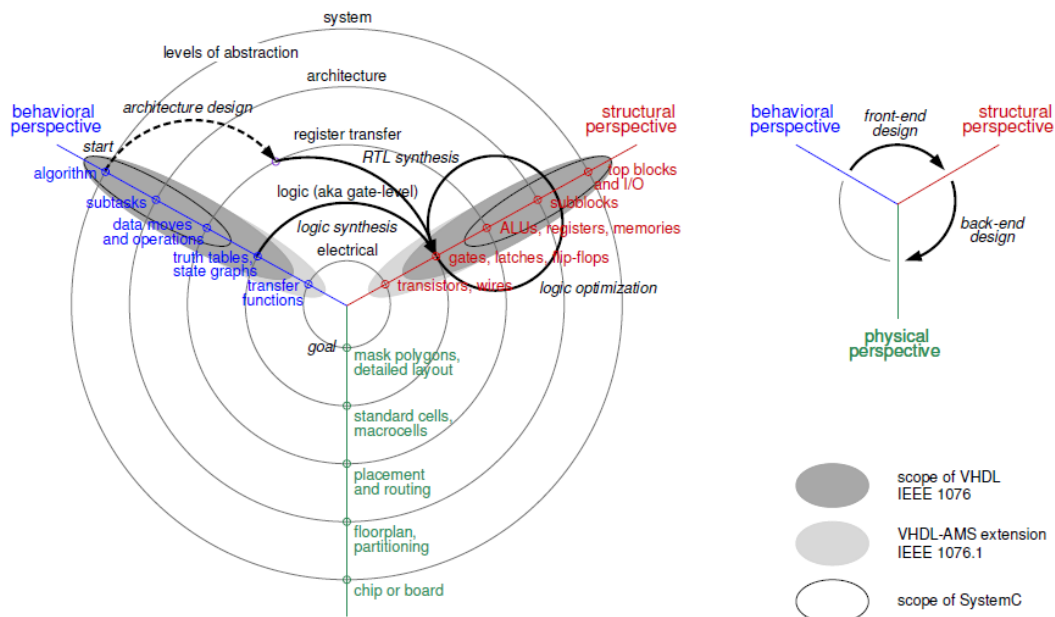


Hardware Synthesis



Key Concept I: Grundlagen, Modelle

- Kommentar: mit -- gestartet (bis Ende der Zeile)
- Bezeichner: Start mit Buchstabe, kein _ am Ende oder doppelt, case-insensitiv

Jedes VHDL-Modell besteht aus folgenden 3 Einheiten: Library, Entity, Architecture

Library

Work: Arbeitsbibliothek, Blöcke werden automatisch hier gespeichert
 Std: Alle Standardfunktionen von VHDL
 IEEE: Durch IEEE spezifizierte Funktionen

Verwendete Libraries werden am Anfang des Codes deklariert. Elemente werden mit Elementname.bibliotheksname angesprochen.

library ieee;

Mit use wird der Bibliotheksinhalt im ganzen VHDL Modul sichtbar gemacht.

use ieee.std_logic_1164.all;

Entity

Beschreibt Schnittstelle nach aussen

```
entity AND_OR_GATE is
  port(
    A, B, C    : in bit;
    Y          : out bit);
end AND_OR_GATE;
```

Signalrichtungen: in, out, inout, buffer

Architecture

Beschreibt inneren Aufbau des Moduls.

Strukturmodell

I/Os von vorhanden Komponenten verbinden (Netzliste), Blöcke werden unabhängig voneinander (parallel) ausgeführt.

Instanziierung realisiert Hierarchie

```
-- Architetre declarations
architecture structural of lfsr is

-- Signal declaration
signal verb1, verb2, verb3: std_logic;

-- Component declarations
component MUX
port (
    in1, in2:    in std_logic;
    oup:         out std_logic);
end component;

component FD2
port (
    CD, CP, D:   in std_logic;
    Q:           OUT std_logic;
end component;
```

Port Map realisiert Verbindungen

Implizit

```
mux_1: mux
    PORT MAP(RSTxRB, ENAxS, verb1 );
mux_2: mux
    PORT MAP(verb1, ENAxS, verb2);
U1_FF: core_block
    PORT MAP(CLKxC, ENAxS, verb1, verb2);
```

Explizit

```
mux_1: mux
    PORT MAP(in1 => RSTxRB, in2 => ENAxS, oup => ver
mux_2: mux
    PORT MAP(in1 => verb1, in2 => ENAxS, oup => ver
U1_FF: core_block
```

Datenflussmodell

Verhalten wird durch logische Gatter dargestellt

```
architecture DATENFLUSS of MUX is
signal NODE1, NODE2, NODE3: bit;
-- lokale Koppelsignale
begin
    NODE1 <= IB and S;
    NODE2 <= not S;
    NODE3 <= IA and NODE2;
    Y <= NODE1 or NODE3;
end DATENFLUSS;
```

Operatoren:

not, and, or, nand, nor, xor, xnor

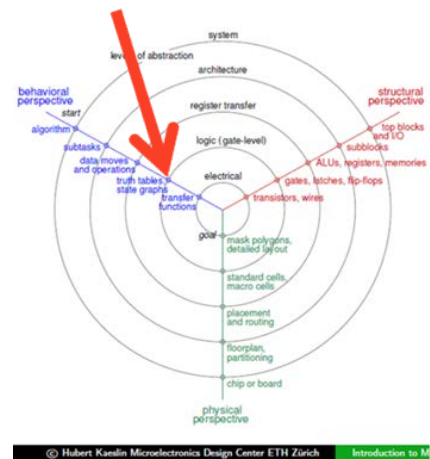
Anwendbar auf Typen:

bit, bit_vector, boolean

Verhaltensmodell

RTL-Modell (Register-Transfer-Logic), Prozesse

```
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;
```



Key Concept II: Nebenläufige Signalzuweisungen, Prozesse

Signalzuweisungsoperator:

$y \leq x;$

y: Signal, dem ein neuer Wert zugewiesen wird

x: bit Konstanten: '0', bit-Vector: „010“, Logik: A and B

Aggregate:

Klammerausdruck, der mehrere
bit zu bit_vector zusammenfasst

```
DATABUS <= (D1,D2,D3,D4);
```

```
-- ist identisch mit:
```

```
DATABUS (3) <= D1;
```

```
DATABUS (2) <= D2;
```

```
DATABUS (1) <= D3;
```

```
DATABUS (0) <= D4;
```

Unbedingte, gleichzeitige Signalzuweisung

A, U, V haben immer gleichen Wert

```
U <= A;
```

```
V <= U;
```

```
W <= A xor B;
```

```
X <= (A and S) or (B and not (S));
```

Verwendung: in kombinatorischer Logik, wenn keine Verzweigung erforderlich ist

Selektive Signalzuweisung

alle Bedingungen werden parallel abgearbeitet

alle Möglichkeiten müssen angegeben sein

```
with S select
```

```
Y <= E(0) when "00",
```

```
E(1) when "01",
```

```
E(2) when "10",
```

```
E(3) when others;
```

Verwendung: Kombinatorische Logik mit Verzweigung, typisch für Multiplexer

Bedingte Signalzuweisung

Bedingungen werden hintereinander abgearbeitet,

bis eine Bedingung true ist. Ist keine true, behält

Y seinen alten Wert.

```
Y <= E(0) when S = "00" else
```

```
E(1) when S = "01" else
```

```
E(2) when S = "10" else
```

```
E(3) when S = "11";
```

Verwendung: Kombinatorische Logik mit Verzweigung

Prozesse

- Alle Prozesse innerhalb einer Architecture sind nebenläufig
- Prozess=Hardwareblock
- Prozess wird aktiviert, wenn sich ein Signal in seiner Sensitivitätsliste ändert.
- Nur unbedingte Signalzuweisungen und sequentielle Anweisungen erlaubt
- Selektive und bedingte Signalzuweisungen VERBOTEN!
- Zuweisung erfolgt erst am Ende des Prozess!
- Wird kombinatorische Logik beschreiben, müssen sich alle Signale, die auf der rechten Seite einer Signalzuweisung stehen oder die sich in Entscheidungs-
ausdrücken sequenzieller Anweisungen befinden, in der Sensitivityliste des
Prozesses befinden.

```
process ( E, S)
begin
  case S is
    when "00" => y <= E(0);
    when "01" => y <= E(1);
    when "10" => y <= E(2);
    when others => y <= E(1);
  end case;
end process;
```

```
process (E, S)
begin
  if (S = "00") then y <= E(0);
  elsif (S = "01") then y <= E(1);
  elsif (S = "10") then y <= E(2);
  else y <= E(1);
  end if;
end process;
```

Prozess ohne Sensitivitätsliste: Nicht synthesefähig! Nur in Simulation / Testbenches erlaubt. Dann mit wait-Statement.

- Variablen: - Zugriff über := anstatt <=
- zwischen process und begin deklariert: *variable name : type*
- in kombinatorischen Prozessen: immer zuerst Wert zuweisen um D-Latch zu verändern.
- Schleifen: Ausgangssignal in Sensitivitätsliste → läuft dauernd! vermeiden
- Latches: unbeabsichtigter unerwünschter Speicher. Massnahmen: Default-Werte, explizite else-Blöcke, alle gelesenen Signale in Sensitivitätsliste

Clock: Flankenerfassung mit SIGNAL'event. Pos. Flanke:

D-Flipflop:

```
P1: process( CLK, RST ) -- nur clk und rst
begin
    if RST = '1' then -- zuerst asynchronen
        q <= '0';      -- Reset abarbeiten

    elsif CLK'event and CLK = '1' then
        -- clk - Flanke erfassen
        q <= d;         -- Synchroner Prozessteil
    end if;
end process;
```

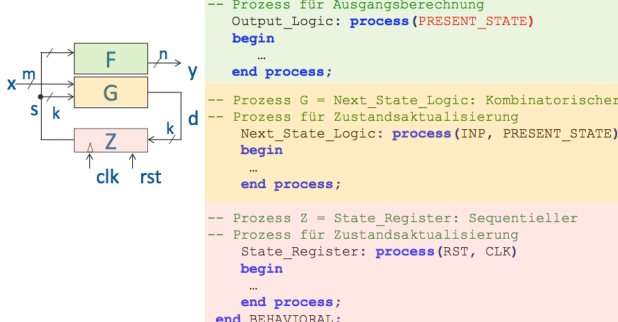
Zustandscodierung

Darstellung mit selbst erstelltem STATE_TYPE

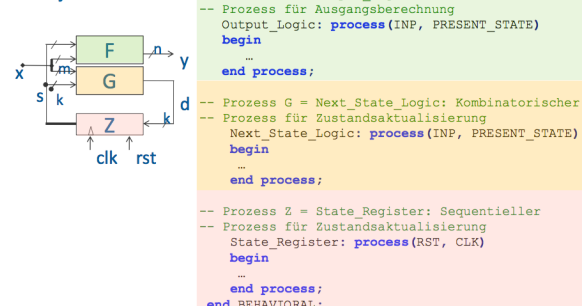
- Synthesewerkzeug: `type STATE_TYPE is (ST_NORMAL, ST_HSET, ST_MSET, ST_SSYNC);`
- ENUM_ENCODING: `type STATE_TYPE is (S0, S1, ... , Sn);`
`attribute ENUM_ENCODING: STRING;`
`attribute ENUM_ENCODING of STATE_TYPE: type is`
`"0001 0010 0100 ...";`
`signal PRESENT_STATE, NEXT_STATE: STATE_TYPE;`
`subtype STATE_TYPE is bit_vector(3 downto 0);`
`constant S0: STATE_TYPE:= "0001";`
`constant S1: STATE_TYPE:= "0010";`
`...`
`constant Sn: STATE_TYPE:= "1000";`
`signal PRESENT_STATE, NEXT_STATE: STATE_TYPE;`
- Konstanten:

Sequentielle Systeme

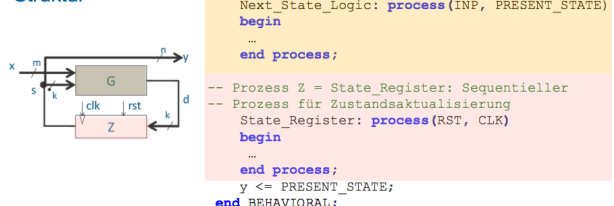
Moore Struktur

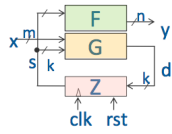


Mealy Struktur



Medwedjew Struktur

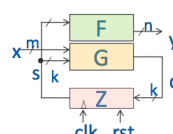


Prozess G:
Folgeberechnung

```

Next_state_logic: process(INP, PRESENT_State)
begin
    NEXT_State <= ST_RESET; -- Default Folgeberechnung setzen
    case PRESENT_State is -- Folgeberechnung berechnen
        when ST_1 =>
            if (Bedingung 1 als Fkt. der Eingänge)
            then NEXT_State <= ST_x1;
            else NEXT_State <= ST_y1;
            end if;
        when ST_2 =>
            if (Bedingung 2 als Fkt. der Eingänge)
            then NEXT_State <= ST_x2;
            else NEXT_State <= ST_y2;
            end if;
        when ...
        when others => null; -- ERROR state: Erforderlich für
    end case; -- CASE-Prüfung;
end process; -- Nur wenn Error state durch
-- Default gesetzt, kann null = no
-- action verwendet werden.

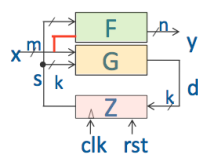
```

Prozess Z:
Zustandsspeicher aktualisieren

```

State_Register: process (RST, CLK)
-- Asynchroner Reset
-- taktischer Reset
-- Zustandswechsel
begin
    if (RST = '1') then
        PRESENT_State <= ST_RESET;
    -- Reset
    -- Zustandswechsel
    elsif (CLK = '1') and CLK'event then
        PRESENT_State <= NEXT_State;
    end if;
end process;
end BEHAVIORAL;

```

Prozess F:
Ausgangs-
berechnung

```

Output_Logik: process(
    PRESENT_State)
-- Ausgang bei Moore = f(s)
begin
    OUP <= '0';
    -- Defaultwert für
    -- Ausgang setzen
    IF (PRESENT_State = ST_1 and INP = i)
    then OUP <= ...;
    -- Ausgangszuweisung
    elsif (PRESENT_State = ...;
    -- abhängig von
    -- Zustand
    -- und Eingang
    -- (nur Mealy)
    end if;
end process;

```

Key Concept III: Elektrische Signale und Busse

Probleme bei Bit: Pull-Up, Pull-Down, Tri-State, Treiberkonflikte, Initialisierung

IEEE 1164

```

-- Library declarations
library ieee;
use ieee.std_logic_1164.all;

```

Wert	Bedeutung	Verwendung
'U'	Nicht initialisiert	Das Signal ist im Simulator (noch) nicht initialisiert
'X'	Undefinierter Pegel	Simulator erkennt mehr als einen aktiven Signaltreiber (Buskonflikt)
'0'	Starke logische 0	L-Pegel eines Standardausgangs
'1'	Starke logische 1	H-Pegel eines Standardausgangs
'Z'	Hochohmig, floatend	Three-State-Ausgang
'W'	Schwach unbekannt	Simulator erkennt Buskonflikt zwischen schwachen L- und H-Pegeln
'L'	Schwacher L-Pegel	Open-Source-Ausgang mit Pull-Down-Widerstand
'H'	Schwacher H-Pegel	Open-Drain-Ausgang mit Pull-Up-Widerstand
'-'	Don't-Care	Logikzustand des Ausgangssignals bedeutungslos, kann für Minimierung verwendet werden

Logikwert		Logischer Zustand		
		low	unknown	high
Treiberstärke	strong	0	X	1
	weak	L	W	H
	high impedance	Z	Z	Z

	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

std_logic: akzeptiert mehrere Treiber pro Kanal Gefahr: versehentliche Mehrfachzuweisungen werden nicht erkannt. Für Busse nötig.

```

bcd : in std_logic_vector(3 downto 0);
sseg_a : out std_logic;

```

std_ulogic: akzeptiert nur 1 Treiber pro Kanal. Compiler bricht bei Konflikten ab

```

bcd : in std_ulogic_vector(3 downto 0);
sseg_a : out std_ulogic;

```

Bussystem:

```

architecture TRI_STATE_OUT of BUSTREIBER is
begin
    WRITE: process (ENABLE, OUT_I)
    begin
        if ENABLE = '1' then I_O <= OUT_I;
        else I_O <= (others => 'Z');
        end if;
    end process WRITE;
    READ: IN_I <= I_O; -- paralleler Lesevorgang
end architecture;

```

Typkonversionen

Auf beiden Seiten eines Operators müssen die gleichen Datentypen stehen!

→ Konversionsfunktionen nötig

Konversionsfunktion	Argumenttyp	Ergebnistyp
To_bit	- std_ulogic	- bit
To_StdULogic	- bit	- std_ulogic
To_bitvector	- std_ulogic_vector - std_logic_vector	- bit_vector
To_StdULogicVector	- bit_vector - std_logic_vector	- std_ulogic_vector
To_StdLogicVector	- bit_vector - std_ulogic_vector	- std_logic_vector

IEEE 1076.3

numeric.std: Typen: signed (Zahl mit Vorzeichen), unsigned (Zahl ohne Vorzeichen)
Zahlen werden intern auf std_logic_vector abgebildet

numeric.bit: Typen: signed (Signal mit Vorzeichen), unsigned (Sig. ohne Vorzeichen)
Signale werden intern auf bit abgebildet

```
library ieee;
use ieee.std_logic_1164.all; signal A,B: unsigned(15 downto 0);
use ieee.numeric_std.all; signal C : signed(8 downto 0);

A_unsigned <= "1111"; -- = 15 decimal
B_signed <= "1111"; -- = -1 decimal
```

Vergleichsoperatoren: =, /= (ungleich),

<, <=, >, >=

Operatoren:

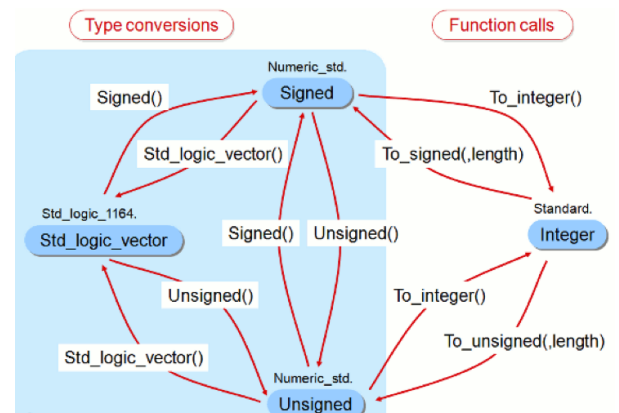
+, -, *, /,

abs(), [Absolutwert]

mod() [Modulo],

rem() [Rest],

** [Potenz]



Key Concept IV: Testbenches

Bestandteile: wie normal, library, entity (normalerweise leer), architecture

Prozesse: Clock-Erzeugung, Stimulus-Generator, Response-Monitor

Timing-Information als Konstante oder generic

Einfacher Signalverlauf (Stimulus):

```
s <= transport '0',
    '1' after 20 ns,
    '0' after 30 ns,
    '1' after 60 ns;
```

Clock Prozess

Variante 1:

```
signal clk: bit;
```

Variante 2:

```
signal clk: bit;
```

Lazy version 3:

```
signal clk: bit:= '0';
```

In Test-Benches kommen häufig Prozesse ohne Sensitivitätsliste zum Einsatz. Erlaubt, wenn Prozess ein **wait**-Statement steht.

Clock1: process

```
begin
```

```
    clk <= '0'; wait for (sim_cyc / 2);
```

```
    clk <= '1'; wait for (sim_cyc / 2);
```

```
end process;
```

Clock2: process

```
begin
```

```
    clk <= '0';
```

```
    loop
```

```
        wait for (sim_cyc / 2); clk <= not clk;
```

```
    end loop;
```

```
end process;
```

Clock3: clk <= not clk after (sim_cyc / 2);

Wiederkehrende Signalverläufe:

```
sequence: process
```

```
begin
```

```
    s <= '0';
```

```
    wait for 20 ns;
```

```
    s <= '1';
```

```
    wait for 10 ns;
```

```
end process sequence;
```

Auswahl einer architecture:

(nur nötig wenn mehrere vorhanden)

```
-- configuration
```

```
for all: exor use entity
```

```
work.exor(behavioral);
```

Allgemeiner Block zur Erzeugung von Input-Vektoren der Länge i.

```
-- Signal assignments for stimuli
Stimuli: process
    variable tb_in_vec: std_logic_vector(1 downto 0);
begin
    for i in 0 to 3 loop -- 2^4=16 Eingangsvektoren für 2 Eingänge
        tb_in_vec := int_2_v(i,2); -- Funktionsaufruf
        (tb_a, tb_b) <= tb_in_vec; -- Eingangssignale anlegen
        wait for sim_cyc; -- warten auf nächsten Zyklus
    end loop;
    wait;
end process;
```

Response-Monitor: - von Auge, nur bei kleinen, einfachen Blöcken möglich
- in separatem Prozess

Assert: gibt Fehlermeldung aus, wenn Bedingung nicht erfüllt.

```
-- Signal assignments for responses
Response: process
begin
    -- Resultate werden kurz vor Ende des
    wait for (sim_cyc - 1 ns); -- Simulationszyklus' abgefragt
    assert (tb_y = '0') report "error at vector 00" severity error;
    wait for sim_cyc;
    assert (tb_y = '1') report "error at vector 01" severity error;
    wait for sim_cyc;
    assert (tb_y = '1') report "error at vector 10" severity error;
    wait for sim_cyc;
    assert (tb_y = '0') report "error at vector 11" severity error;
end process;
```

Funktionen

- Unterprogramm mit Eingängen und einem Rückgabewert (wie in C)
- in Deklarationsteil (vor begin)

Beispiel:
Erzeugen von binären Testvektoren.

```
-- function declarations
function int_2_v(int, bits: integer) return
std_logic_vector is
    variable out_vec:
        std_logic_vector((bits-1) downto 0) := (others => '0');
    variable tmp: integer := int;
begin
    for i in 0 to (bits-1) loop
        if ((tmp mod 2) = 1) then
            out_vec(i) := '1'; -- einzelne bits des Testvektors
                                -- werden erzeugt
        end if;
        tmp := tmp/2; -- ganzzahlige Division, integer
    end loop;
    return out_vec;
end int_2_v;
```

```
process (inp,b)
begin
    for i in 0 to 7 loop
        out(i) <= inp(i) and b; -- 8bit bus "inp" mit Einzel-
                                -- signal "b" and-verknüpfen
    end loop;
end process;
```

FOR-Loop

Sequentielles Statement. Wird in process, function, procedure angewendet.

Zeitablauf-Steuerung

Der Simulator arbeitet ereignisbasiert, d.h. jedes Ereignis oder event wird in eine Liste eingetragen, diese wird dann abgearbeitet. Jedes Event wird zu einer genau definierten Zeit (Simulationspunkt) ausgeführt. Simulationspunkte sind um infinitesimal kleine Zeiteinheit Δ verschoben.

Der Delta-Zyklus stellt Ursache-Wirkung dar:

1. Warten bis Prozess durch Signal-Event in sensitivity list getriggert
2. Prozessausführung: alle aktiven Prozesse bis zum Ende bzw. nächsten wait ausführen
3. Signalzuweisung: entsprechende Zuweisungen ausführen. Evtl. weitere Prozesse aktivieren

inertial delay model: Änderung nur, wenn die Änderung des Eingangs länger ist als die Verzögerung. Verwendung: um Glitches, Störungen zu filtern

$Y <= A \text{ or } B \text{ after } 9 \text{ ns};$
 $Y <= \text{inertial } A \text{ or } B \text{ after } 9 \text{ ns};$

transport delay: Änderung wird in jedem Fall nach der definierten Zeit ausgegeben
 $Y <= \text{transport } A \text{ or } B \text{ after } 9 \text{ ns};$

Key Concept V: Modell-Parametrisierung

Zähler: immer synchron realisieren → keine Timing-Probleme
verschiedene Zähler sind ähnlich → Re-Use Prinzip → generic

Generic

Übergabe von Parametern an ein Modell

```
entity COUNT_X is
  generic(
    MAXCOUNT: integer := 127);
  port(
    CLK, NRST, ENA: in std_logic;
    OUP: out integer range 0 to MAXCOUNT);
end; ...
```

Anwendungen:

```
begin
  dut1 : COUNT_X
    generic map(maxcount1)
    port map(tb_clk, tb_nrst, tb_ena, tb_oup1);
  dut2 : COUNT_X
    generic map(maxcount=>256)
    port map(tb_clk, tb_nrst, tb_ena, tb_oup2);
```

- Parameter zu Designzeit: Konstanten (z.B. Zählbereich 0...4)
- Parameter zu Compilezeit: Generic (z.B. Zählbereich 0...maxcount)
- Parameter zu Laufzeit: Signale

