B. `E <= "1010"`;
    - ein logischer Ausdruck, in dem Signale mit logischen Operatoren verknüpft werden; z.B. `Y <= A and B`;
- 
- Alle nebenläufigen Signalzuweisungen und Prozesse einer `architecture` werden parallel ausgeführt.
  - Jede nebenläufige Anweisung bzw. jeder Prozess repräsentiert einen Hardware-Funktionsblock.

# Logikoperatoren in VHDL

- Operatoren `not`, `and`, `or`, `nand`, `nor`, `xor`, `xnor`

Bei Bussignalen gelten diese bitweise.

- Sind deklariert für die automatisch deklarierten Datentypen:

- `type bit is ('0', '1');`
- `type boolean is (false, true);`

- Klammerung von Logikoperatoren:

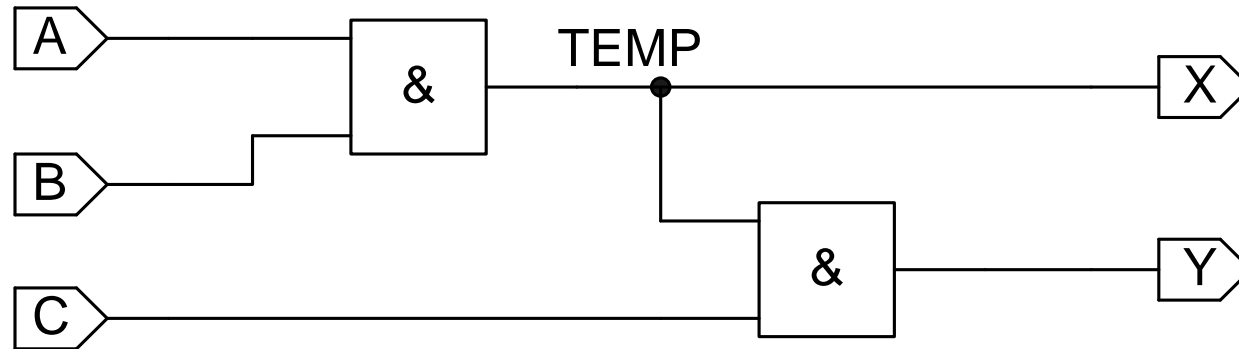
`Y <= not (A and B and C);` -- NAND3, ein NAND mit 3 Eingängen

`Y <= (A nand B) nand C;` -- ist erlaubt, aber kein NAND3

`Y <= A nand B nand C;` -- ist falsch; Syntaxfehler !



# Datenflussmodell mit Logikoperatoren



```
entity TEST is
port( A, B, C : in bit;      -- Eingangssignale
      X, Y : out bit);      -- Ausgangssignale
end TEST;
architecture ARCH1 of TEST is
signal TEMP: bit;           -- Deklaration eines lokalen Koppelsignals
begin
    TEMP <= A and B;         -- Zuweisung an das Koppelsignal
    Y <= TEMP and C;
    X <= TEMP;              -- Kopie als Ausgangssignal
end ARCH1;
```

# Nebenläufige Signalzuweisungen

---

- **Drei Arten von nebenläufigen Anweisungen:**
  1. Die **unbedingte**,
  2. die **selektive** und
  3. die **bedingte** Signalzuweisung.

# Wahrheitstabelle eines 2-zu-1-Multiplexers

- Es soll ein 2-zu-1-Multiplexer mit Low-aktivem Freigabeeingang entworfen werden.
- Die Wahrheitstabelle verwendet Don't-Care-Einträge auf der linken Seite

Minterme	$\bar{E}$	S	IB	IA	Y
$m_8, \dots, m_{15}$	1	X	X	X	0
$m_4, m_5$	0	1	0	X	0
$m_6, m_7$	0	1	1	X	1
$m_0, m_2$	0	0	X	0	0
$m_1, m_3$	0	0	X	1	1

**Beachte:**

**Die Zeilen in dieser  
Wahrheitstabelle  
sind nicht in  
„natürlicher Reihenfolge“ !**

# VHDL-Modell eines 2-zu-1-Multiplexers

Drei verschiedene Modelle des Multiplexers:

```
entity MUX2_1 is
  port( IA, IB      : in bit;  -- Dateneingänge
        S : in bit;          -- Selektionssignal
        nE : in bit;         -- Freigabe (Low aktiv)
        Y1, Y2, Y3 : out bit); -- Ausgangssignale
end MUX2_1;

architecture MUX of MUX2_1 is
begin
  Y1 <= (IA and not nE and not S) or
        (IB and not nE and S);

  with S select
    Y2 <= (IA and not nE) when '0',
          (IB and not nE) when '1';

  Y3 <= (IA and not nE) when S = '0' else
        (IB and not nE);

end MUX;
```

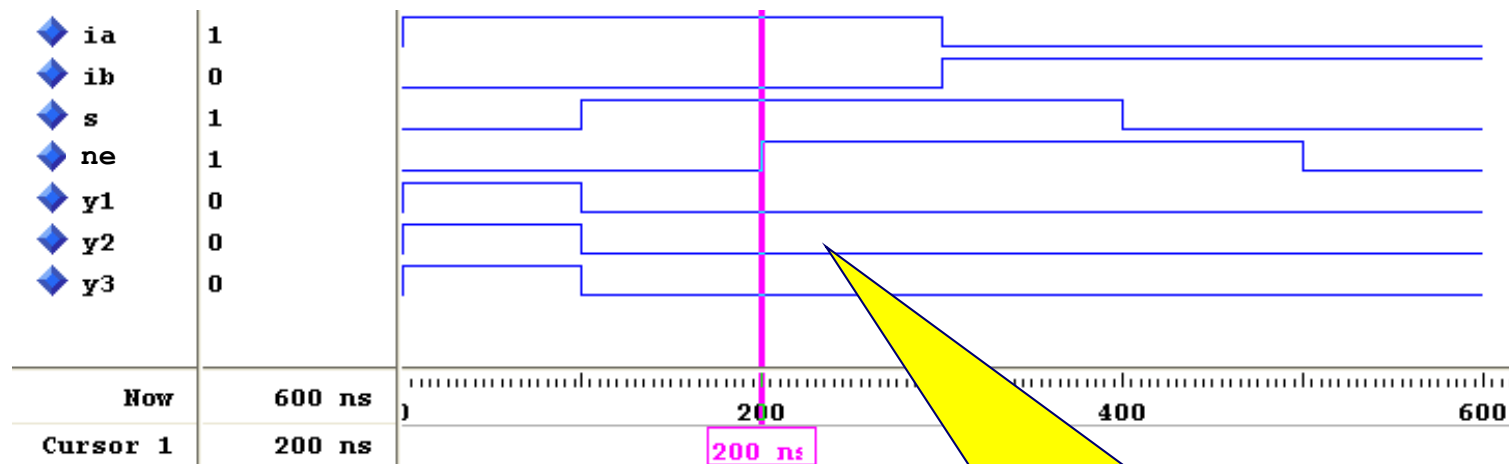
Unbedingte Signalzuweisung

Selective Signalzuweisung

Bedingte Signalzuweisung

# Simulationsergebnis für den Multiplexer

- IA, IB, S und nE sind extern vorgegebene Stimulussignale
- Y1, Y2 und Y3 sind die Ausgangssignale der drei Modellierungsvarianten



Das Zeitverhalten der  
drei Modelle  
y1, y2 und y3 ist gleich

# Simulations- und Syntheseergebnisse des MUX

## Ergebnis der Synthese:

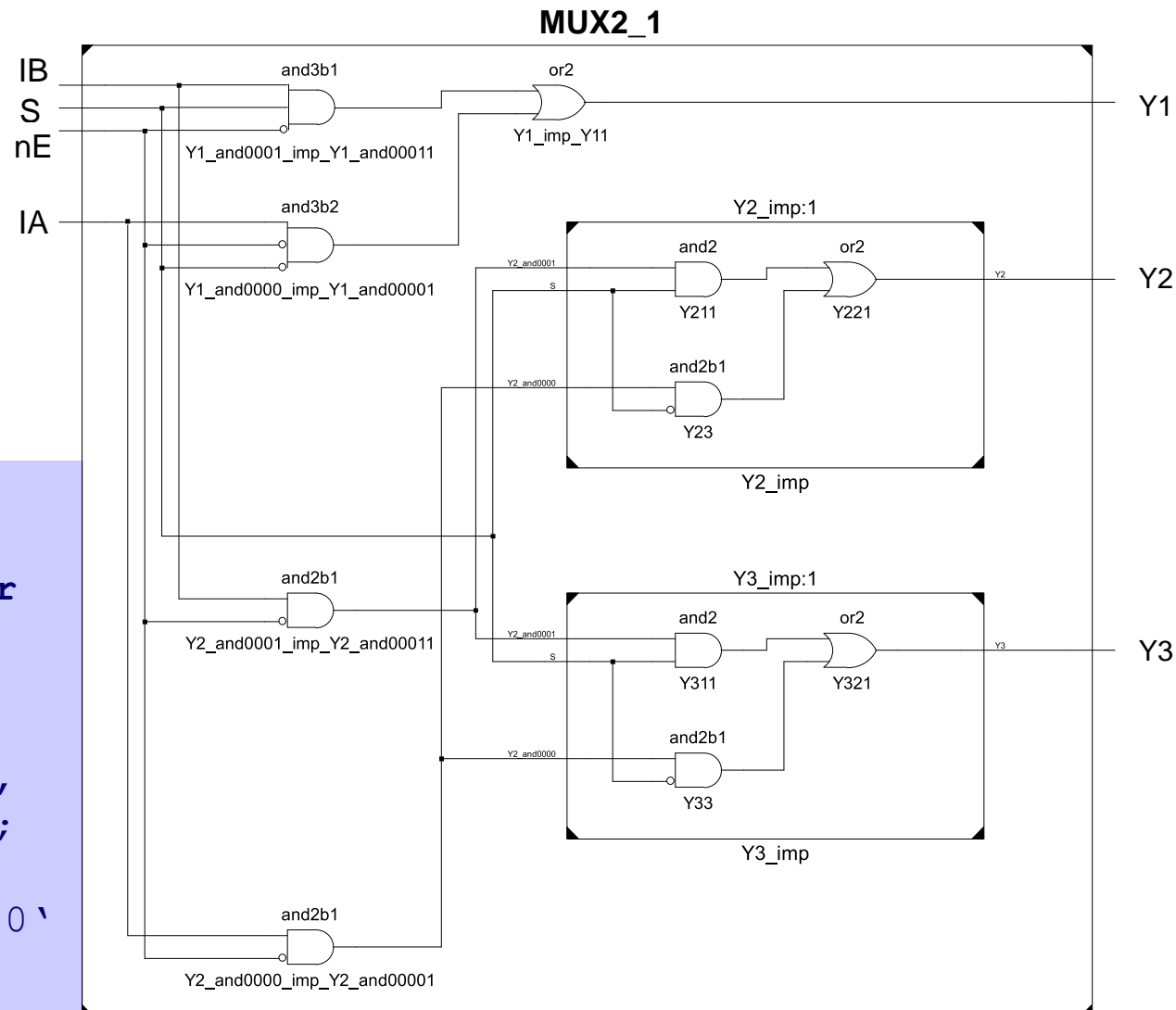
- Alle drei Ausgangssignale werden mit der gleichen Logikfunktion gebildet.
- Der **Schaltplan** unterscheidet sich: Selektive und bedingte Signalzuweisung verwenden eine „black-box“ Y2\_imp bzw. Y3\_imp, die intern einem Multiplexer entspricht.

```
architecture MUX of MUX2_1 is
begin
    Y1 <= (IA and not nE and not S) or
          (IB and not nE and S);

    with S select
        Y2 <= (IA and not nE) when '0',
              (IB and not nE) when '1';

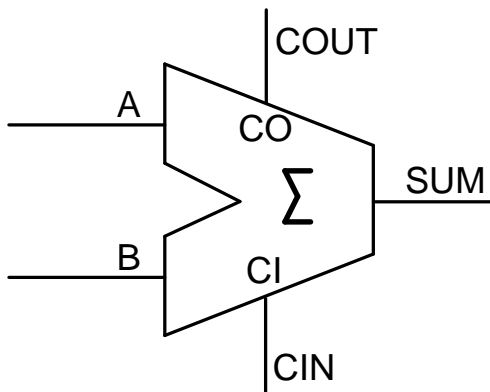
    Y3 <= (IA and not nE) when S = '0'
          else (IB and not nE);

end MUX;
```



# Modellierung von Wahrheitstabellen

- Wahrheitstabellen lassen sich mit selektiven Signalzuweisungen implementieren.  
Z.B.: ein Volladdierer



CIN	B	A	COUT	SUM
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

# VHDL-Modell eines Volladdierers

```
entity FULL_ADD is
    port( A, B, CIN : in bit      ;
          SUM, COUT : out bit );
end FULL_ADD;

architecture FA_1 of FULL_ADD is
    signal YINT1: bit_vector(2 downto 0);
    signal YINT2: bit_vector(1 downto 0);
begin
    YINT1 <= (CIN,B,A); -- Aggregat zur Buendelung von einzelnen Bits
    with YINT1 select
        YINT2    <=  "00" when "000",
                     "01" when "001",
                     "01" when "010",
                     "10" when "011",
                     "01" when "100",
                     "10" when "101",
                     "10" when "110",
                     "11" when "111";

    SUM    <= YINT2(0);
    COUT   <= YINT2(1);
end FA_1;
```

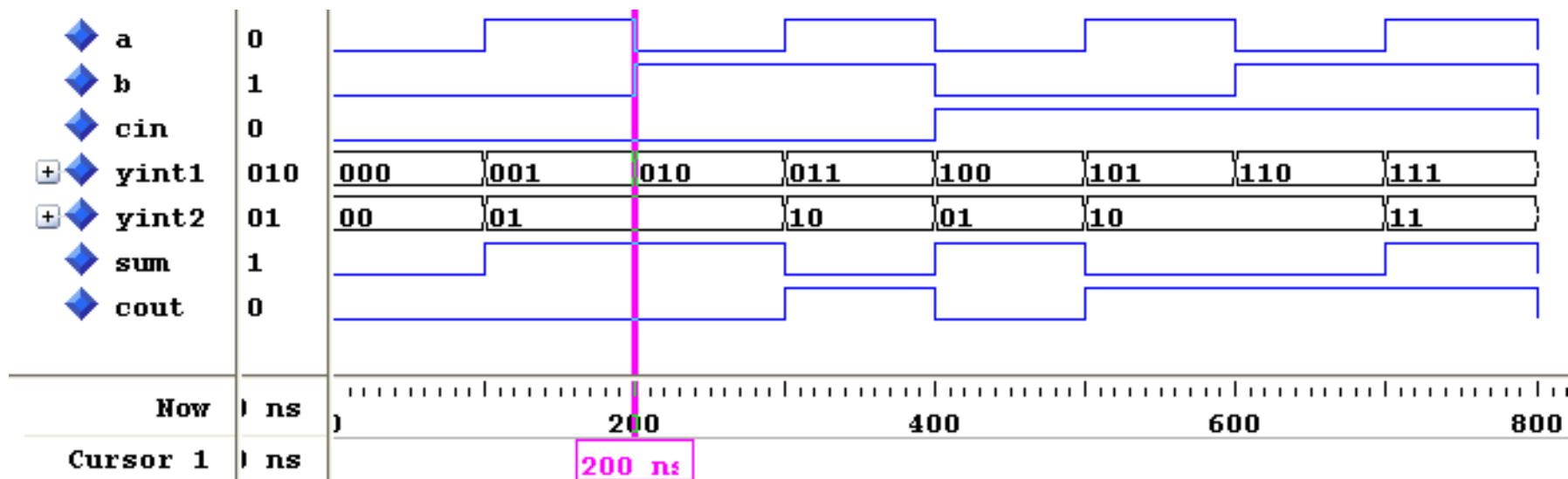
rechte Seite der  
Wahrheitstabelle

linke Seite der Wahrheitstabelle



# Simulationsergebnis des Volladdierers

Die Stimuli für die Eingangssignale A, B, CIN werden entweder im Simulator definiert, (force-Kommandos), aus einer Makro-Datei eingelesen (\*.do-Datei), oder durch eine **VHDL-Testbench** vorgegeben.

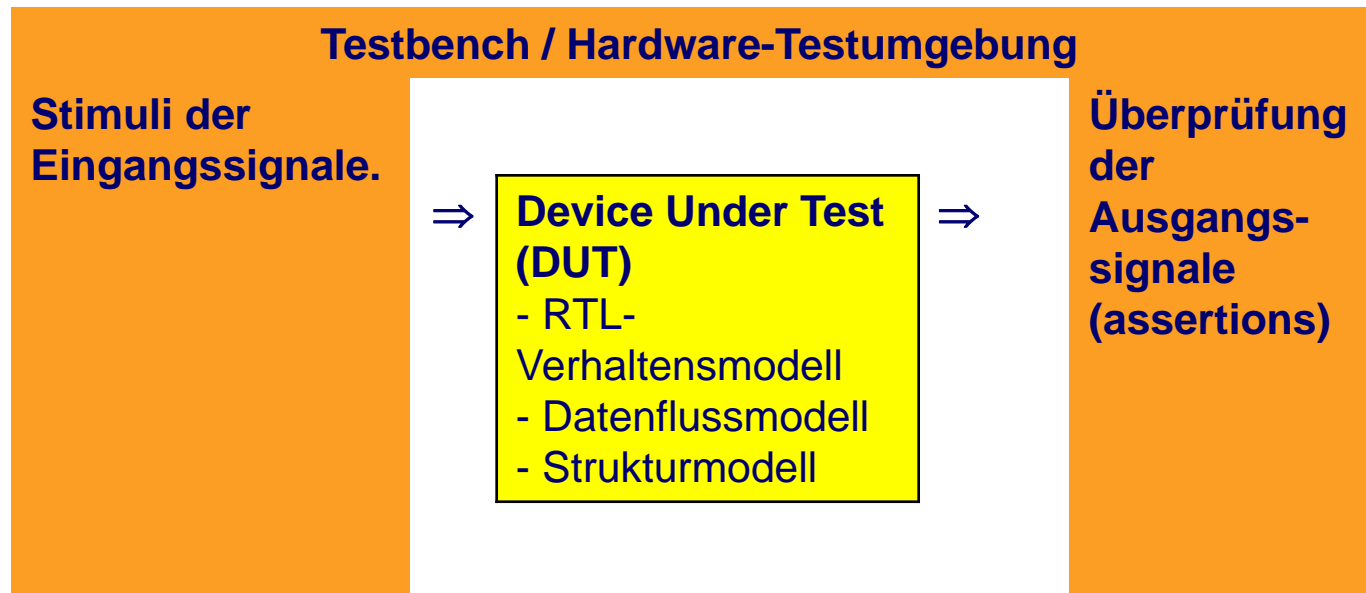


# Wahrheitstabelle eines Paritätsgenerators

- Paritätsbits dienen zur Fehlerüberprüfung
- Ein Paritätschecker überprüft, ob die empfangenen Datenbits zu dem empfangenen Paritätsbit passen.
  - Gerade Parität P\_E:  
Anzahl der Einsen ist gerade.
  - Ungerade Parität P\_O:  
Anzahl der Einsen ist ungerade.

C	B	A	P_O	P_E
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

# VHDL-Testbenches



Überprüfung der Ausgangssignale entweder als Wave-Form  
oder mit `assertion`-Anweisungen:

```
assert <Boole'scher Ausdruck> [report "<Textstring>"];
```

# Paritätsgenerator

```
-- Parity Generator
entity PARGEN is
    port( A, B, C : in bit      ;
          P_O : out bit );
end PARGEN;

architecture PG1 of PARGEN is
    signal YINT: bit_vector(2 downto 0);
    signal ODD: bit;
begin
    YINT <= (C,B,A); -- Aggregat zur Buendelung von einzelnen Bits
    ODD <= A xor B xor C xor P_O;
    with YINT select
        P_O    <=  '1' when "000",
                   '0' when "001",
                   '0' when "010",
                   '1' when "011",
                   '0' when "100",
                   '1' when "101",
                   '1' when "110",
                   '0' when "111";
end PG1;
```

# Testbench für den Paritätsgenerator (1)

```
entity PARGEN_TB is          -- Die Testbench besitzt keine Ports!
end PARGEN_TB;
architecture TESTBENCH of PARGEN_TB is
  component PARGEN is
  port(A, B, C : in bit;
        P_O : out bit);
  end component;

  signal A, B, C, P_O: bit;          -- lokale Signale

begin
  -- Stimulus Signale wie in der Wahrheitstabelle
  A <= '0', '1' after 100 ns, '0' after 200 ns,
        '1' after 300 ns, '0' after 400 ns, '1' after 500 ns,
        '0' after 600 ns, '1' after 700 ns;
  B <= '0', '1' after 200 ns, '0' after 400 ns,
        '1' after 600 ns;
  C <= '0', '1' after 400 ns;

  -- Device under test (DUT): Paritätsgenerator
  DUT: PARGEN port map(A, B, C, P_O);
```

Lokale Signale; „zufällig“ mit gleichen Namen wie ports.

Die entity/architecture PARGEN muss bereits erfolgreich übersetzt worden sein (library work)

# Testbench für den Paritätsgenerator (2)

```
-- Response Monitor
CHECK: process
begin
    wait for 50 ns;
    assert P_O = '1' report "Err: test# 0";
    wait for 100 ns;
    assert P_O = '0' report "Err: test# 1";
    wait for 100 ns;
    assert P_O = '0' report "Err: test# 2";
    wait for 100 ns;
    assert P_O = '1' report "Err: test# 3";
    wait for 100 ns;
    assert P_O = '0' report "Err: test# 4";
    wait for 100 ns;
    assert P_O = '1' report "Err: test# 5";
    wait for 100 ns;
    assert P_O = '1' report "Err: test# 6";
    wait for 100 ns;
    assert P_O = '1' report "Err: test# 7";-- falsche Erwartung
    wait for 100 ns;
end process CHECK;
end TESTBENCH;
```

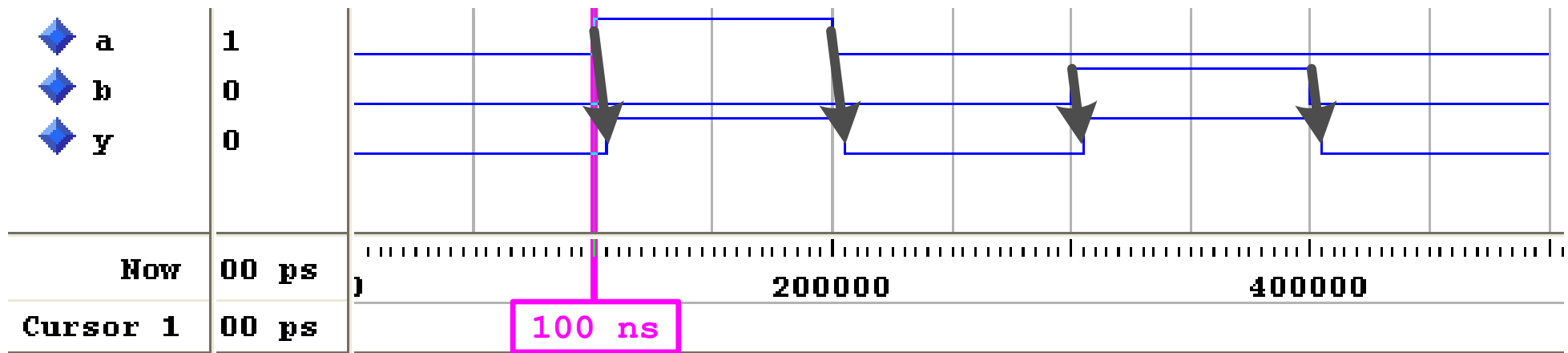
# \*\* Error: Err: test# 7  
# Time: 750 ns Iteration: 0  
Instance: /pargen\_tb

# **KOMBINATORISCHE LOGIK - GETAKTETE LOGIK (SEQUENTIELLE LOGIK)**

# Kombinatorische Logik

Charakteristisch für kombinatorische Logikbausteine ist die nahezu sofortige Reaktion des Ausgangssignals auf Änderungen der Eingangssignale.

Kombinatorisches Schaltverhalten eines ODER-Gatters:



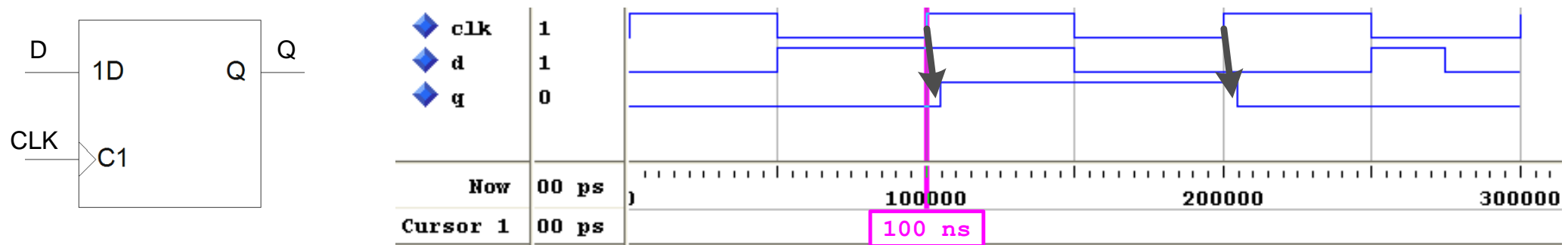


# Getaktete Logik

Ausgangssignale ändern sich nur nach einer Pegeländerung (Flanke) eines Taktsignals.

Steigende Flanke:  $0 \rightarrow 1$

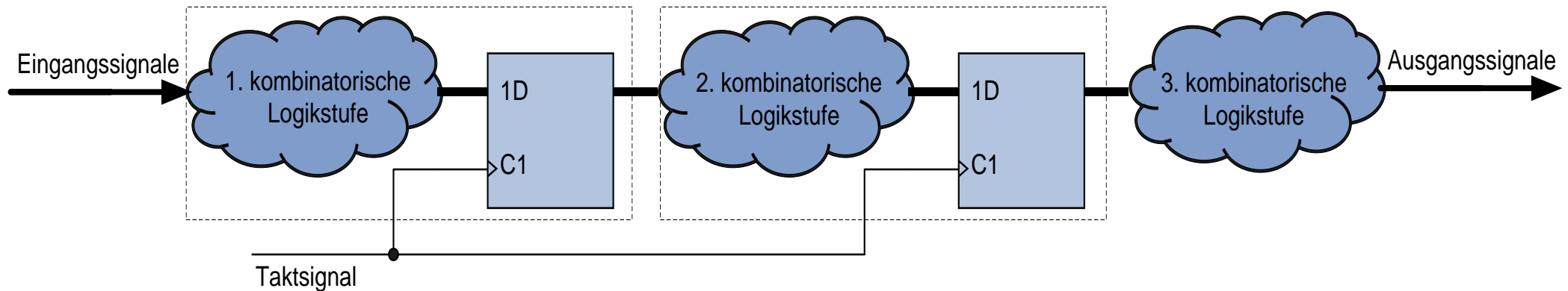
Fallende Flanke:  $1 \rightarrow 0$ .



- Schaltsymbol und Verhalten eines getakteten Logikbausteins (D-Flipflop). Die Pfeile zeigen die Wirkung des Taktsignals CLK auf den Ausgang Q.
- Änderungen des Dateneingangs allein haben keine unmittelbare Wirkung auf den Ausgang, sie können evtl. auch „überlesen“ werden.

# RTL: Register-Transfer Modelle

## Trennung von kombinatorischer Hardware und getakteter Hardware



- **Die RTL-Modellierung dieser Schaltung erfordert mindestens vier Prozesse:**
  - je einen Prozess für die drei kombinatorischen Schaltungsteile (Logikwolken).
  - einen Prozess für die beiden getakteten Schaltungsteile, die beide mit dem gleichen Takt arbeiten
- **Ursache dieser Restriktionen: VHDL-Synthesewerkzeuge**
- **Definition des RTL-Synthesestandards in der Norm IEEE-1076.3**

---

# VHDL Prozesse

# Eigenschaften von VHDL-Prozessen

**Nebenläufig:** Alle Prozesse innerhalb einer Architektur sind nebenläufig

**Aktivierung:** Aktivierung aufgrund von Signaländerungen in der *Sensitivityliste*, oder, repetitiv wenn Sensitivity leer ist.

**Sequentiell:** Diese Anweisungen werden *strikt nacheinander* ausgeführt.

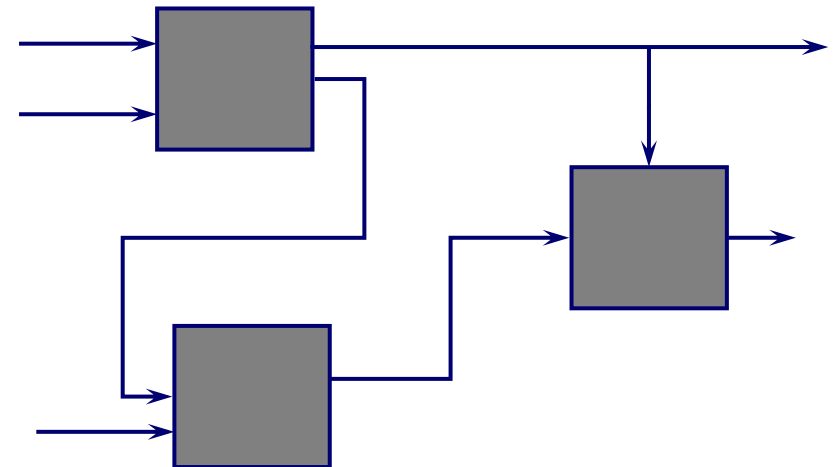
**Unbedingte Signalzuweisungen:** Nur unbedingte Signalzuweisungen und spezielle *sequenzielle Anweisungen* erlaubt (if, case, for, while).

**Bedingte Signalzuweisungen:** Bedingte und selektive nebenläufige Signalzuweisungen dürfen innerhalb von Prozessen nicht verwendet werden (select, when).

**Signale:** In Prozessen werden die Signalwerte festgelegt. Die Zuweisung der tatsächlichen Signalwerte eines Prozesses erfolgt *am Ende des Prozesses*.

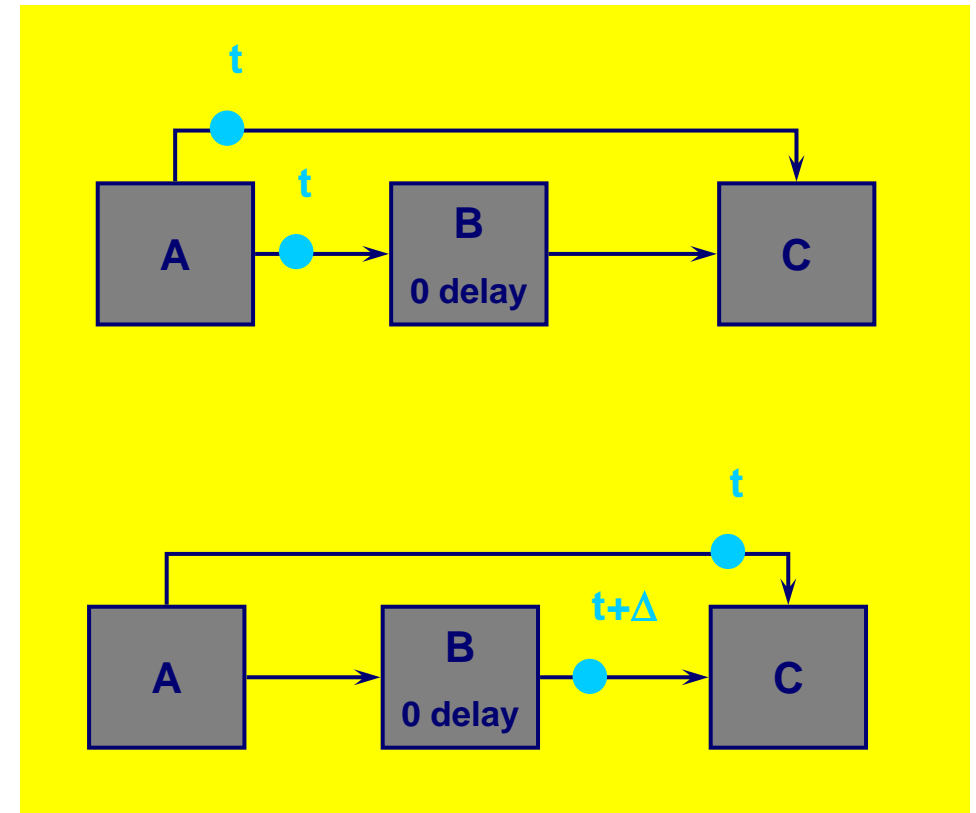
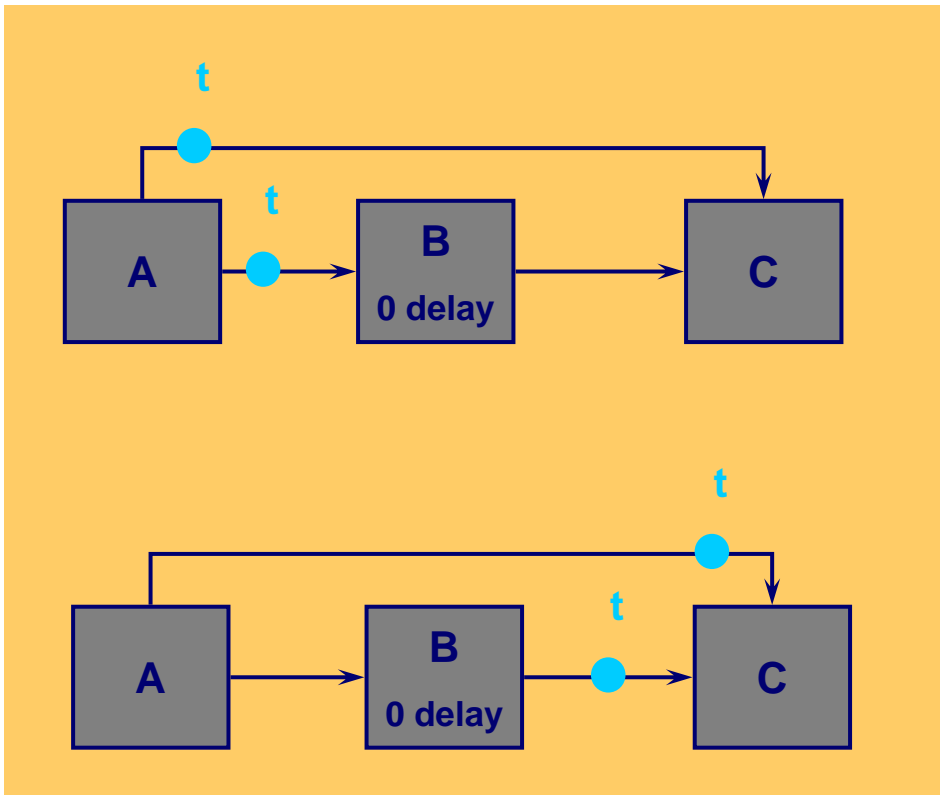
# Ereignisgesteuerte Modelle (Discrete Event)

- Ereignisgetriebene Dynamik
- Ereignis:
  - Input Stimuli
  - Intern erzeugte Ereignisse
- Ereignisse haben vollständig geordnete Zeit Stempel (time stamps)
- Komponenten mit beliebiger Verzögerung
- In VHDL sind Ereignisse und Zeit diskret

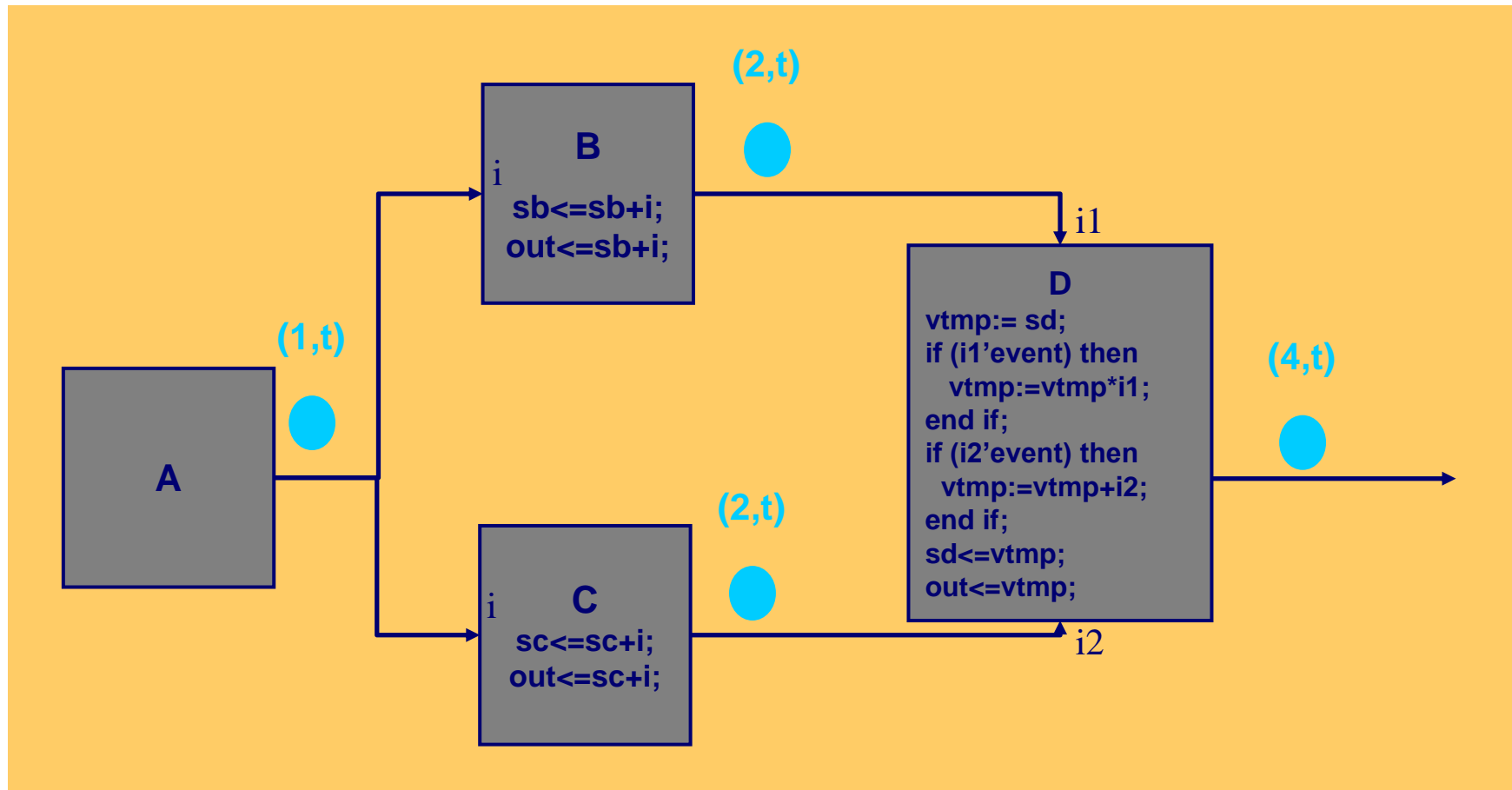


# Simultane Ereignisse - 0 -

## $\Delta$ delay Modell



# Simultane Ereignisse - 1 -



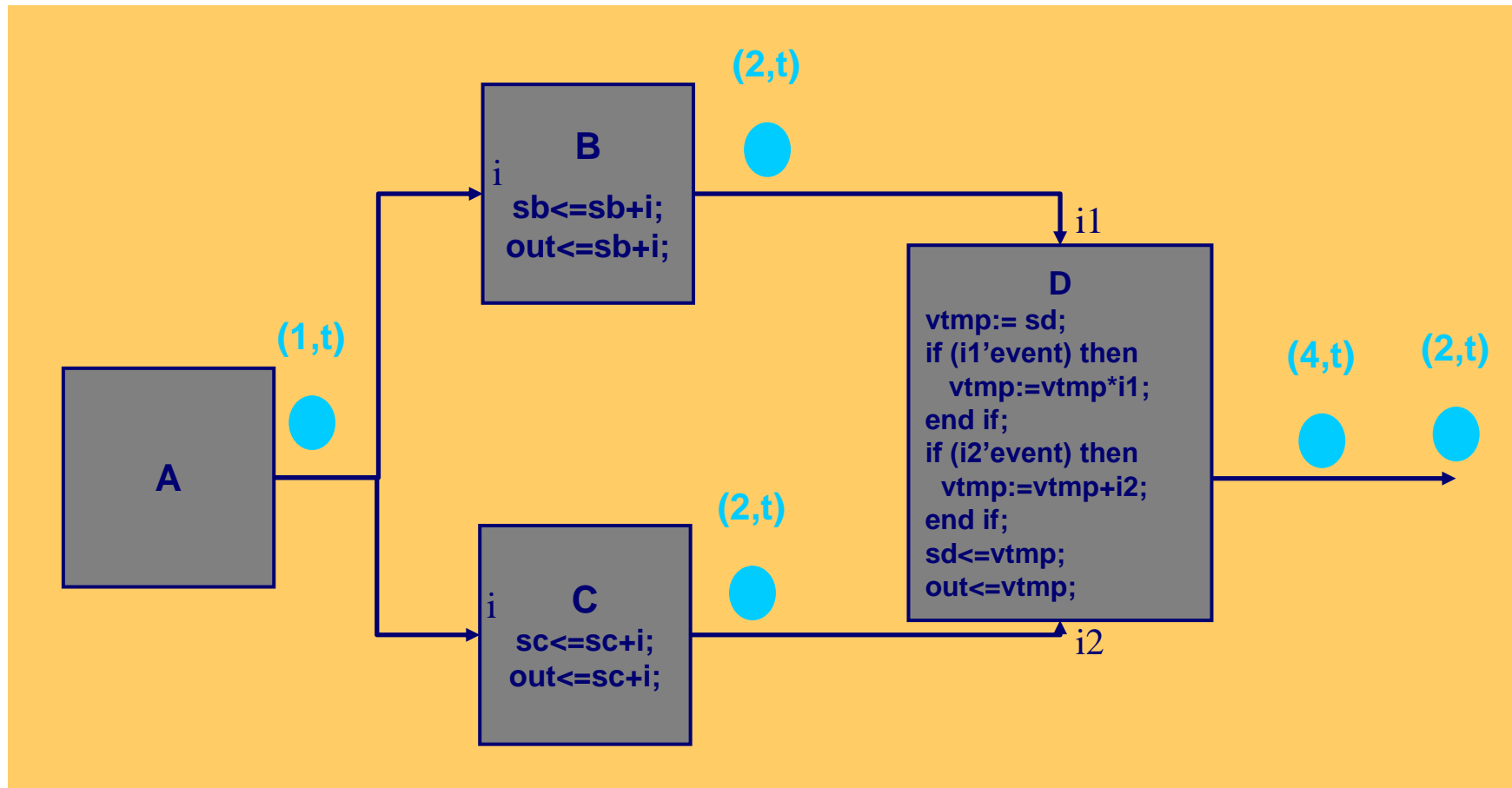
Initial state:  $sb \leq 1$ ;  $sc \leq 1$ ;  $sd \leq 1$ ;

Schedule:

**A** **B** **C** **D**

(4,t)

# Simultane Ereignisse - 2 -



Initial state:  $sb \leq 1$ ;  $sc \leq 1$ ;  $sd \leq 1$ ;

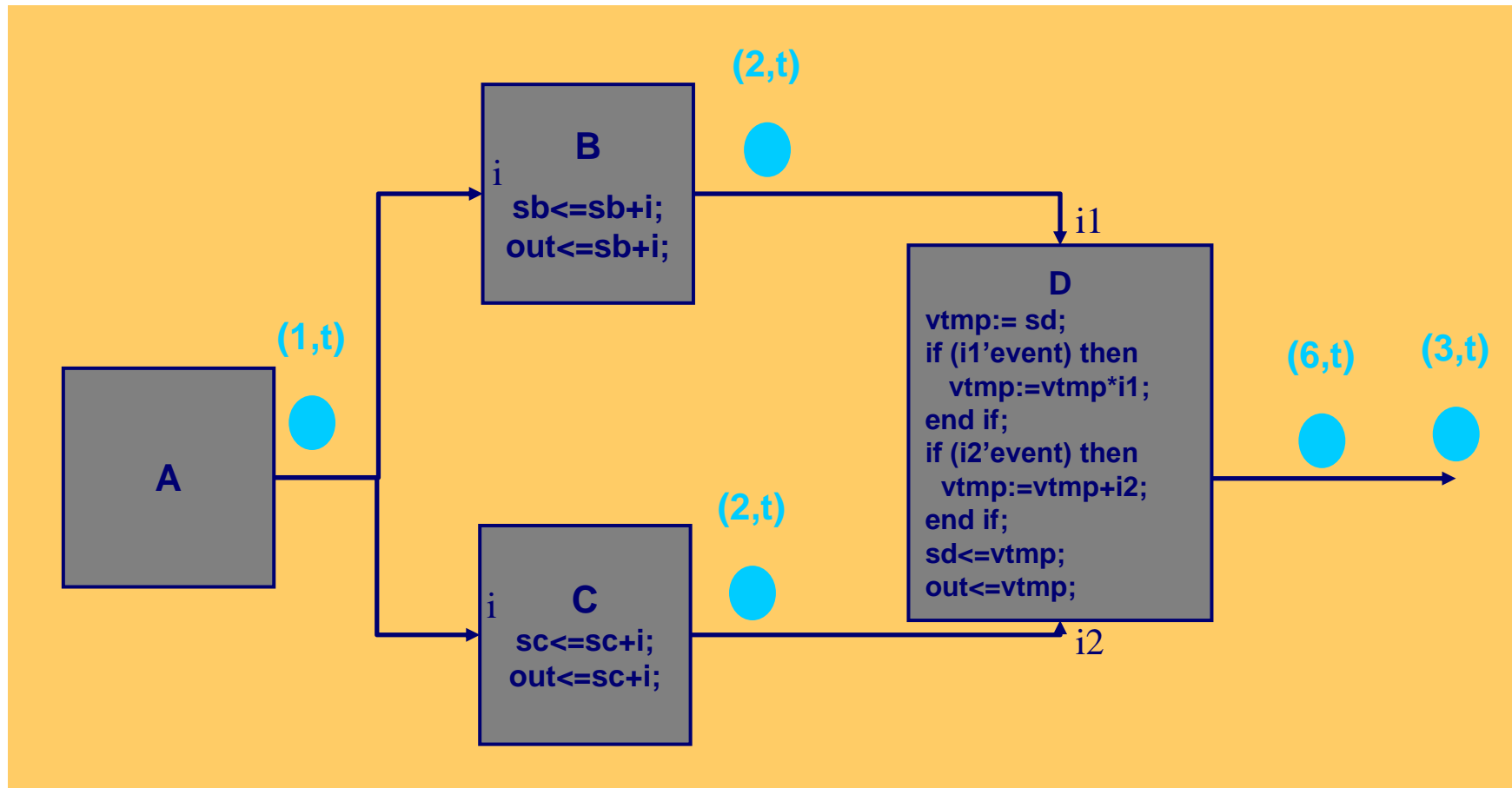
Schedule:

**A** **B** **C** **D** (4,t)

**A** **B** **D** **C** **D** (4,t) (2,t)



# Simultane Ereignisse - 3 -



Initial state:  $sb \leq 1$ ;  $sc \leq 1$ ;  $sd \leq 1$ ;

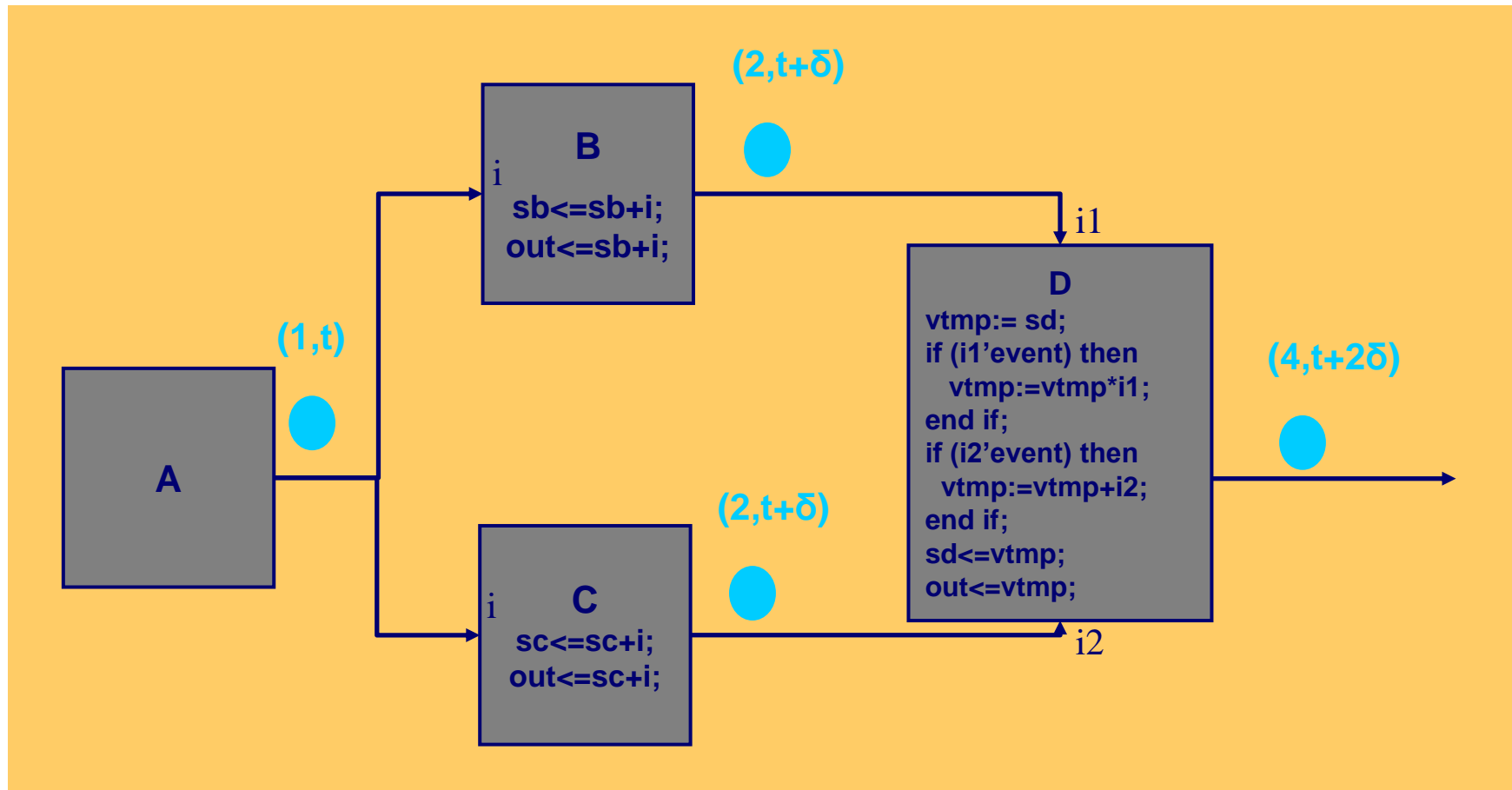
Schedule:

**A** **B** **C** **D** (4,t)

**A** **B** **D** **C** **D** (4,t) (2,t)

**A** **C** **D** **B** **D** (6,t) (3,t)

# Simultane Ereignisse mit Delta Delays

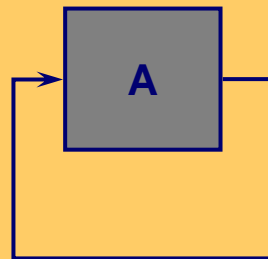
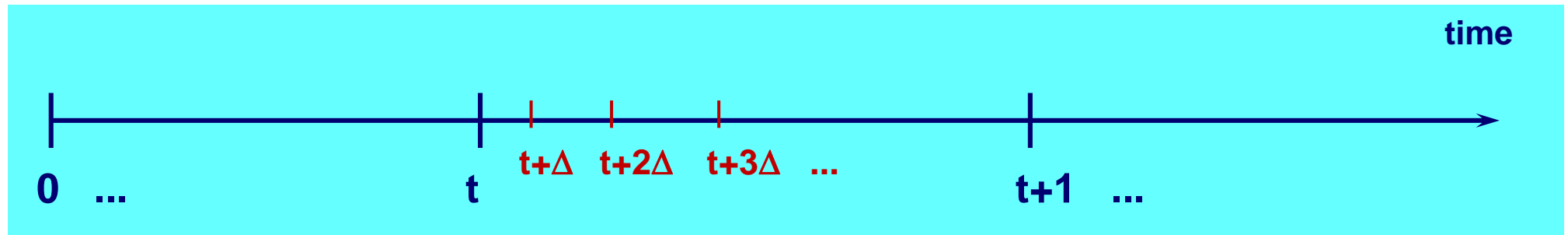


Initial state:  $sb \leq 1$ ;  $sc \leq 1$ ;  $sd \leq 1$ ;

Schedule:

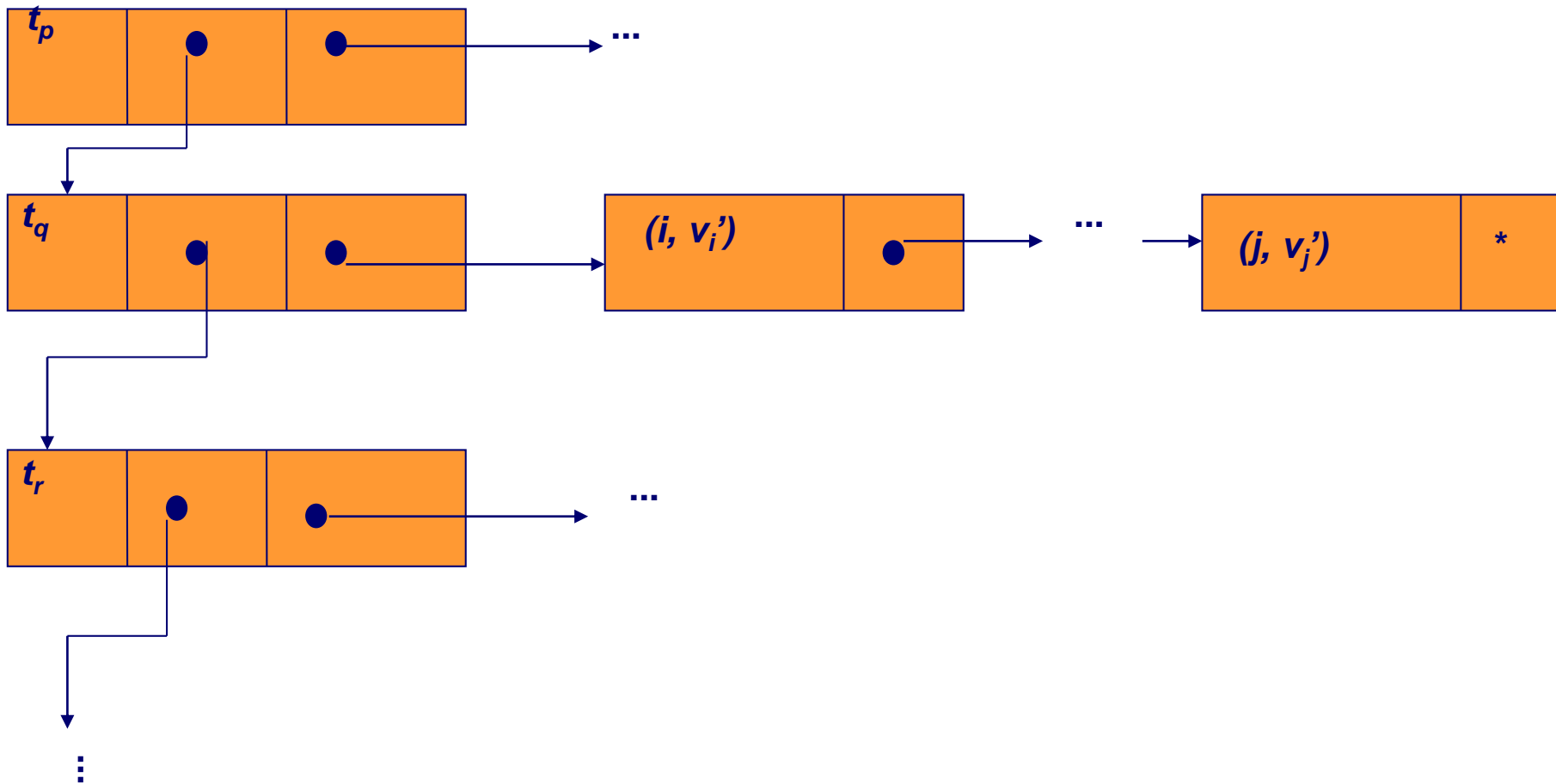


# Delta Time

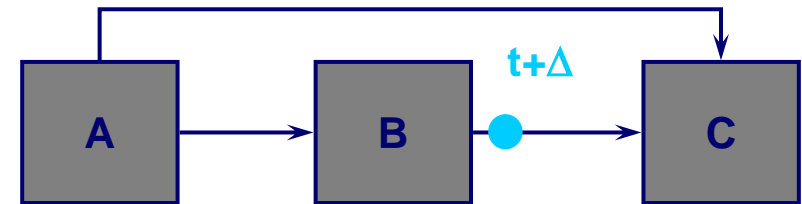
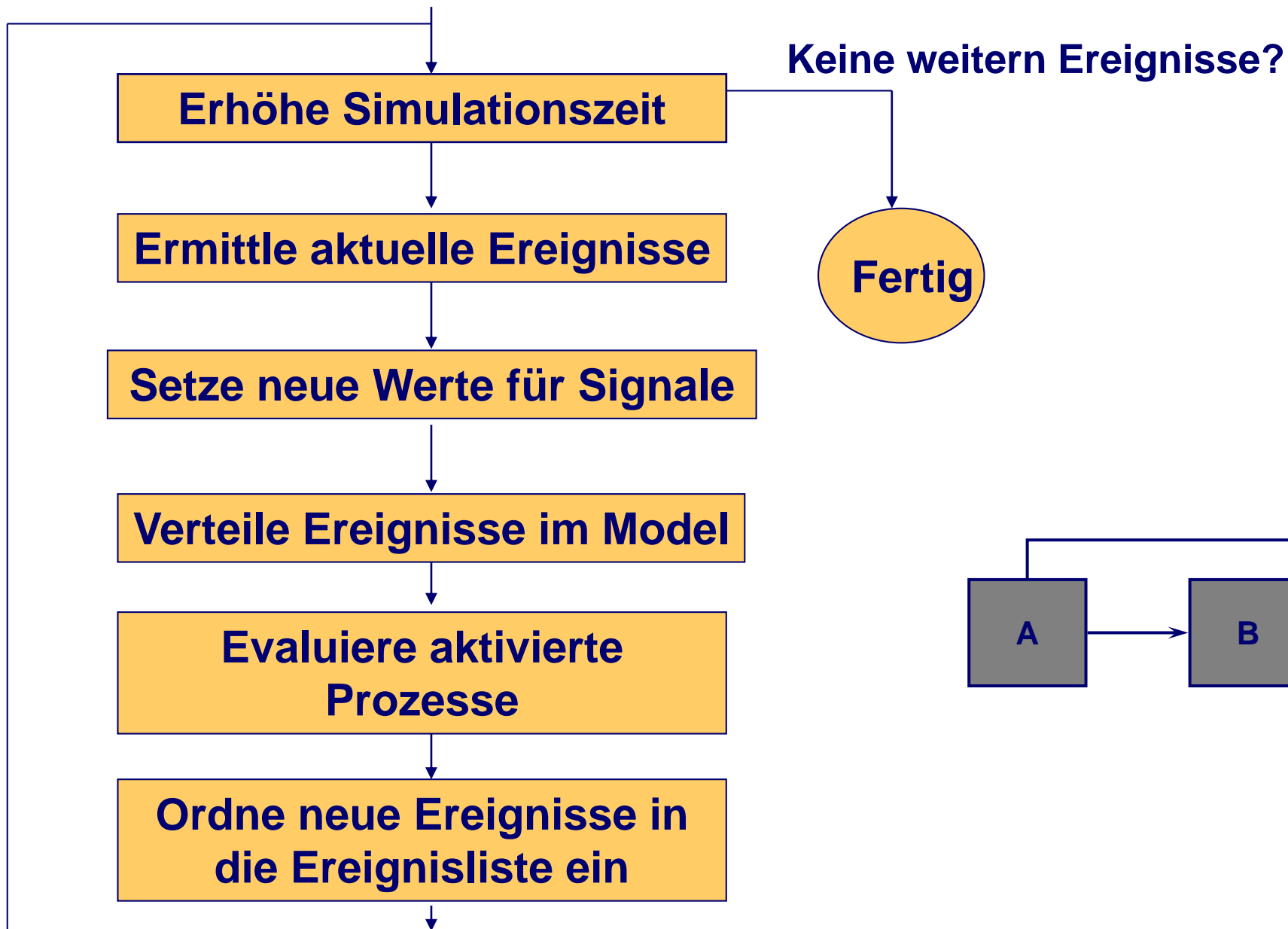


Das Modell erlaubt unendlich  
viele Ereignisse zwischen  
 $t$  und  $t+1$

# Event List

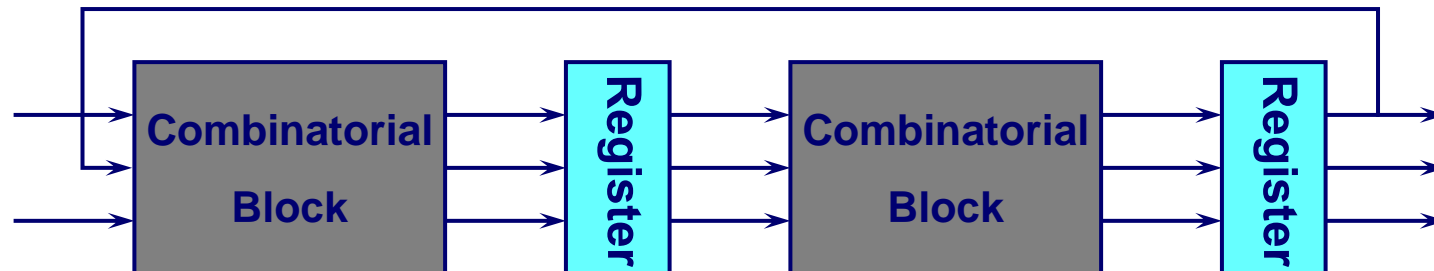


# Ereignisgetriebene Simulation

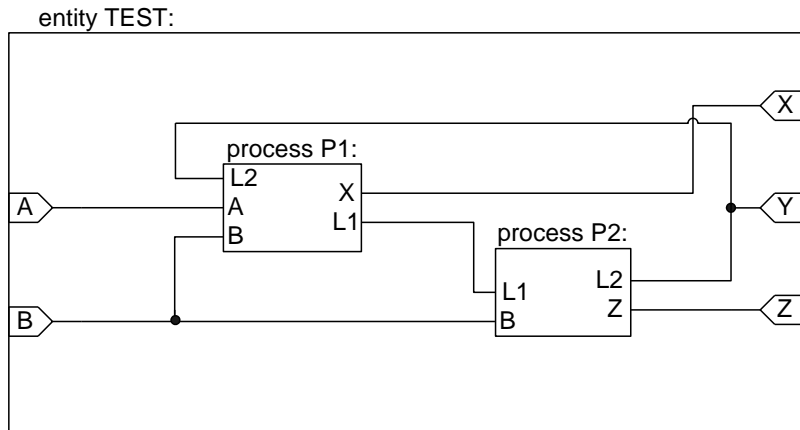


# Discrete Event Models

- **Zeitmodell ist nahe zur physikalischen Zeit**
  - Gut für die Simulation;
    - Aber: Globale Ereignisliste ist ein Engpass
  - Schwer zu synthetisieren;
  - Schwer zu verifizieren;
- DE Modelle werden entsprechend einem synchron getakteten Modell (Register Transfer) interpretiert



# Interaktion zwischen Prozessen



```
entity TEST is
port( A, B : in bit;           -- Eingangssig.
      X, Y, Z : out bit);      -- Ausgangssig.
end TEST;

architecture ARCH1 of TEST is
signal L1, L2: bit;           -- Dekl.lok. Sig.
begin
```

```
P1: process (L2, A, B)           -- Deklaration von P1 mit Sens.liste
begin
    ...
    X <= ...;                   -- Zuweisung an Ausgangssignal
    L1 <= ...;                  -- Zuweisung an lokales Signal
    ...
end process P1;

P2: process (L1, B)             -- Deklaration von P2 mit Sens.liste
begin
    ...
    L2 <= ...;                  -- Zuweisung an lokales Signal
    Z <= ...;                   -- Zuweisung an Ausgangssignal
    ...
end process P2;

Y <= L2;                        -- Kopie an Ausgangssignal
end ARCH;
```

**Bei VHDL-Prozessen, die kombinatorische Logik beschreiben, müssen sich alle Signale,**

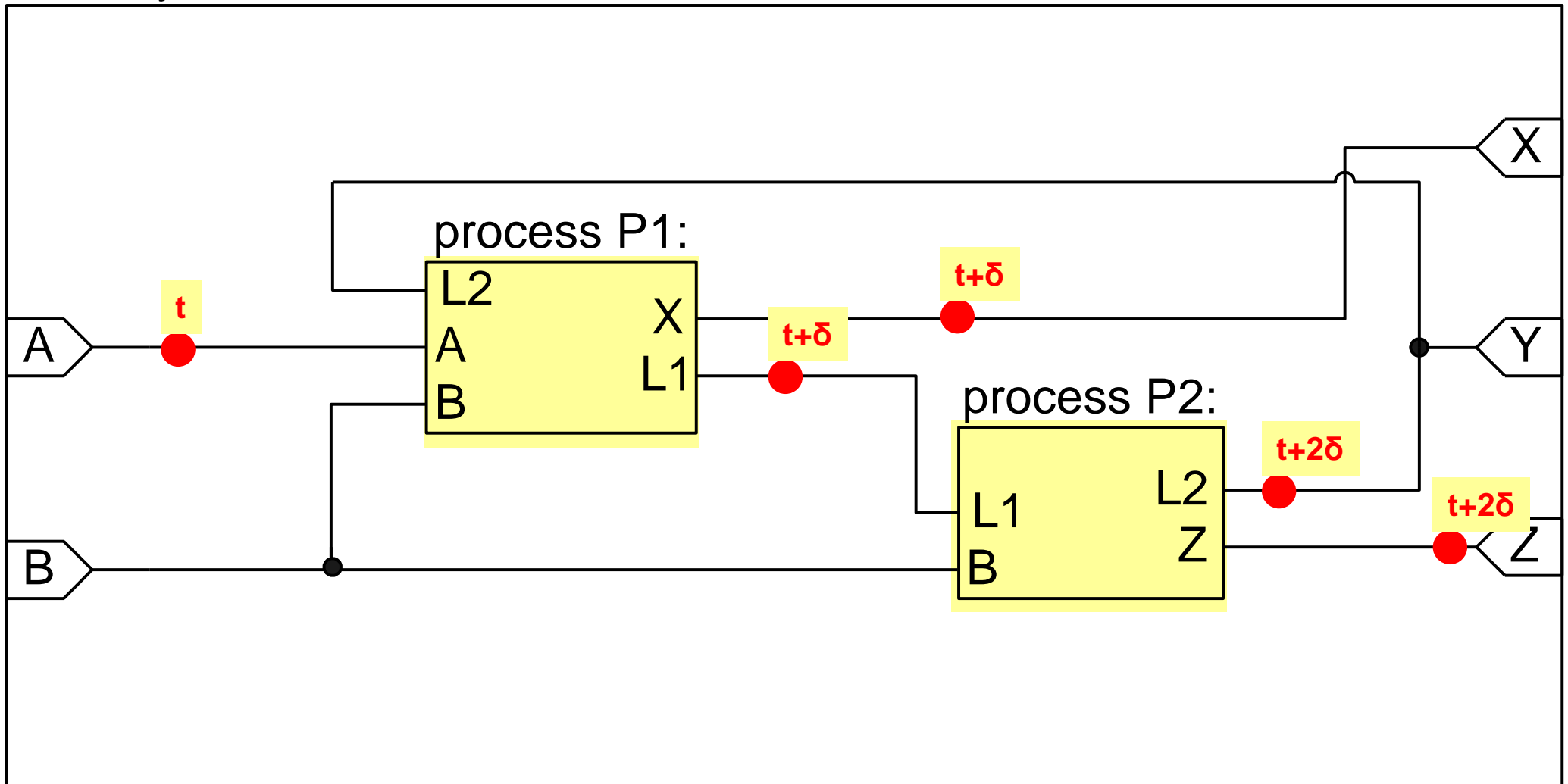
**die auf der rechten Seite einer Signalzuweisung stehen**

**oder**

die sich in  
Entscheidungsausdrücken  
sequenzieller  
Anweisungen befinden,  
in der **Sensitivityliste** des  
Prozesses befinden.

# Interaktion zwischen Prozessen

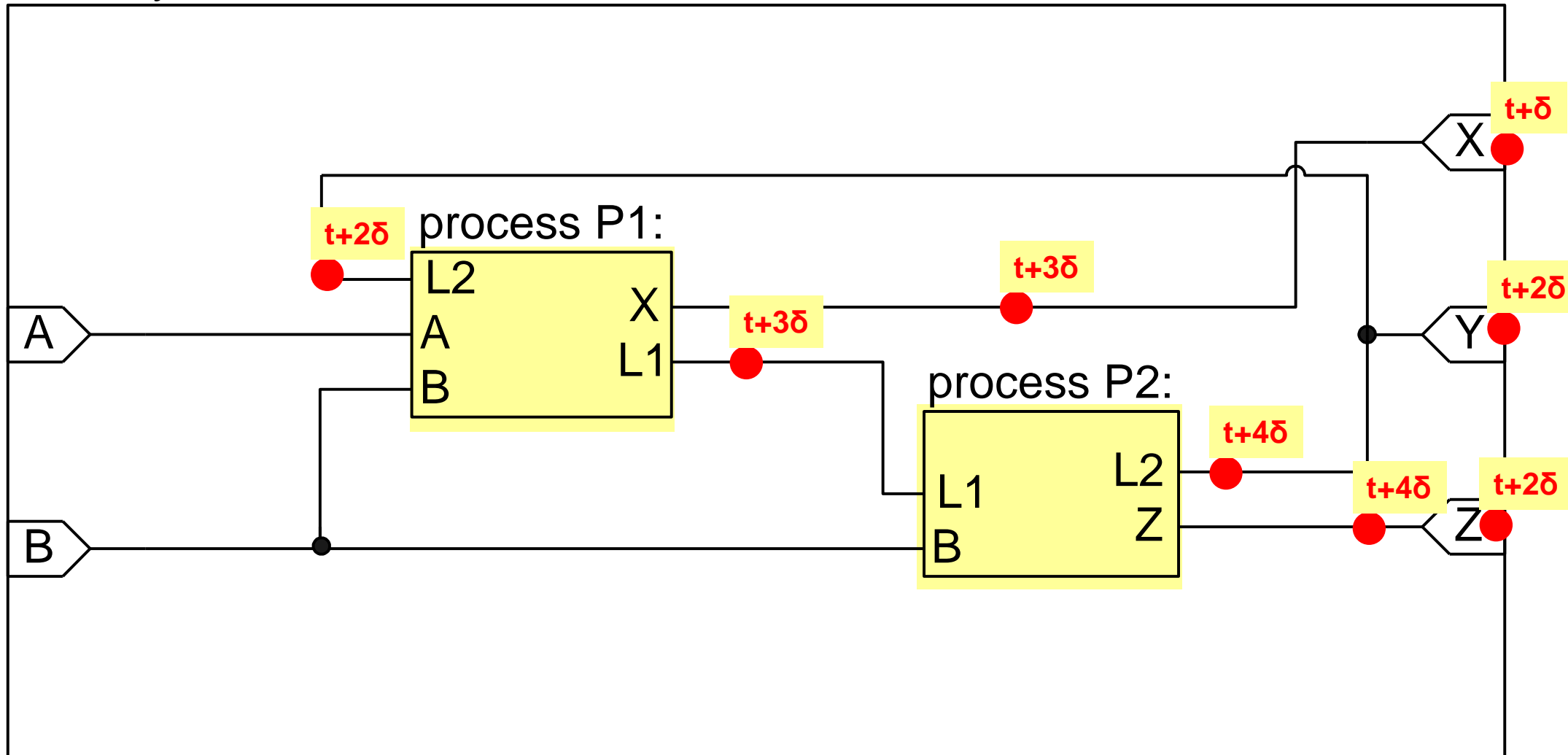
entity TEST:





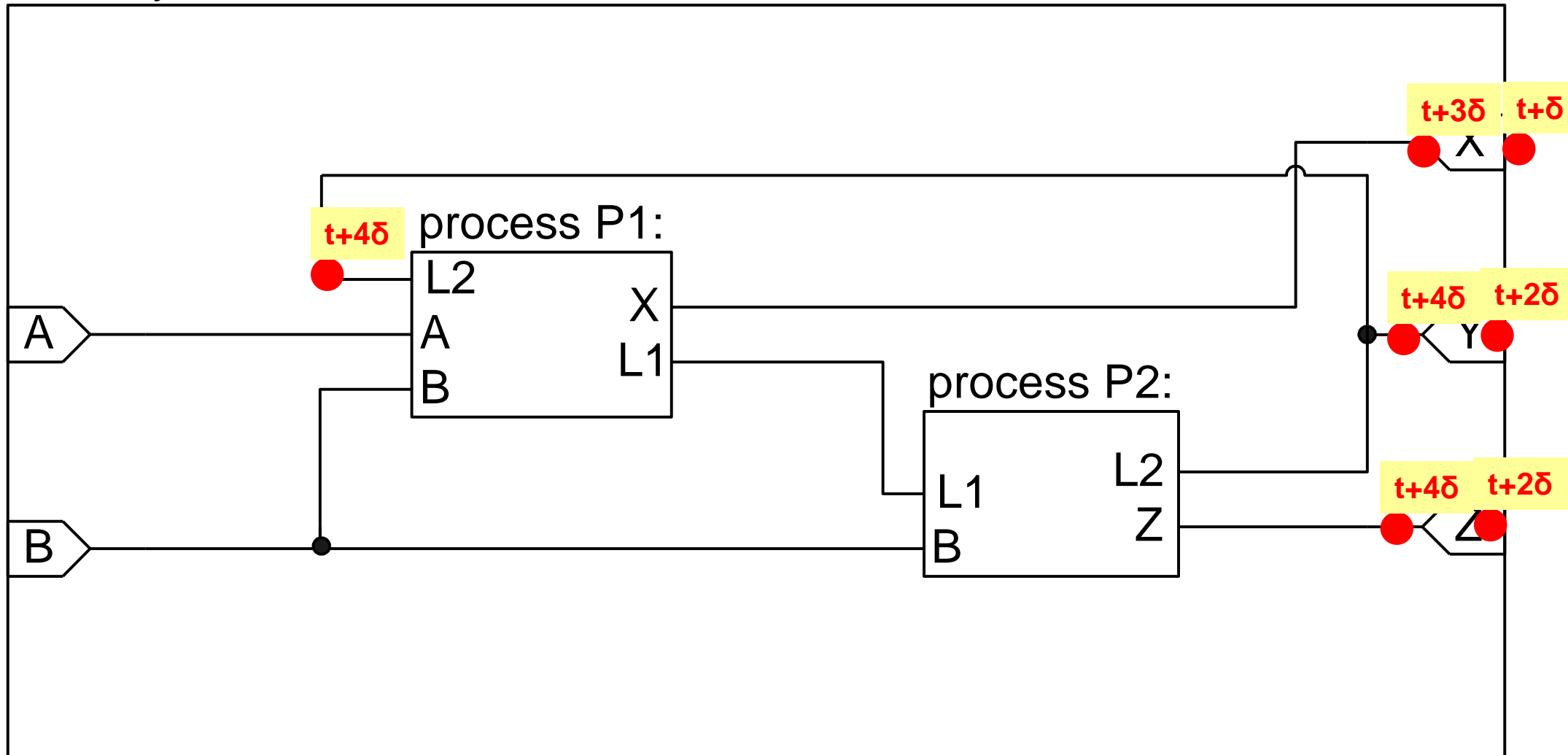
# Interaktion zwischen Prozessen

entity TEST:

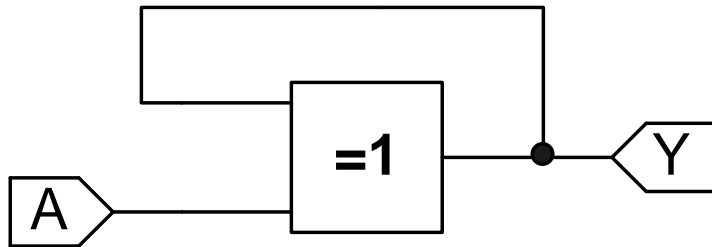


# Interaktion zwischen Prozessen

## entity TEST:



# Kombinatorische Schleifen



Nehme  $A=1$  und  $Y=0$  bei  $t=0$  sowie eine Gatterverzögerung von 5 ns an. Wie verhält sich der Ausgang für  $t>0$ ?

Kombinatorische Schleifen sind unbedingt zu vermeiden. Insbesondere sollten Ausgangssignale eines kombinatorischen Prozesses nicht in der Sensitivityliste des gleichen Prozesses stehen.

```
entity KOMB_SCHLEIFE is
port( A : in bit;           -- Eingangssignal
      Y : out bit);        -- Ausgangssignal
end KOMB_SCHLEIFE;
architecture ARCH1 of KOMB_SCHLEIFE is
signal TEMP: bit;          -- Deklaration eines lokalen Signals
begin
P1: process (A, TEMP)      -- Deklaration von P1 mit Sens.liste
begin
    TEMP <= A xor TEMP;    -- Zuweisung an lokales Signal
end process P1;
Y <= TEMP;                -- Zuweisung an Ausgangssignal
end ARCH1;
```

# VHDL-Signalverzögerungsmodelle (1)

**Delta-Delay:** Reaktion eines kombinatorischen Gatterausgangs ohne Delay:

```
Y0 <= A xor B; -- Delta Delay
```

**Inertial-Delay:** Modelliert Gatterträgheit: Kurze Impulse werden absorbiert:

```
Y1 <= A xor B after 2 ns; -- Short Inertial Model
```

```
Y2 <= A xor B after 8 ns; -- Long Inertial Model
```

**Transport-Delay:** Alle Eingangsimpulse werden verzögert am Ausgang abgebildet.

```
Y3 <= transport (A xor B) after 4 ns; -- Transport Model
```

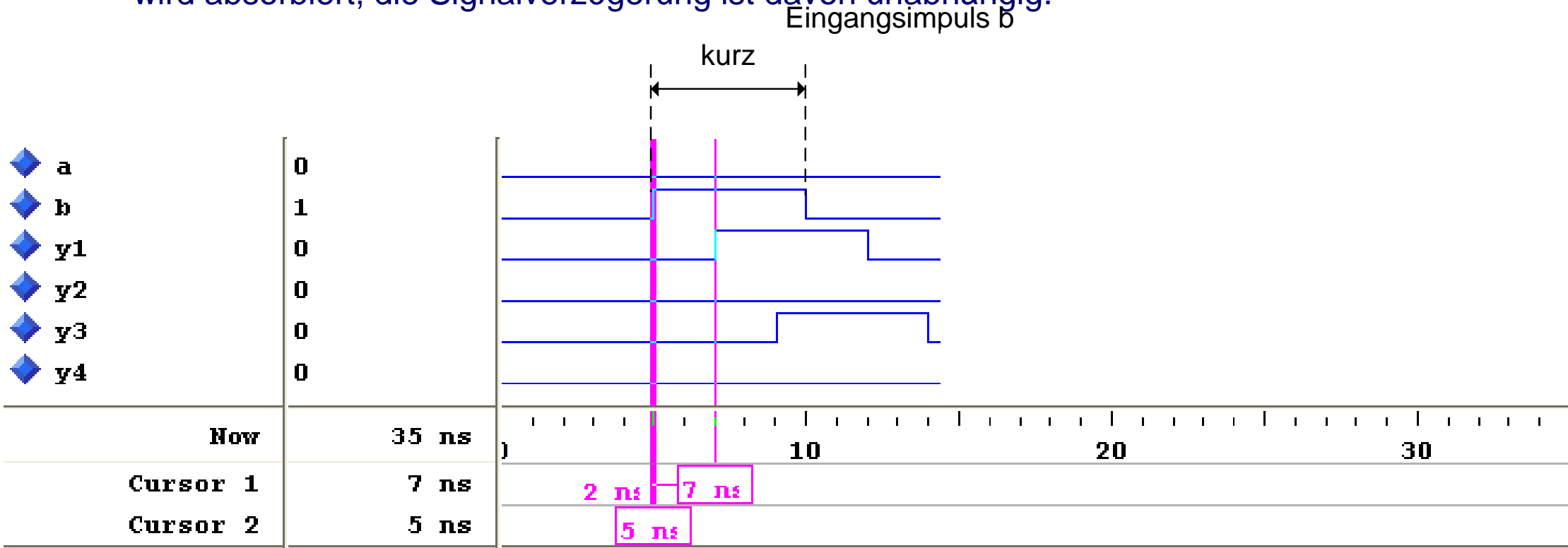
**Rejecting Inertial-Delay:** Getrennte Zeiten für die Mindestimpulsbreite (Schlüsselwort `reject`) und Ausgangssignalverzögerung (Schlüsselwort `inertial`).

```
Y4 <= reject 6 ns inertial (A xor B) after 7 ns;
```

# VHDL-Signalverzögerungsmodelle (2)

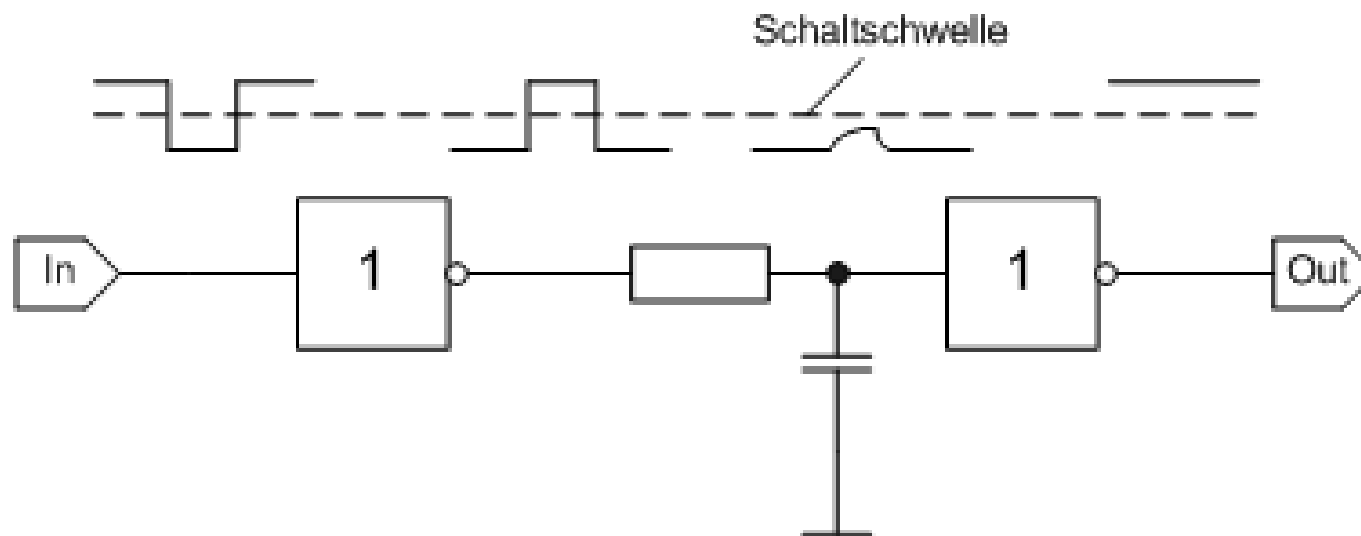
Darstellung verschiedener Verzögerungsmodelle am Beispiel eines XOR-Gatters:

- **y1**: Inertial-Modell mit kurzer Verzögerungszeit (2 ns)
- **y2**: Inertial-Modell mit langer Verzögerungszeit (8 ns),  
der kurze Eingangsimpuls der Dauer 5 ns wird absorbiert
- **y3**: Transport-Modell (4ns), beide Eingangsimpulse erscheinen verzögert am Ausgang
- **y4**: Rejecting-Inertial-Modell (6ns inertia, 7ns transport), der kurze Eingangsimpuls wird absorbiert, die Signalverzögerung ist davon unabhängig.



# Begründung für das Inertial-Delay Modell

- Der Leitungswiderstand der Signalverdrahtung stellt zusammen mit der Eingangskapazität des Gatters ein RC-Glied dar.
- Die Schaltschwelle des 2. Inverters wird verzögert erreicht.
- Falls der Eingangsimpuls zu kurz ist, so wird die Schaltschwelle evtl. gar nicht erreicht, das Ausgangssignal bleibt unverändert !



# Sequenzielle Anweisungen

Die Verhaltensmodellierung erfordert den Einsatz von Verzweigungs- und Schleifenanweisungen, so wie sie aus prozeduralen Programmiersprachen bekannt sind:

Innerhalb von Prozessen sind nur unbedingte Signalzuweisungen sowie **sequenzielle Anweisungen** erlaubt. Dazu zählen:

- Die **if**-Anweisung, die **case**-Anweisung, die **for**- und **while**-Schleifenanweisungen sowie die **wait**-Anweisung.
- Wertzuweisungen an dasselbe Signal können in Prozessen an verschiedenen Stellen erfolgen, es wird der Signalwert angenommen, der zuletzt zugewiesen wurde.

In Prozessen nicht erlaubt sind bedingte und selektive Signalzuweisungen!  
(select, when)

# Die case-Anweisung (1)

- **Modell eines 4-zu-1-Multiplexers:**

```
entity MUX4X1_2 is
    port( E : in bit_vector(3 downto 0);
          S : in bit_vector(1 downto 0);
          Y : out bit);
end MUX4X1_2;

architecture VERHALTEN of MUX4X1_2 is
begin
    MUXPROC: process(S, E)
    begin
        case S is
            when "00" => Y <= E(0);
            when "01" => Y <= E(1);
            when "10" => Y <= E(2);
            when others => Y <= E(3);
        end case;
    end process MUXPROC;
end VERHALTEN;
```

Abhängig von einem einzigen Bedingungsausdruck muss bei der **case-Anweisung** für alle möglichen Werte, die dieser Bedingungsausdruck annehmen kann, hinter dem Schlüsselwort

**when <Fallausdruck> =>**

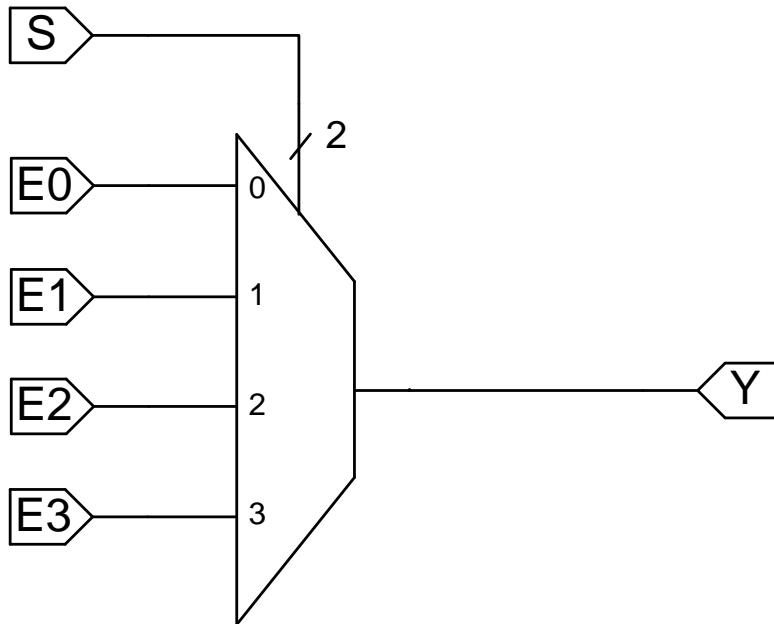
angegeben werden, welchen Wert ein Signal annehmen soll.

Prinzipiell sind hinter dem Schlüsselwort **when <Fallausdruck> =>** weitere sequenzielle Signalzuweisungen erlaubt (vgl. die Zustandsautomatenmodelle).



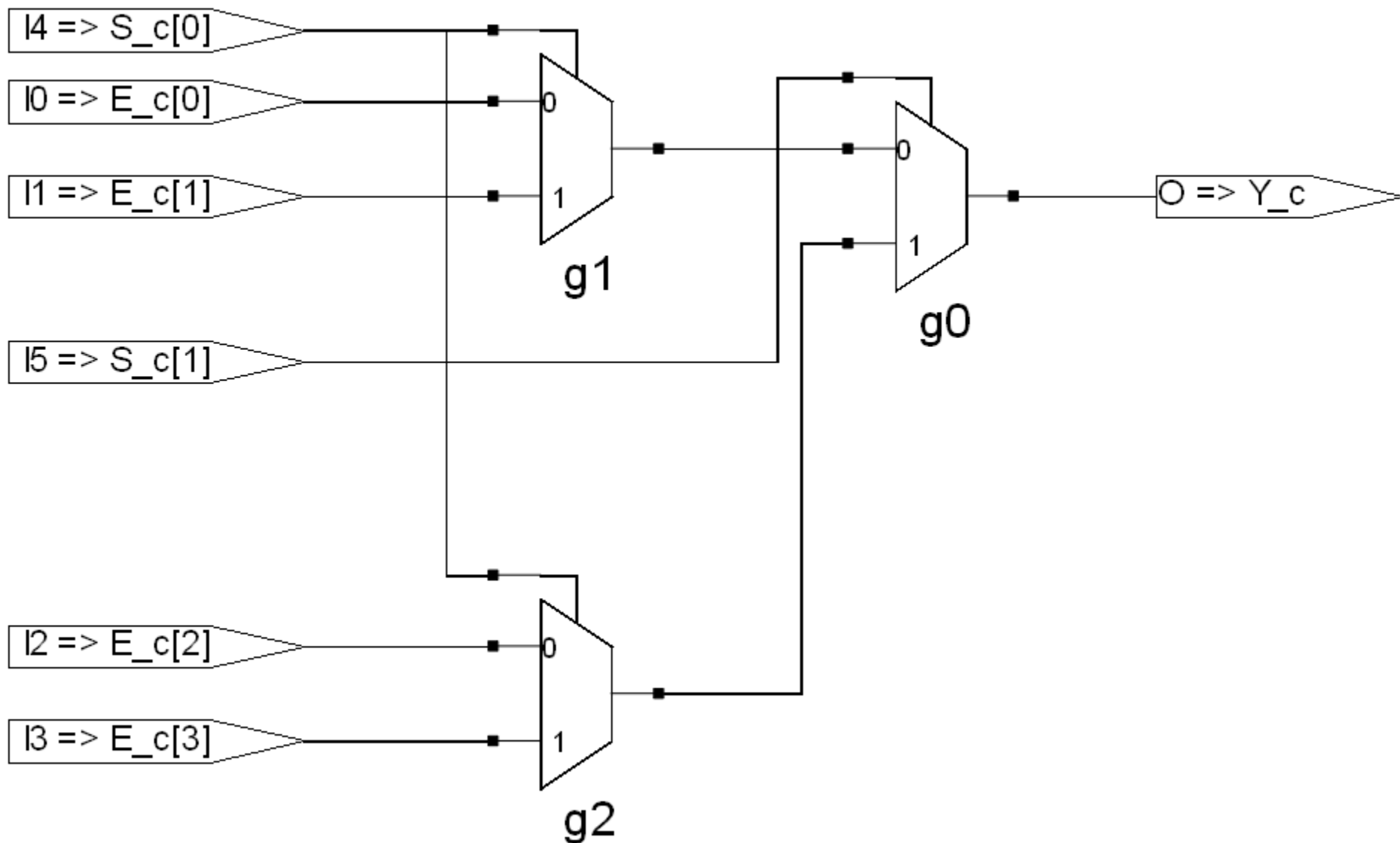
# Die case-Anweisung (2)

- **Syntheseeergebnis der case-Anweisung: 4-zu-1 Multiplexer**



Eine VHDL case-Anweisung wird zu einem Multiplexer synthetisiert. Daher muss die case-Anweisung vollständig spezifiziert sein (ggf. durch ein `when others =>`

# Synthese mit vollständiger case Anweisung



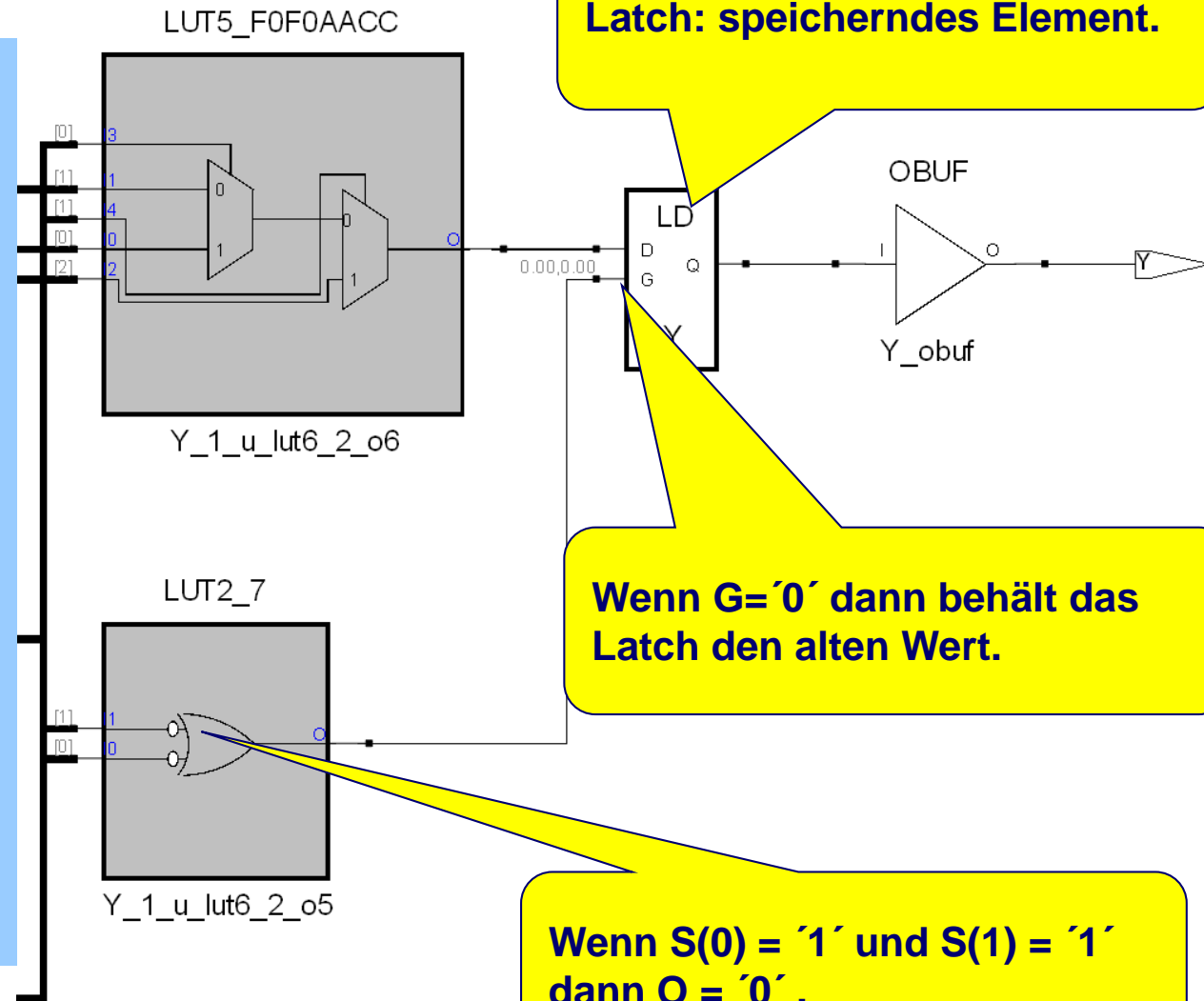
# Synthese mit unvollständiger case Anweisung

```

entity MUX4X1_2 is
    port( E : in bit_vector(3 downto 0);
          S : in bit_vector(1 downto 0);
          Y : out bit);
end MUX4X1_2;

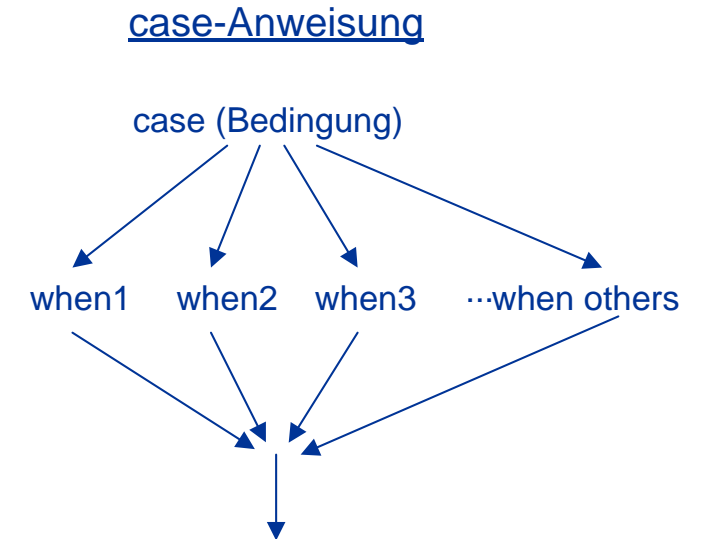
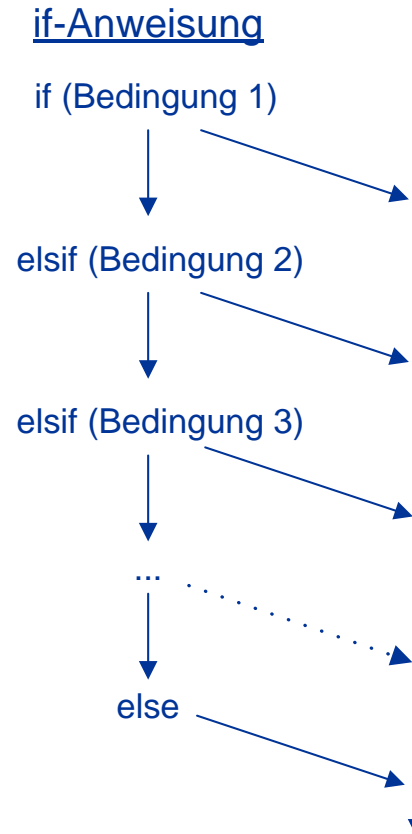
architecture VERHALTEN of MUX4X1_2 is
begin
    MUXPROC: process (S, E)
    begin
        case S is
            when "00" => Y <= E(0);
            when "01" => Y <= E(1);
            when "10" => Y <= E(2);
            -- when others => Y <= E(3);
        end case;
    end process MUXPROC;
end VERHALTEN;

```



# Vergleich von case- und if-Anweisungen

- Bei der case-Anweisung wird ein Ausdruck mit verschiedenen Wertmöglichkeiten geprüft. → Multiplexer / Demultiplexer-Struktur.
- Bei der if/elsif-Anweisung können mehrere voneinander unabhängige Bedingungen nacheinander überprüft werden. → Prioritätsencoderstruktur.



# Prioritätsencoder mit if-Anweisung (1)

```
entity PRIORITAETS_ENCODER is
    port( A, B : in bit; X : in bit; S : in bit_vector(1 downto 0);
          Y : out bit);
end PRIORITAETS_ENCODER;
architecture VERHALTEN of PRIORITAETS_ENCODER
begin
    P1: process(S, A, B, X)
    begin
        if S(0) = '1' then
            Y <= A and B;
        elsif S(1) = '1' then
            Y <= A or B;
        elsif X = '1' then
            Y <= A xor B;
        else
            Y <= '0';
        end if;
    end process P1;
end VERHALTEN;
```

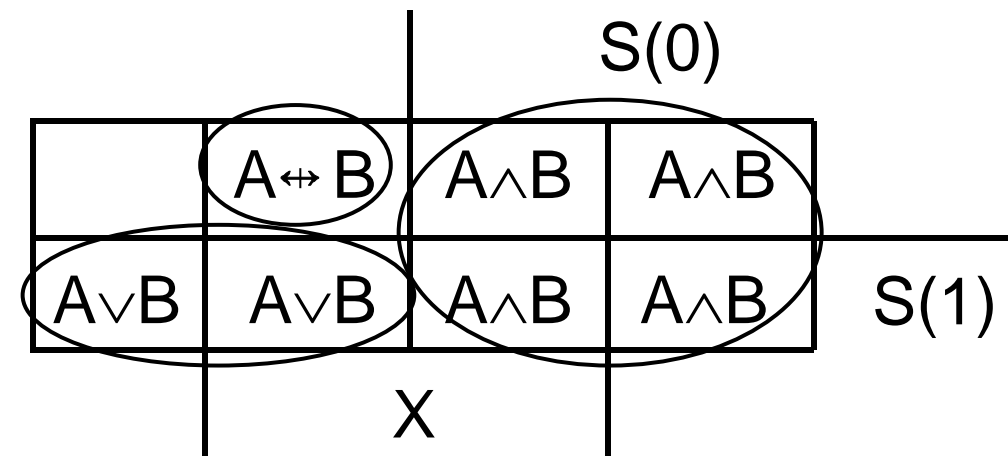
S(0) hat höchste Priorität.

Wie sieht das  
Syntheseresultat  
aus?

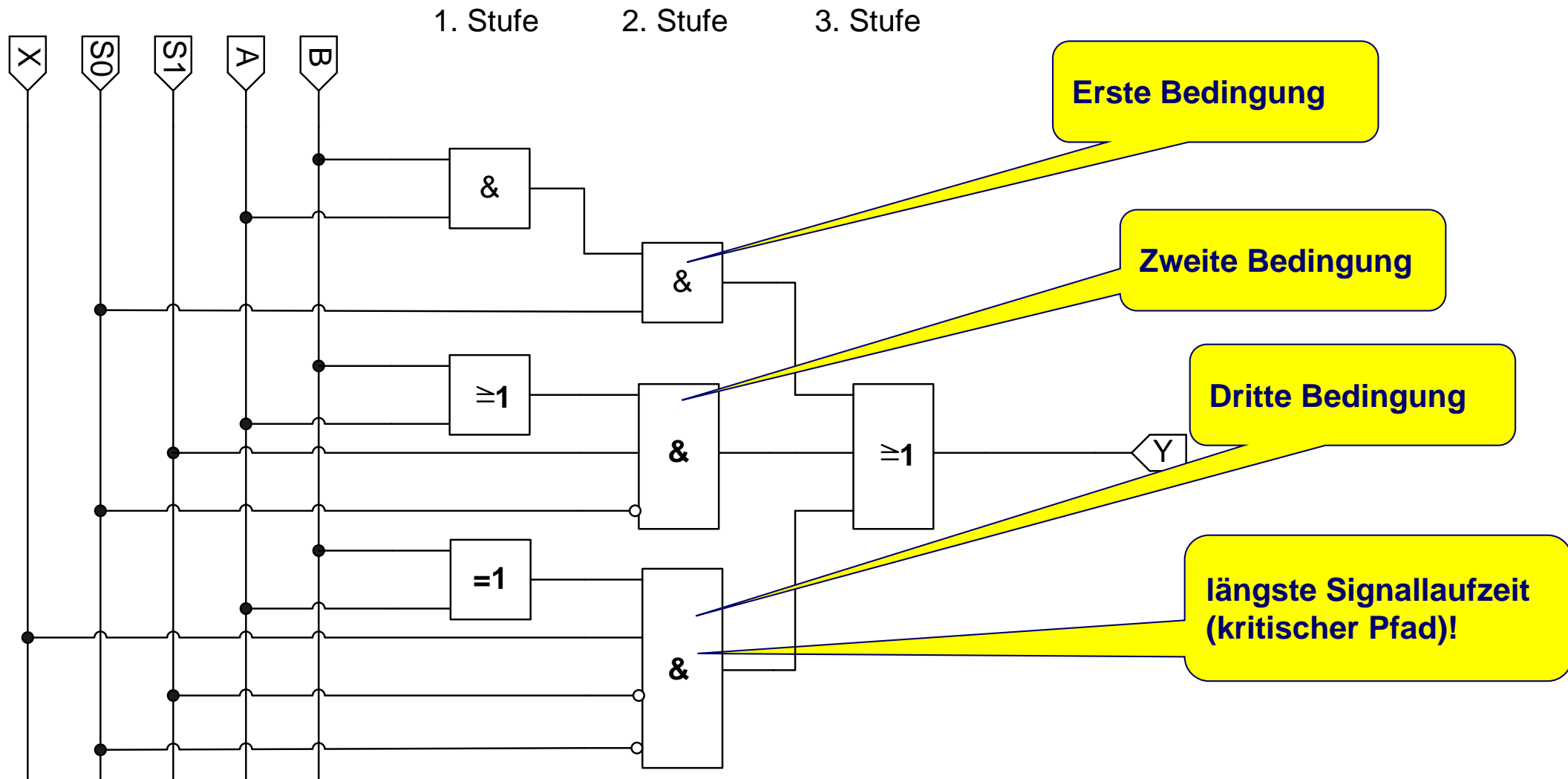
# Prioritätsencoder mit if-Anweisung (2)

- Interpretation der Schaltung als Schaltnetz mit Steuersignalen:

X	S(1)	S(0)	Y
0	0	0	0
0	0	1	$A \wedge B$
0	1	0	$A \vee B$
0	1	1	$A \wedge B$
1	0	0	$A \leftrightarrow B$
1	0	1	$A \wedge B$
1	1	0	$A \vee B$
1	1	1	$A \wedge B$



## Prioritätsencoder mit if-Anweisung (3)



# Modellierung von Signalflanken

- Verwendung der `if`-Anweisung im Zusammenhang mit dem Signalattribut `'event'`.
- Ein `else`-Zweig ist hier nicht erforderlich, da explizit ein speicherndes Verhalten gewünscht ist.
- Bsp. D-Flipflop mit steigender Flanke:

```
entity DFF is
    port( CLK, D : in bit;
          Q : out bit);
end DFF;
architecture VERHALTEN of DFF is
begin
    P1: process (CLK)
    begin
        if CLK='1' and CLK'event then -- ansteigende Signalflanke
            Q <= D;
        end if;
    end process P1;
end VERHALTEN;
```

Nur das Taktsignal muss sich in der Sensitivityliste des Prozesses befinden.

- ansteigende Flanke: `CLK='1' and CLK'event`
- abfallende Flanke: `CLK='0' and CLK'event`



# Prozesse ohne Sensitivitätsliste

- sind in der Simulation erlaubt;
- erfordern mindestens eine wait-Anweisung damit sie sich nicht „aufhängen“;
- Bsp.: 10-MHz-Taktgenerator:

```
CLKGEN: process  
begin  
    CLK <= '1';  
    wait for 50 ns;  
    CLK <= '0';  
    wait for 50 ns;  
end process CLKGEN;
```

**Prozesse ohne Sensitivityliste werden wie alle Prozesse bei Simulationsbeginn automatisch gestartet und bei jeder wait-Anweisung an dieser Stelle unterbrochen. Nach Beendigung werden sie automatisch neu gestartet.**

# Verwendung von Variablen in Prozessen

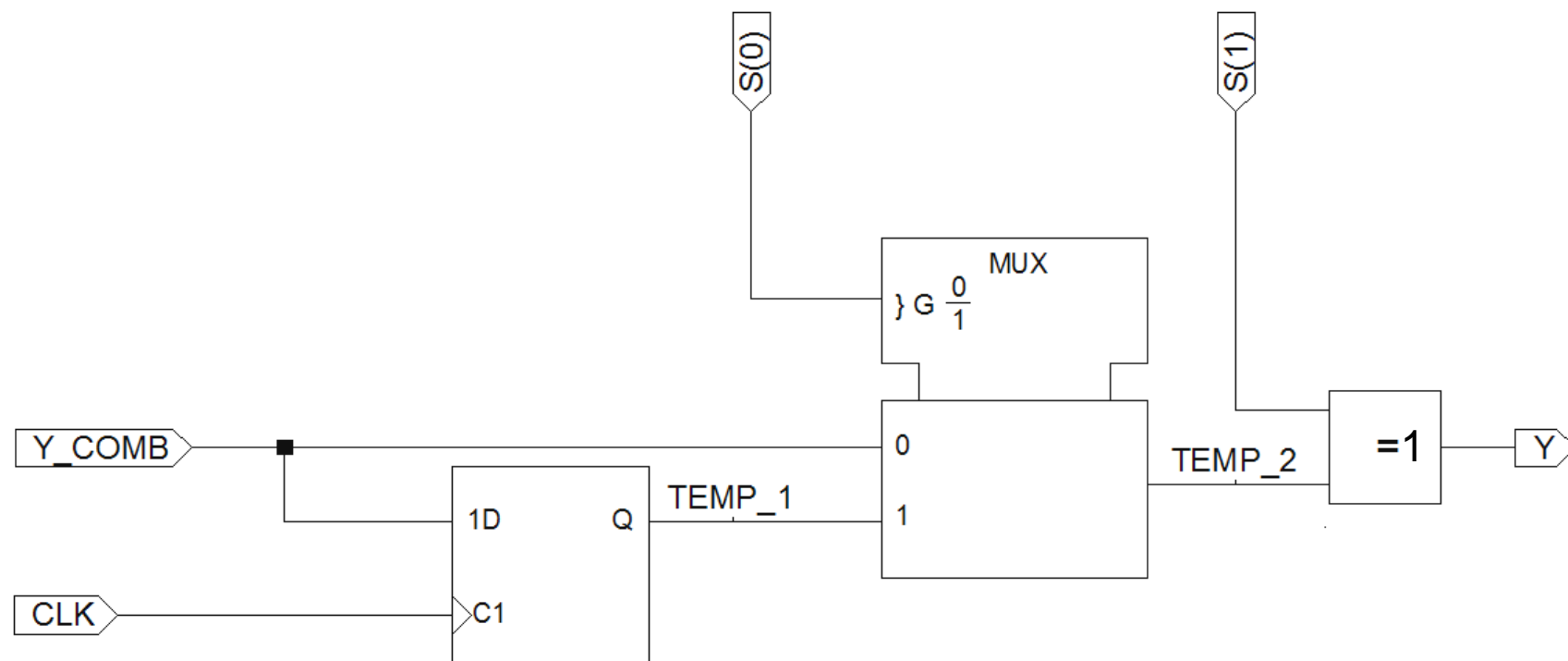
- VHDL-Variablen können zugewiesen und *sofort* abgefragt werden.
- Der Wertzuweisungsoperator für Variable ist das Symbol `,:=''`.
- Die Gültigkeit einer Variablen beschränkt sich auf den Prozess, in dem sie deklariert ist.

```
entity VAR_TEST is
    port( I1, I2 : in bit_vector(3 downto 0);
          Y : out bit);
end VAR_TEST;
architecture VERHALTEN of VAR_TEST is
begin
    COMB: process(I1, I2)
    variable TEMP: bit_vector(7 downto 0); -- Deklaration der Variablen
    begin
        TEMP := I1 & I2;                    -- Variable zur Verkettung zweier Bussignale
        if TEMP = "10101010" then          -- sofortige Auswertung der Variable
            Y <= '1';
        else
            Y <= '0';
        end if;
    end process COMB; end VERHALTEN;
```

# Entwurfsbeispiel: Ausgangsmakrozelle eines PLDs

**S(0) und S(1) sind Steuersignale, die die Funktion eines PLD-Ausgangs steuern.**

S(1)	S(0)	Ausgangsfunktion
0	0	nichtinvertiert, kombinatorisch: $Y = Y\_COMB$
0	1	nichtinvertiert, Registerausgang: $Y = TEMP\_1$
1	0	invertiert, kombinatorisch: $Y = \overline{Y\_COMB}$
1	1	invertiert, Registerausgang: $Y = \overline{TEMP\_1}$



# VHDL-Modell mit Testbench (Deklarationen)

Als Testbench besitzt die entity keine Port-Signale.

```
entity OLMC_TB is
end OLMC_TB;

architecture VERHALTEN of OLMC_TB is
-- port Signale des DUT:
signal CLK, Y_COMB : bit;
signal S : bit_vector(1 downto 0);      -- in port
signal Y : bit;                          -- out port
-- end port Signale des DUT
signal TEMP_1, TEMP_2: bit;              -- lokale Signale im DUT
signal TEST: integer range 1 to 8;       -- reines Testbench Signal
begin
```

Die Port-Signale des DUT werden als lokale Signale der architecture deklariert.

# VHDL-Modell (synthesefähige Prozesse)

```
-- Synthesefähige Prozesse (Device under Test DUT):
```

```
MUX: process (Y_COMB, TEMP_1, S(0))    -- aktivierende Signale f. komb.  
Prozess
```

```
begin
```

```
case S(0) is
```

```
when '0' => TEMP_2 <= Y_COMB;
```

```
when '1' => TEMP_2 <= TEMP_1;
```

```
end case;
```

```
end process MUX;
```

```
D_FF: process (CLK)                    -- nur Takt gesteuert
```

```
begin
```

```
if CLK'event and CLK = '1' then      -- ansteigende Flanke
```

```
TEMP_1 <= Y_COMB;                    -- Signalübernahme
```

```
else
```

```
TEMP_1 <= TEMP_1;                    -- gespeichertes Signal
```

```
end if;
```

```
end process D_FF;
```

```
Y <= TEMP_2 xor S(1);                -- gesteuerter Inverter nebenläufig
```

```
-- end DUT Prozesse
```

Multiplexer Prozess

Flip-Flop Prozess

Output Zuweisung  
mit XOR

# VHDL-Modell mit Testbench (Stimuli-Prozesse)

-- Testbench Prozesse:

CLKGEN: process

-- Taktgenerator 5 MHz

begin

CLK <= '0'; wait for 100 ns;

CLK <= '1'; wait for 100 ns;

end process CLKGEN;

Taktgenerator

STIMULI: process

-- Diskrete Stimuli f. 8 Tests

begin

TEST <= 1; Y\_COMB <= '1'; S <= "00"; wait for 200 ns;

TEST <= 2; Y\_COMB <= '1'; S <= "10"; wait for 200 ns;

TEST <= 3; Y\_COMB <= '0'; S <= "00"; wait for 200 ns;

TEST <= 4; Y\_COMB <= '0'; S <= "10"; wait for 200 ns;

TEST <= 5; Y\_COMB <= '1'; S <= "01"; wait for 200 ns;

TEST <= 6; Y\_COMB <= '1'; S <= "11"; wait for 200 ns;

TEST <= 7; Y\_COMB <= '0'; S <= "01"; wait for 200 ns;


TEST <= 8; Y\_COMB <= '0'; S <= "11"; wait for 200 ns;

end process STIMULI;

Test Stimuli

# VHDL-Modell mit Testbench (Response-Monitor)

```
RESPONSE_MONITOR: process          -- Prüfe Testergebnisse
begin
    wait for 150 ns;                -- 50 ns nach der steigenden Flanke
    for I in 1 to 8 loop            -- Prüfe 8 mal (insgesamt 1600 ns)
        case TEST is
            when 1 => assert Y = '1' report "Error: test 1";
            when 2 => assert Y = '0' report "Error: test 2";
            when 3 => assert Y = '0' report "Error: test 3";
            when 4 => assert Y = '1' report "Error: test 4";
            when 5 => assert Y = '1' report "Error: test 5";
            when 6 => assert Y = '0' report "Error: test 6";
            when 7 => assert Y = '1' report "Error: test 7"; -- Fehler
            when 8 => assert Y = '1' report "Error: test 8";
        end case;
        wait for 200 ns;            -- nächster Test nach 200 ns
    end loop;
end process RESPONSE_MONITOR;
-- end Testbench Prozesse:
end VERHALTEN;
```



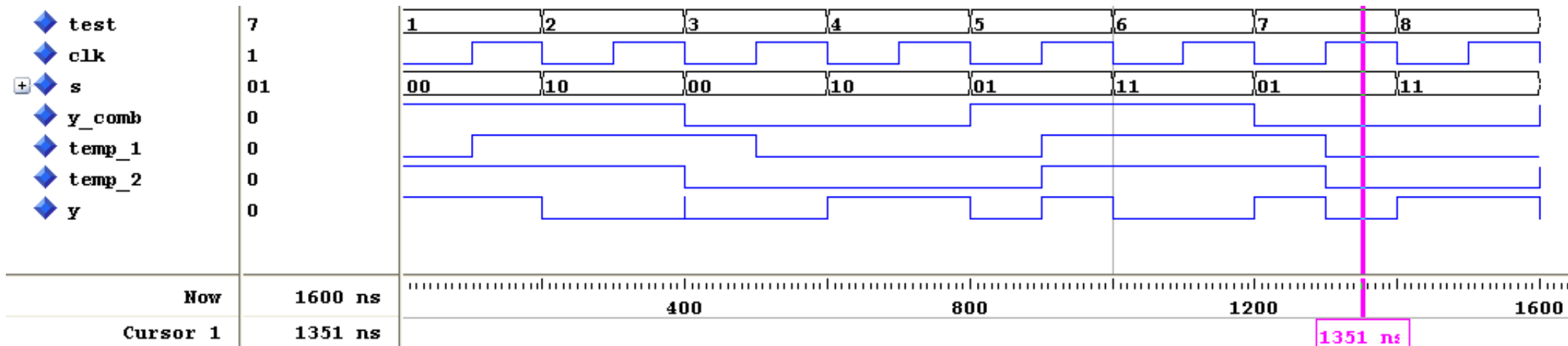
50 ns nach der steigenden Taktflanke

# Simulation des VHDL-Modells

- Meldung auf der Simulatorkonsole:

```
# ** Error: Error: test 7
```

```
# Time: 1350 ns Iteration: 0 Instance: /oc_tb
```





# Einführung in VHDL - Zusammenfassung

- **Modellierungsstile und Abstraktionsebenen**
- **Entity, Architecture, Signal**
- **DUT und Testbench**
- **Register Transfer Level Design**
- **VHDL Simulationssemantik, Delta Delay Model**
- **Prozesse**
- **Delaymodel und Zeitverhalten**
- **Sequentielle Anweisungen**