# 1 Realisierungs Methoden

## 1.1 ROM

Mit einem ROM lassen sich sich rein kombinatorische Schaltungen in Form einer Look up Table realisieren.

- Eingangsvariablen = Adresse
- Speicherwert = Ausgang (programmierbar)

## 1.2 PLD

Programmierbares Device aus AND und OR-Matrix, mindestens eine Matrix programmierbar.

- PAL → OR-Matrix fest, AND-Matrix programmierbar, Fuses
- PLA → OR und AND Matrix frei programmierbar, Fuses
- GAL → Wie PLA plus programmierbare Ausgangsnetzwerke (Tristate), EEPROM

SPLD (Simple PLD): Für Funktionen die als DNF vorliegen geeignet, heute grösstenteils von CPLD und FPGA verdrängt.

## 1.3 CPLD (Complex PLD)

- Verbund PLD Makrozellen die mit Bussen verbunden sind, Speicherung der Konfiguration in Flash.
- Durch regelmässige Struktur sind Signallaufzeiten vorhersagbar.
- Wegen grosser Zahl an Logikblöcken sehr gut für parallele Prozesse geeignet.

## 1.4 FPGA

2D-Array von Logikblöcken, die über Routing Kanal und Schaltmatrizen miteinander und mit I/O verbunden werden.

- Logikblock (LogicCell) → Lookuptable mit D-FlipFlop, kann beliebige Funktionen ausführen
- Schaltmatrizen → programmierbare Verbindungen
- Makrozellen → Feste Funktionen z.B. Memory, Clock Managment...

Die Konfiguration wird im RAM gespeichert (flüchtig). D.h. bei jedem Boot muss der Code von einem Festspeicher geladen werden.

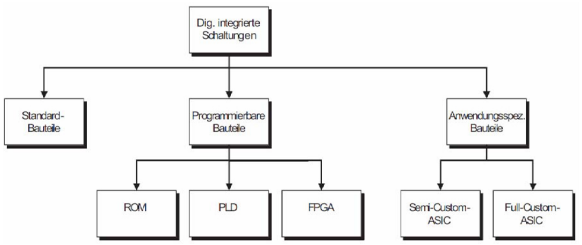
## 1.5 Semicustom IC

- Mikrozellen aus p- und n-FETs werden durch Verdrahtung zu Gates →Gate-Array/Sea of Gates
- Gates können durch Verdrahtungskanäle verbunden werden.
- Standardfunktionen können mit IP (intelegent properity)-Cores implementiert werden.
- In Mixed-Signal Arrays sind zusätzlich spezifische Analogbauteile enthalten

## 1.6 Fullcustom IC

Völlig kundenspezifische ICs, oft werden IP-Cores für Standardfunktionen verwendet. Digitale und analoge Komponenten auf einem IC möglich. Voll auf Anwendung anpassbare Eigenschaften (Stromverbrauch, Grösse, Geschwindigkeit etc.).

## 1.7 Vergeichstabelle




| Kriterien | Standard Bauteile | ROM | PLD | FPGA | Semicustom | Fullcustom |
|---|---|---|---|---|---|---|
| Machbarkeit | ++ | - - | - - | + | + | +++ |
| Zeit Realisiertung | + | ++ | ++ | ++ | - | - - |
| Iterationszeit | - | ++ | ++ | ++ | - | - - |
| NRE | ++ | + | + | + | - | - - - |
| Stückpreis | - - | + | + | - | + | +++ |

# 2 VHDL

Die vollständige Beschreibung des Designs (Grundstruktur) besteht aus:

- Bibliothekenbeschreibung [library]
- Schnittstellenbeschreibung [entity]
- Architekturbeschreibung [architecture]

## 2.1 Key Concepts

| | |
|---|---|
| Key Concept I: | Schaltungshierarchie und Verbindung von Sub-Blöcken (hierarchy and connectivity). |
| Key Concept II: | Nebenläufige (concurrent) Prozesse und Prozess-Interaktion. |
| Key Concept III: | Modellierung des elektrischen Verhaltens von Signalen. |
| Key Concept IV: | Event-Based time: Simulationsmodell, das auf Events und nicht auf kontinuierlicher Zeit beruht. |
| Key Concept V: | Parametrisierung von Modellen. |

## 2.2 Identifier

- Start mit Buchstabe
- _ nicht am Ende oder doppelt
- case insensitive
- Eigene Identifier → GROSSBUCHSTABEN
- Library Identifier → kleinbuchstaben

## 2.3 Bibliotheken (library)

| | |
|---|---|
| work | Default-Bibliothek des Benutzers |
| std | standard (Vordefinierte Datentypen und Funktionen) |
| | textio (Text - Filehandling) |
| work und std | müssen nicht deklariert werden (automatisch eingebunden) |
| ieee | std_logic_1164: Datentypen für mehrwertiges Logiksystem |
| | numeric_std: Mathe-Bibliothek für std_logic |

```
library <library_name> {,<libary_name>};
use <library_name>.<element_name>;
--oder
use <library_name>.all
```

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

Mit **use** wird Bibliotheksinhalt im ganzen VHDL Modul sichtbar

## 2.4 Schnittstellenbeschreibung (entity)

Die einzelnen Blöcke einer VHDL-Beschreibung kommunizieren über ihre Schnittstellen miteinander. Die Kommunikationskanäle nach aussen sind die sogenannten Ports. Für diese werden in der Schnittstellenbeschreibung Name, Signalflussrichtung und Datentyp festgelegt. Mit der Signalflussrichtung werden Eingänge (IN), Ausgänge (OUT) und bidirektionale Ports (INOUT) unterschieden.

```
entity <ENTITY_NAME> is
  port(
    {<PORT_NAME>: <mode> <type>;}
    );
end <ENTITY_NAME>;
```

```
entity DEVICE is
  port(
    A, B: in bit_vector (3 downto 0);
    Y: out bit);
end DEVICE;
```

### 2.4.1 Signalflussrichtungen (mode)

| | |
|---|---|
| in: | Eingangssignal. Darf nur rechts stehen. |
| out: | Ausgangssignal. Darf nur links stehen. |
| buffer: | Ausgangssignal. Darf auch rechts stehen → **problematisch** |
| inout: | Bidirektionales Signal, in Verbindung mit Typ std_logic. |

### 2.4.2 Signaltypen (type)

| | |
|---|---|
| boolean | Werte: true, false |
| bit | Werte: '0','1' |
| bit_vector: | eindimensionaler Array von bits |
| integer | interne Darstellung mit 32bit Range Einschränkung nötig! |
| std_logic(_vector) | wie bit(_vector), für mehrwertige Logik |
| std_ulogic(_vetor) | wie std_logic, nur für einen Treiber |

## 2.5 Architekturbeschreibung (architecture)

Die Architektur legt die Funktion eines Blocks fest. Dabei kann eine Entity mehrere Architecturen besitzen. Für das Erzeugen der FPGA-Programmierung sollten aber alle nicht verwendeten Architekturen auskommentiert werden.

```
architecture <ARCHITECTURE_NAME> of <ENTITY_NAME> is
  -- Signaldeklerationen
  signal ....
  -- Komponentdeklerationen
  component ...
begin
  -- Anweisungsteil
end <ARCHITECTURE_NAME>;
```

### 2.5.1 Signaldeklaration

Hier werden die Signale, die Innerhalb der Architektur verwendet werden, deklariert.

```
signal <sig_name> {, <sig_name>}: type;
```

Bsp: `signal sig1, sig2: bit;`

### 2.5.2 Komponentendeklaration

Mit Hilfe des Schlüsselwortes **component** erfolgt die Deklaration von Komponenten, ähnlich wie einer **entity**.

```
component <COMPONENT_NAME>
  port(
    {<PORT_NAME>: <mode> <type>;}
    );
end component;
```

### 2.5.3 Anweisungsteil

Im Anweisungsteil wird das Verhalten beschrieben. Diese wird dabei durch verschiedene Modelierungsstile unterschieden.

#### 2.5.3.1 Strukturmodell (Instanzierung)

Die Ein- und Ausgänge von Komponenten, die in einer Bibliothek abgelegt sind werden durch lokale Signale miteinenander verbunden → Es entsteht eine Netzliste.

```
-- Implizite form:
[identifier]: component_name
  port map(signal1, signal2, signal3);
-- Zuweisung entspr. Reihenfolge von
-- Auflistung in Entity des Komponenten
```

```
--Explizite form:
[identifier]: component_name
  port map(
    in2 => signal2,
    out => signal3,
    in1 => signal1);
-- Allg: <SigComponent> => <SigArchitecture>
```

#### 2.5.3.2 Datenflussmodell

Es werden logische Grundfunktionen verwendet. Logikoperatoren für bit, bit_vetor, boolean: **not, and, or, nand, nor, xor, xnor** Wenn auf bit_vector angewendet, dann muss Bitbreite von beiden Operanden gleich sein.

```
-- Beispiel fuer NAND3
Y <= not (A and B and C);
-- Beispiel fuer bit_vector(Q,R,S)
R <= S(3 downto 0);
X <= not Q(2);
```

```
-- Falsch, Syntaxfehler:
Y <= A nand B nand C;
-- Erlaubt (aber nicht NAND3):
Y <= (A nand B) nand C;
```

#### 2.5.3.3 Verhaltensmodell (→ Häufigster Modellierungsstil.)

Die Modellierung findet durch Sprachelemente ähnlich wie in einer prozeduralen Programmiersprache statt.
Dabei werden abgeschlossene Hardware-Blöcke durch Prozesse abgebildet

## 2.6 Signalzuweisung

Alle Signalzuweisungen und alle Prozesse laufen parallel zueinander. Signalzuweisungen sind immer aktiv. Signale können auf verschiedene Arten zugewiesen werden:

```
-- Zuweisungsoperator <=
-- type: bit
Y <= X;
A <= '0';
-- type: bit_vector
Q <= "1010";
-- logische Ausdruecke
C <= D and E;
```

```
-- Unbedingte Signalzuweisung
architecture UNBEDINGT of ... is
begin
  U <= A;
  V <= not U;
  W <= A xor B(0);
  X <= (A and S) or (B(1) and not S);
end UNBEDINGT;
```

```
--Selektive Signalzuweisung
architecture SELEKTIV of ... is
begin
  with S select
    Y <=  E(0) when "00",
          E(1) when "01",
          E(2) when "10",
          E(3) when others;
end SELEKTIV;
```

```
--Bedingte Signalzuweisung
architecture BEDINGT of ... is
begin
  Y <=  E(0) when S = "00" else
        E(1) when S = "01" else
        E(2) when S = "10" else
        E(3) when others;
end BEDINGT;
```

### 2.6.1 Aggregat

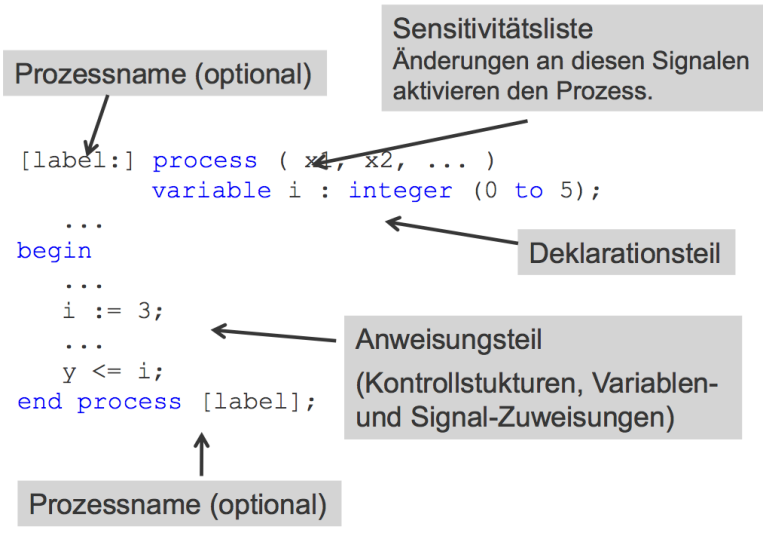
Ein Aggregat ist ein Klammerausdruck, der mehrere Einzelelemente zu einem Vektor zusammenfasst.

```
signal DATABUS : bit_vector(3 downto 0);
signal D1, D2, D3, D4 : bit;
...
-- implizit
DATABUS <= (D1, D2, D3, D4);
-- explizit
DATABUS <= (1=> D3, 0=> D4, 3=> D1, 2=> D2);
-- ist identisch mit:
DATABUS(3) <= D1;
DATABUS(2) <= D2;
DATABUS(1) <= D3;
DATABUS(0) <= D4;
```

```
--oder mit Werten
DATABUS <= ('1', '0', '1', '0');
DATABUS <= (3 => '1', 1 => '1', others => '0');
DATABUS <= "1010";
```

## 2.7 Prozesse

- Prozesse werden durch Änderungen der Signale in der Sensitivitätsliste aktiviert und ausgefährt.
- Prozesse verhalten sich gegen aussen nebenläufig, innerhalb werden Anweisungen sequentiell abgearbeitet
- **Selektive und bedingte Signalzuweisungen sind verboten.**
- Unbedingte Signalzuweisung sind erlaubt.
- **Aktualisierung aller Signale geschieht immer erst am Prozessende!**
- Zuweisen eines **Default-Wertes** an alle Ausgangssignale vor der ersten if-Anweisung zur Vermeidung von Latches
- Hint: Damit Code übersichtlich und verständlich bleibt, sollte **nicht zu viel Funktionalität** in einen einzigen Prozess verpackt werden.
- Alle Signale, die auf der rechten Seite einer Signalzuweisung stehen gehören in Sensitivitätsliste
- Ohne Sensitivitätsliste nur in Simulation erlaubt



### 2.7.1 Variablen

- Variablen werden im Deklarationsteil des Prozesses deklariert und nur in diesem Prozess sichtbar.
- Zugewiesener Wert kann sofort abgefragt werden
- Wertzuweisung durch Operator **:=** (nicht <=)
- Vor Verwendung ein aktueller Wert (evt. Default-Wert) zuweisen, sonst entsteht Latch

```
variable var_name {, var_name}: type_name
                [:= value];
```

### 2.7.2 Signalzuweisung durch Prozesse

```
-- Kontrollstruktur mit if.. then..
--                      elsif..else
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;
```

```
--Kontrollstruktur mit case
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;
```

## 2.8 Sequentielle Systeme

### 2.8.1 Flankenerfassung

```
--Erfassen einer positiven Flanke
--     des Signals CLK
if CLK'event and CLK = '1' then ...
```

```
--Erfassen einer negativen Flanke
--     des Signals CLK
if CLK'event and CLK = '0' then ...
```

### 2.8.2 Zustandscodierung

- Die Deklaration der Zustandscodierung erfolgt im Deklarationsteil der Architektur
- Bei expliziten Zustandscodierung muss die Zustandcodierung der IDE auf "User" umgestellt werden.

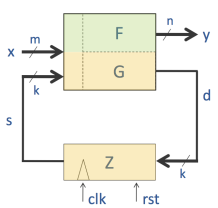
```
-- Implizite Form:
-- Darstellung mit Aufzaehltyp
type STATE_TYPE is (S0, S1, ..., Sn);
signal PRESENT_STATE, NEXT_STATE: STATE_TYPE;
```

```
--Explizite Form:
--Mit 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;

-- Mit Konstanten
subtype STATE_TYPE is bit_vector(3 downto 0);
constant S0: STATE_TYPE:= "0001";
constant S1: STATE_TYPE:= "0010";
...
signal PRESENT_STATE, NEXT_STATE: STATE_TYPE;
```

### 2.8.3 FSM in 3-Prozess Struktur

Die Zustandsmaschine wird in 3 Prozesse aufgeteilt. Diese Prozesse werden ganz einfach einzeln im Anwendungsteil der Architektur implementiert und arbeiten jeweils nebenläufig.




- Prozess F = Output_Logic
- Prozess G = Next_State_Logic
- Prozess Z = State_Register

#### 2.8.3.1 Output_Logic

Dieser kombinatorische Teil ist abhängig von der Struktur der FSM.

```
-- Mealy-Struktur
Output_Logic: process(INPUT, PRESENT_STATE)
begin
   ...
end process;
```

```
-- Moore-Struktur
Output_Logic: process(PRESENT_STATE)
begin
   ...
end process;
-- Medwedjew-Struktur
Output_Logic:
   y <= PRESENT_STATE;
```

#### 2.8.3.2 Next_State_Logic

Dieser kombinatorische Teil ist nicht abhängig von der Struktur der FSM.

```
Next_state_logic:process(INPUT,PRESENT_STATE)
  begin
    -- Default Folgezustand
    NEXT_STATE <= PRESENT_STATE;
    case PRESENT_STATE is
      when S0 =>
        if (Bedinung in f(Eingaenge)) then
          NEXT_STATE <= Sx0;
        else NEXT_STATE <= Sy0;
        end if;
      when S1 =>
        ....
      when others => null;
      -- oder  NEXT_STATE <= Reset_State;
    end case;
  end process;
```

#### 2.8.3.3 State_Register

Dieser kombinatorische Teil ist nicht abhängig von der Struktur der FSM und sieht bei allen (fast) FSM immer gleich aus.

```
State_Register : process(CLK, NRST)
  begin

    --Asynchroner Reset
    if NRST = '0' then
      PRESENT_STATE <= Reset_State;

    --taktsynchroner Zustandswechsel
    elsif CLK'event and CLK = '1' then
      PRESENT_STATE <= NEXT_STATE;
    end if;
  end process;
```

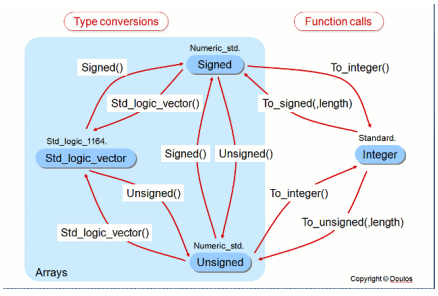
## 2.9 IEEE 1164 Logiksystem

Durch Einbinden der ieee.std_logic_1164 Library werden die std_logic, sowie std_ulogic Datentypen verfügbar. Damit verbunden ist auch die 9-wertige Logik:

| 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 |

### 2.9.1 Datentypenkonversion

Ebenfalls in der ieee.std_logic_1164 Library sind folgende Typenkonversionen enthalten:




| Konversionsfunktion | Argumenttyp | Ergebnistyp |
|---|---|---|
| To_bit | - std_ulogic | - bit |
| To_StdULogic | - bit | - std_ulogic |
| To_bitvector | - std_ulogic_vector<br>- std_logic_vector | - bit_vector |
| To_StdULogicVector | - bit_vector<br>- std_logic_vector | - std_ulogic_vector |
| To_StdLogicVector | - bit_vector<br>- std_ulogic_vector | - std_logic_vector |

## 2.10 Arithmetik

Durch zusätzliches Einbinden der ieee.numeric_std neben ieee.std_logic_1164 stehen signed und unsigned Datentypen auf Basis des std_logic_vector Datentyp, sowie arithmetische Operatoren und Vergleichsfunktionen zur Verfügung.

| Vergleichsoperator | Bedeutung | Beispiel |
|---|---|---|
| = | gleich | ... when A = B ... |
| /= | ungleich | ... when A /= B ... |
| < | kleiner | ... when A < B ... |
| <= | kleiner oder gleich | ... when A <= B ... |
| > | größer | ... when A > B ... |
| >= | größer oder gleich | ... when A >= B ... |

| Operator | Bedeutung | Beispiel |
|---|---|---|
| + | Addition | Y <= A + B |
| - | Subtraktion | Y <= A - B |
| abs | Absolutwertbildung | Y <= abs(A) |
| * | Multiplikation | Y <= A * B |
| / | Division | Y <= A / B |
| ** | Potenzbildung | Y <= 2**A |
| mod | Rest der Division A/B<br>Das Vorzeichen des Ergebnisses ist gleich dem von B. | Y <= A mod B |
| rem | Rest der Division A/B.<br>Das Vorzeichen des Ergebnisses ist gleich dem von A. | Y <= A rem B |

## 2.11 Simulation

### 2.11.1 Aufbau einer Test-Bench

Normales VHDL Modul bestehend aus:

- library
- entity (i.d. Regel ohne Ports nach aussen)
- architecture
  - Komponenten(DUT) deklerations
  - ggf. Auswahl der DUT-Architektur
  - (Timing-Informationen als Konstante (oder Generic) deklarieren)
  - Signale deklarieren(Bennenung: <DUT_SIG>_tb)
  - Instanzierung der DUT
  - Prozess für Clock-Erzeugung
  - Prozess für die Anwendung von Stimuli (Stimulusgenerator)
  - Prozess für das Erfassen der Systemantwort (Response-Monitor)

#### 2.11.1.1 DUT einbinden und konfigurieren (in Deklarationsteil)

DUT wird als Komponente ganz normal deklariert. Sind mehrere Architekturen vorhanden, muss die zu stimulierende Architektur ausgewählt werden:

```
for all : <ENTITY_NAME> use entity work.<ENTITY_NAME>(<ARCH_NAME>);
```

#### 2.11.1.2 Timing-Konstante

```
constant sym_cyc: time := 100 ns;
```

#### 2.11.1.3 Clock-Prozess

```
signal clk_tb : bit;

stimuli_clk : process
  begin
    clk_tb <= '0';
    loop
      wait for (sym_cyc / 2);
      clk_tb <= not clk_tb;
    end loop;
    wait;
  end process;
```

#### 2.11.1.4 Stimulusgenerator

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

```
-- Wiederkehrender Signalverlauf
stimuli : process
  begin
    s <= '0';
    wait for 20 ns;
    s <= '1';
    wait for 10 ns;
-- Mit [wait;] ebenfalls nur einfach
  end process;
```

#### 2.11.1.5 Response-Monitor (inkl. Assert)

```
-- Response Monitor
Response: process
  begin       -- Resultate werden kurz vor Ende des Sim_cyc
    wait for (sim_cyc - 1 ns);    -- abgefragt (Versatz 1ns)
    assert (tb_y = '0') report "error at vector 00" severity error;
    wait for sim_cyc;
    assert (tb_y = '0') report "error at vector 01" severity error;
    wait for sim_cyc;
    ...
    [wait;] -- Wiederkehrend, oder nur einfach
  end process;

--Assert im allgemeinen (ist condition = false, so wird assert angezeigt)
assert <condition> [report <"str_expression">] [severity note|warning|error|failure];
```

#### 2.11.2 For-Loop

For-Loops finden Verwendung in der Simulation, oftmals in Verbindung mit wait-Statemants und daher ohne Sensititvitätsliste.
Sie können aber auch synthetisiert werden, dafür muss aber der range endlich sein.

```
process
  begin
    for i in 0 to 9 loop
      ...
      wait for sim_cyc;
    end loop;
  [wait;] -- Wird einmal abgehandelt
end process;
```

#### 2.11.3 Verzögerungszeiten

- Δ-Time → funktioneller Zusammenhang zwischen Ursache/Wirkung (automatisch)
- inertial delay (Trägheit) →Ausgang ändert erst, wenn Eingang länger konstant bleibt als mit after definiert (Nicht verwenden für Verzögerungszeit)

```
Y <= inertial A or B after 9 ns;
--Achtung ist das gleiche:
Y <= A or B after 9 ns;
```

- transport delay → Ausgang ändert nach Eingansänderung mit after definierter Zeit

```
Y <= transport A or B after 9 ns;
```

### 2.12 Generic

→ Re-Use! So kann VHDL mit Generic, Parameter an ein Modell übergeben. Dabei wird der Parameter wie eine Konstante im Anweisungsteil verwendet.
Dafür muss in jeder Schnittstellenbeschreibung der Generic-Block eingebaut werden:

```
generic(param_name: param_type:=initial_value[;param_name: param_type:=initial_value]);
```

Und bei der Instanzierung wird mit generic map der Parameter-Wert gesetzt.

#### 2.12.1 Beispiel für Implementation

Die Implemantation geschieht wie auch bei den anderen Komponenten meistens in einer seperaten Datei, einfach enthält hier die entity noch ein generic.

```
entity genCount is
  generic(
    -- 255 ist einfach Default-Value
    MAXCOUNT: integer := 255);
  port(
    clk, nrst, ena : in bit;
 --MAXCOUNT wird als "Konstante" verwendet
    oup : out integer range 0 to MAXCOUNT;
    overflow : out bit);
end genCount;
```

#### 2.12.2 Beispiel für Instanzierung

Auch hier wird wie gewohnt zu erst die Komponente deklariert und anschliessend im Anwendungsteil instanziert.

```
-- Deklaration des Komponenten
  component genCount
    generic(MAXCOUNT : integer := 255);
    port(
      clk, nrst, ena : in bit;
      oup : out integer range 0 to MAXCOUNT;
      overflow : out bit);
  end component;

-- Instanzierung des Komponenten
count_TIME : genCount
  generic map(MAXCOUNT_TIME)
  port map(clk => clk,
           ...);
```

### 2.13 VHDL Code Example Statemachine

```
architecture tool_codierung of driver is
type state_type is (reset, s1, s2);
signal present_state, next_state : state_type;
begin
  State_Register: process(nrst, clk)
  begin
    if nrst = '0' then
      present_state <= reset;
    elsif clk = '1' and clk'event then
      present_state <= next_state;
    end if;
  end process;
  Next_state: process(present_state, <other>)
  begin
    case present_state is
      when reset =>   <Do something>;
      when s1 =>      <Do something>;
      when s2 =>      <Do something>;
      when others =>  <Do something>;
    end case;
  end process;
  Output: process(present_state, <other>)
  begin
    case present_state is
      when reset =>   <Do something>;
      when s1 =>      <Do something>;
      when s2 =>      <Do something>;
      when others =>  <Do something>;
    end case;
  end process;
end tool_codierung;
```