

Digital Design

Digitaltechnik mit VHDL

F.Dubacher

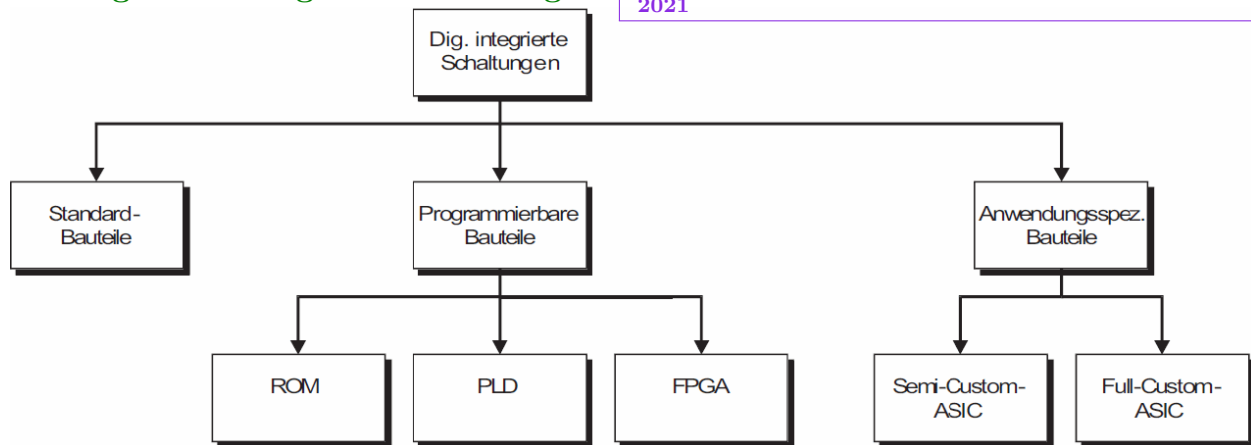
1. September 2021

Inhaltsverzeichnis

1	Allgemeines	1
1.1	Realisierungsformen digitaler Schaltungen	1
1.1.1	Entscheidungskriterien	1
1.1.2	Standardbauteile	1
1.1.3	Programmierbare Bauteile	1
1.1.4	ASIC (Application Specific Integrated Circuit)	1
1.2	Digitaler Design Flow	2
1.2.1	Synthese und Optimierung	2
1.2.2	Technology Mapping	2
1.2.3	Place & Route	2
1.2.4	Simulation	2
1.3	Hardware-Beschreibung mit VHDL	2
2	Key Concept 1: Hirarchie und Konnektivität	2
2.1	Library	3
2.1.1	Arten von Libraries	3
2.2	Entity	3
2.3	Architecture	3
2.3.1	Strukturbeschreibung	3
3	Key Concept 2: Nebenläufigkeit und Prozesse	3
3.1	Nebenläufige Signalzuweisung	3
3.2	Nebenläufige Prozesse	4
3.3	Modellierung sequenzieller Schaltungen	5
3.3.1	Bubble Diagram	5
3.4	Endliche Zustandsautomaten mit VHDL (3 Prozess-Modell)	5
3.4.1	Allgemeine Struktur	5
3.4.2	Output Logic Mealy	6
3.4.3	Output Logic Moore	6
3.4.4	Output Logic Medw.	6
3.4.5	Manuelle State-Codierung	6
4	Key Concept 3: Diskreter Ersatz für elektrische Signale	6
4.1	Das IEEE 1164 Logik System	6
4.1.1	std_ulogic (_vector)	6
4.1.2	std_logic (_vector)	6
4.2	Bidirektionale Busse in VHDL	7
5	Key Concept 4: Ereignisbasiertes Konzept zur zeitlichen Darstellung bei Sim.	7
5.1	Modellierung des zeitlichen Verhaltens in VHDL	7
5.1.1	Event Queue	7
5.1.2	Delta Time Modell	7
5.1.3	Transport Delay Modell	7
5.1.4	Inertial Time Modell	7
5.2	Simulationsaufbau	7
5.2.1	Beispiel einer Test-Bench für die Komponente "xor2"	7

6	Arithmetik und Datentypen in VHDL	8
6.1	ieee.numeric_std	8
6.1.1	Type Casting	8
6.1.2	Type Conversion	8
6.2	Diverse weitere Funktionen und Operatoren der 1164-Paket-Familie	8
7	Key Concept 5: Parametrisierbarkeit von Modellen	8
7.1	Beispiel: counter_generic	8
8	Beispiele	8
8.1	FSM asynchroner Reset	8
8.2	Clock-Generator	8

Achtung : Siehe auch *VHDL Quick Reference Card*!



1.1.1 Entscheidungskriterien

- Machbarkeit (nicht alle digitalen Schaltungen lassen sich mit allen Klassen von Bauteilen realisieren)
- Realisierungszeit, Iterationszeit
- Kosten (Fixkosten und stückzahlabhängige Kosten)
- etc.

1.1.2 Standardbauteile

- Komponenten mit fixer Funktion, welche in der Regel in grosser Stückzahl produziert werden. Meist realisieren Standardbauteile einfachere Funktionen.
- Beispiele: Digitale Gatter, Flip-Flops, PROM, EEPROM
- ASSP (Application Specific Standard Product) sind Standardbauteile die im Gegensatz zu den klassischen Standardbauteilen komplexere Funktionen realisieren. Z.B: NFC Controller, HD Reader, Video encoder / decoder.

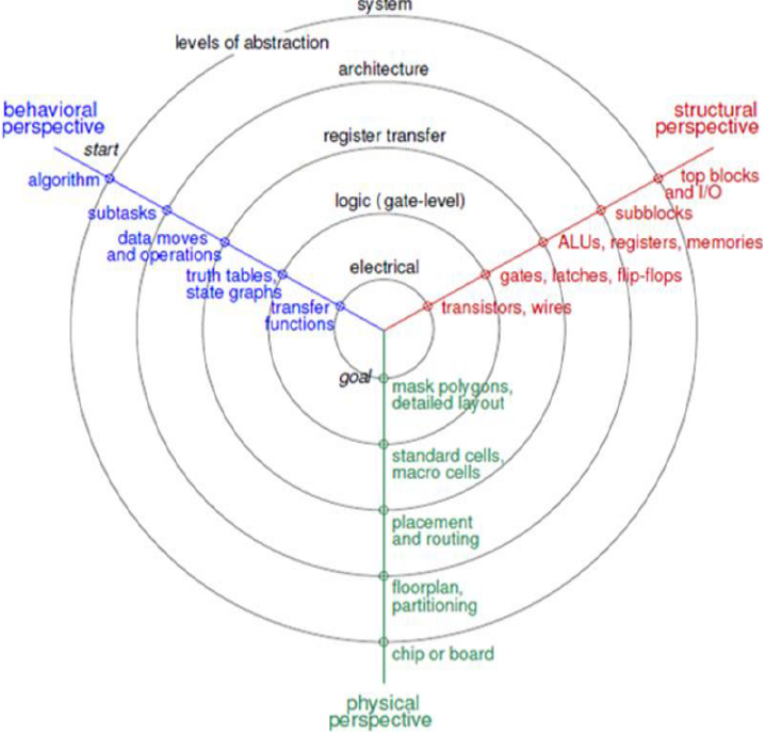
1.1.3 Programmierbare Bauteile

ROM	• Read Only Memory; Je nach Bauart wieder löschbar (EPROM, EEPROM) oder nicht löschbar (PROM, OTP). • Konjunktionen fix vorhanden, Disjunktionen programmierbar. Es kann damit für ein bestimmtes Eingangssignal ein bestimmtes Ausgangssignal programmiert werden. Jedem Eingangssignal ist ein fixes Ausgangssignal eingebrannt. • Werden in Logiksystemen häufig als Look-Up-Table (LUT) eingesetzt.
PLD	• Programmable Logic Device • Disjunktionen fix vorhanden, Konjunktionen programmierbar. Mit dieser Anordnung können logische Funktionen programmiert werden. Im Gegensatz zum ROM wird eine logische Funktion "eingebrannt" und nicht Verbindungen gesetzt, die einem Eingangssignal ein Ausgangssignal zuordnen. • Unter die Kategorie PLD fallen auch PAL und CPLD
FPGA	• Field Programmable Gate Array; Regelmässig angeordnete Logikblöcke und I/O-Blöcke die mittels Routingkanälen und programmierbaren Schaltmatrizen verbunden werden können • Zwei grundsätzlich unterschiedliche Technologien: SRAM-Zellen und Antifuses – <u>SRAM-Zellen</u>: Mehrfach programmierbar; Verliert ihren Inhalt, wenn Speisung wegfällt; Grösser als Antifuse-Zellen; Grössere Verzögerungszeiten – <u>Antifuse-Zellen</u>: Isolationsschicht, welche durchgebrannt wird; Nur einmal programmierbar; Sicherer bei harschen Umgebungsbedingungen (keine Speicherverluste, weil eingebrannt)

1.1.4 ASIC (Application Specific Integrated Circuit)

Semi-Custom-ASIC	Die Grundstruktur des Chips ist vorgegeben und es müssen nur noch die Metallmasken definiert werden. Dieses Verfahren hat den Vorteil, dass die Grundstruktur in sehr grosser Menge vorab produziert werden kann und dank den dabei erreichten Stückzahlen Kosten eingespart werden können. Auch sind nur sehr wenige kundenspezifische Masken notwendig, was sich wiederum in den NRE-Kosten (NRE = Non Recurring Engineering), also den von der Stückzahl unabhängigen Kosten bemerkbar macht. Der Nachteil liegt darin, dass die Grundstrukturen nur in gewissen Abstufungen erhältlich sind und somit eine optimale Nutzung der zur Verfügung stehenden Chip-Fläche kaum erreicht werden kann.
Full-Custom-ASIC	Chip der von Grund auf für eine bestimmte Anwendung entwickelt wird.

1.2 Digitaler Design Flow



- **Verhaltenssicht:** Es geht nur um das Verhalten (Black Box). Der Aufbau der Schaltung interessiert nicht.
- **Strukturelle Sicht:** Hier geht es um die elektronische Schaltung. Welche Baublöcke werden verwendet, wie sind sie verbunden.

- **Geometrisch-physikalische Sicht:** Wie werden die einzelnen physikalischen Elemente angeordnet, wie werden sie verbunden.

1.2.1 Synthese und Optimierung

Der erste Synthese- und Optimierungsschritt übersetzt die Designbeschreibung automatisch in eine generische Netzliste. Generisch ist diese Netzliste insofern, als sie noch nicht von der Zieltechnologie abhängig ist. Das Resultat dieser Synthese kann immer noch für jede Zieltechnologie, sei es ein FPGA oder ein ASIC, verwendet werden. Die Netzliste besteht aus allgemeinen logischen Gattern und Baublöcken. Wird auf dieser Abstraktionsebene eine Simulation durchgeführt, dann berücksichtigt sie keine physikalischen Eigenschaften der Zielhardware. Wir erhalten also immer noch eine rein funktionale Simulation ohne physikalisch begründetes Timing (Zeitverhalten) der Schaltung.

1.2.2 Technology Mapping

Den generischen Baublöcken werden spezifische, in der Zieltechnologie real existierende Baublöcke zugeordnet. Je nach Zieltechnologie wird sich nun die Netzliste drastisch unterscheiden. Das Timing der Schaltung ist hier aber immer noch nicht bekannt.

1.2.3 Place & Route

Bei der Umsetzung vom Schaltschema auf den Baustein (z.B. FPGA) spricht man von *Place & Route*.

1.2.4 Simulation

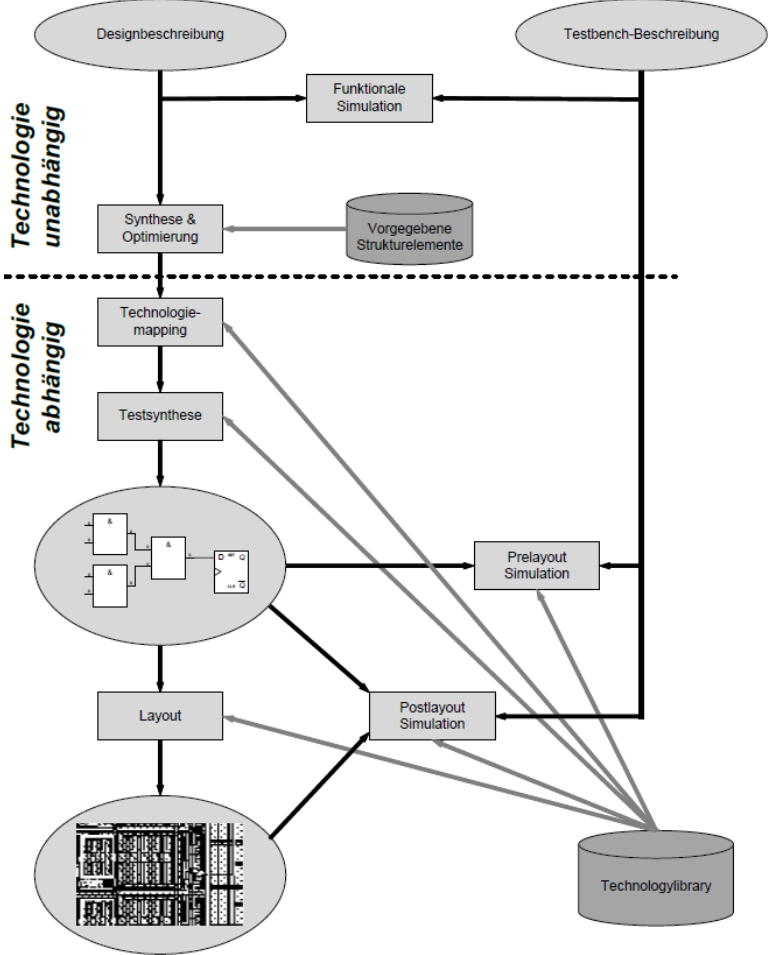
Die Test-Bench erzeugt Eingangssignale (Stimuli) für das DUT (Device Under Test) und überprüft die geforderten Ausgangssignale.

1.3 Hardware-Beschreibung mit VHDL

- VHDL (Very High Speed Integrated Circuit Hardware Description Language) wurde ursprünglich als Modellierungssprache für die Simulation entwickelt. Später kamen "Syntheseprogramme" dazu, die Hardwaresynthese möglich machten. Wichtig: nicht alle Konstrukte von VHDL sind synthesefähig!
- VHDL enthält alle Elemente einer klassischen Programmiersprache die aus Ada übernommen worden sind. Erweitert ist es um Konstrukte, die für den Schaltungsentwurf entscheidend sind. Dies sind:
 - Hierarchie und Konnektivität
 - Beschreibung paralleler Abläufe und der Interaktion paralleler Abläufe
 - Beschreibung elektrischer Signale
 - Beschreibung des zeitlichen Verhalten (ereignisbasiertes Zeitmodell)
 - Parametrisierbarkeit von Modellen
- VHDL ist **nicht** case-sensitive

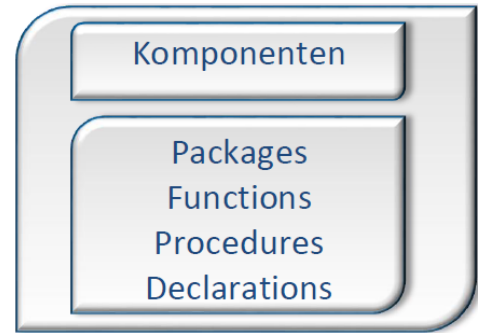
2 Key Concept 1: Hierarchie und Konnektivität

Hierarchie ist zentral, wenn es darum geht, Designs übersichtlich zu machen, Designs für ein Designteam organisierbar zu machen und Designs wiederverwenden zu können. VHDL unterstützt Modularität, hierarchische Konzepte und Wiederverwendbarkeit mit Sprachelementen. Zentral sind **Library**, **Entity** und **Architecture**.



Zeichen in Syntax-Beschreibungen:	
<>	Bezeichner
[]	Optionales Element. Kann weggelassen werden.
{ }	Optional, kann beliebig oft wiederholt werden.

2.1 Library



Element aus Library verwenden:

```
<variable_name> := <library_name>.<const_name>
```

Teile einer Library für das gesamte VHDL-Modul sichtbar machen:

```
library <library_name>;
use <library_name>.all;           --Ganzer Inhalt der Library
use <library_name>.<element_name>;--Spez. Element aus Library
use <library_name>.<package_name>.all;--Ganzes Package der L
use <library_name>.<package_name>.<element_name>; --Spez.
Element aus Package
```

2.1.1 Arten von Libraries

work	Diese Bibliothek ist immer vorhanden. Alle Blöcke, die wir im Design erzeugen, werden automatisch dort gespeichert.
std	Diese Bibliothek ist im IEEE Standard 1076 spezifiziert. Sie ist in jeder VHDL Designumgebung vorhanden. Sie enthält die wichtigsten Datentypen (BIT, INTEGER, BOOLEAN) und ein Package mit Funktionen für die Arbeit mit Files (package textio).
ieee	Diese Bibliothek enthält praktische Packages für die Verarbeitung von elektrischen Signalen (z.B. Datentypen std_logic und die darauf anwendbaren Funktionen).
Hersteller	Viele Hersteller von Hardware (PLD, ASIC) liefern eigene Bibliotheken mit, welche eine Beschreibung ihrer spezifischen Elemente enthalten. Zudem kann jeder Anwender eigene Bibliotheken mit Elementen erzeugen, die er zur Wiederverwendung zur Verfügung stellen will.

2.2 Entity

Entity beschreibt eine Designkomponente. Sie entspricht in dem Sinne genau einem Bauteil, das wir in einem Bauteilkatalog finden. Bei der Komponente kann es sich selbstverständlich sowohl um die Gesamtschaltung, als auch um eine Subkomponente handeln. Die Entity beschreibt zuerst einmal die Schnittstelle einer Entwurfseinheit nach aussen. Die Funktionalität wird gesondert in der Architektur beschrieben.

```
entity <entity_name> is
  port (
    {<port_name> : <mode> <type>;}
  );
end <entity_name>;

mode = in | out | inout | buffer
```

Info : Reale Hardwareblöcke weisen natürlich zusätzliche Pins für die Speisung auf. Speisungen wie VCC und GND werden in VHDL aber nicht angegeben, da sie nichts mit der digitalen Funktionalität zu tun haben.

2.3 Architecture

Die Architektur beschreibt den inneren Aufbau einer Entity. Die Beschreibung kann als **Verhaltens-** oder als **Strukturbeschreibungen** erfolgen. Pro Entity sind mehrere Architekturen möglich, zum Beispiel um verschiedene Abstraktionsniveaus zu beschreiben.

```
architecture <architecture_name> of <entity_name> is
  [type_declaration]
  [subtype_declaration]
  [constant_declaration]
  [signal_declaration]
  [component_declaration]
begin
  {architecture body: concurrent actions}
end [architecture_name];
```

Die Verhaltensbeschreibung wird im **Key Concept 2** behandelt.

2.3.1 Strukturbeschreibung

Beispiel "half adder":

```
entity half_adder is -- Entity declaration
  port(p, q : in bit;
        s : out bit;
        co : out bit);
end half_adder;

architecture structural of half_adder is
  -- component declaration.
  -- No signal declaration needed.
  component xor2 -- must be declared elsewhere
    port(in1, in2 : in bit;
          oup : out bit);
  end component;

  component and2 -- must be declared elsewhere
    port(in1, in2 : in bit;
          oup : out bit);
  end component;

begin
  -- instantiation of components XOR2 and AND2
  U1 : xor2 -- with explicite port map
    port map(in1 => p,
              in2 => q,
              oup => s);

  U2 : and2 -- with implicate port map
    port map(p, q, co);
end structural;
```

3 Key Concept 2: Nebenläufigkeit und Prozesse

3.1 Nebenläufige Signalzuweisung

Nebenläufige Signalzuweisung ist ein Spezialfall der nebenläufigen Prozesse, wo alle Eingangssignal in der Sensitivitätsliste stehen und nur nebenläufige Anweisungen erlaubt sind.

Signal-definition	Nebst den Eingangs- und Ausgangssignalen der Entity können in einer Architektur interne Signale definiert werden. Dies erfolgt vor dem begin. signal <signal_name> {,<signal_name>}: <type> [:= initial_value]	F.Dubacher 3
-------------------	---	----------------

Unbedingte Signalzuweisung	<pre>[<label>:] <signal_name> <= Expression {, Expression}</pre> Beispiele: y <= '1'; y <= a and b;
Selektive Signalzuweisung	Entspricht einer switch case Anweisung in prozeduralen Sprachen. Diese Art der Signalzuweisung wird typischerweise für Multiplexer verwendet. select_signal ist dabei das Steuersignal. <pre>[<label>:] with select_signal select dest_signal <= source_signal_1 when select_value_1, {source_signal_x when select_value_x,} {source_signal_y when others;}</pre>
Bedingte Signalzuweisung	Ist ähnlich wie if..then..else Anweisungen in prozeduralen Sprachen. Die einzelnen Bedingungen werden nacheinander abgearbeitet, bis eine Bedingung erfüllt ist. Dies kann je nach Synthesewerkzeug zu einer Schachtelung von Logikstufen und damit zu unterschiedlichen Signallaufzeiten führen. Dieser Nachteil steht dem Vorteil gegenüber, dass im Gegensatz zur oben beschriebenen selektiven Signalzuweisung nur die wirklich benötigten Bedingungen beschrieben werden müssen. <pre>[<label>:] dest_signal <= expr_1 when condition_1 else {expr_x when condition_x else} expr_y;</pre>
Aggregate	Um einzelne Signale zu einem Signalvektor zusammenzufügen kann ein Aggregat verwendet werden. <pre>vector <= (value_n, value_n-1, ..., value_0); vector <= (element_n=> value_n, ..., element_0 => value_0);</pre> Beispiele: vector_implicit <= ('0', '0', i3, i2); vector_explicite <= (1 => i3, 0 => i2, others => '0');

3.2 Nebenläufige Prozesse

Nebenläufige Prozesse sind gegen Aussen normale Hardwareblöcke. Innen erlauben Prozesse nebst nebenläufigen Abläufen auch sequentielle Anweisungen. Prozesse werden aktiviert, wenn ein Signal in ihrer Sensitivitätsliste ändert. Ausgangssignale ändern erst, wenn der Prozess abgeschlossen ist.

```
[<label>:] process[(<sensitivity list>)]
  <declarations>
begin
  {<sequential statements>}
end process;
```

Alle Signale in die Sens'liste	<pre>[<label>:] process (all) begin ...</pre>
Sequentielle Anweisungen	<pre>if..then..else: if condition_a then {sequential statements} elsif condition_b then {sequential statements} else {sequential statements} end if;</pre> <pre>case: case expression is when choice_a => {sequential statement} {when choice_b => {sequential statement}} when others => {sequential statement} end case;</pre> Achtung : Selektive und bedingte Signalzuweisungen sind innerhalb von Prozessen nicht erlaubt!
Eigenschaften nebenläufiger Prozesse	• Prozesse können zwei oder mehr Signale auf einmal zuweisen. • Das Verarbeiten der Information für die Signalzuweisung erfolgt in einer Sequenz von Befehlen, die einer nach dem anderen abgearbeitet werden (prozedural). • Ein- und Ausgangssignale bleiben während der internen Berechnung eines Prozesses unverändert. Ausgangssignale werden erst am Schluss zugewiesen. • Prozesse erlauben den Einsatz von Variablen für temporäre Speicherung von Signalwerten. • Durch den Einsatz von Sensitivitätslisten ist eine bessere Kontrolle über die Aktivierung gewährleistet.
Variablen in nebenläufigen Prozessen	Innerhalb von Prozessen können Variablen verwendet werden, um mit Zwischenwerten zu arbeiten. • Variablen werden im Teil declarations vom Prozess definiert. • Variablen müssen vor ihrer ersten Verwendung initialisiert werden, um zu verhindern, dass ein D-Latch generiert wird. variable <var_name> ,var_name: <type> [:= expression];
wait-Anweisung	<pre>wait until <condition> Hält den Prozess an, bis die Bedingung true ist. wait on <signals> Sobald ein Signal ändert, wird der Code weiter ausgeführt. wait for <time> Eine bestimmte Zeit lang warten.</pre> Achtung 1: wait geht nur ohne Sensitivitätsliste und äussere Signale werden bei wait aktualisiert. Achtung 2: wait for ist nicht synthetisierbar!

Achtung: Kombinatorische Schleifen und Latches

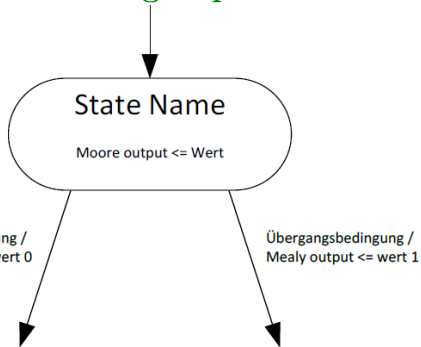
Kombinatorische Schleifen sind Signalführungen in kombinatorischen Prozessen, bei denen das rückgeführte Signal nur durch die Gatterlaufzeit der kombinatorischen Schaltung verzögert wieder an den Eingang derselben kombinatorischen Schaltung geführt wird. Dies ist **äusserst gefährlich**, denn es kann zu **instabilen** Schaltungen führen. Gatter können zu Schwingen beginnen.

```
p1 : process(a, temp) -- Deklaration mit Sens.liste
begin
    temp <= a xor temp; -- Zuweisung an lokales Signal
end process p1;
```

Um kombinatorische Schleifen und Latches zu vermeiden müssen folgende Regeln beachtet werden:

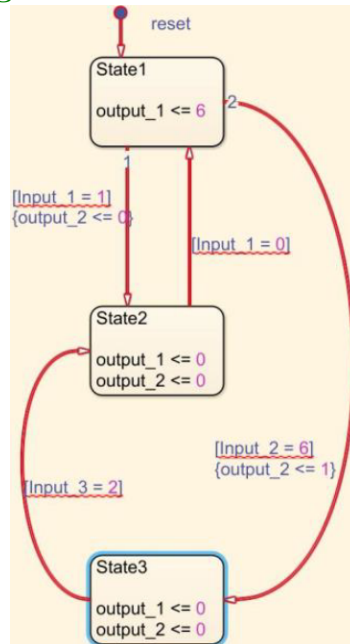
1. Ausgangssignale eines Prozesses dürfen nicht in der Sensitivitätsliste des gleichen Prozesses stehen.
2. In rein kombinatorischen Prozessen müssen alle Signale, die gelesen werden, in der Sensitivitätsliste aufgeführt werden.
3. Wertzuweisung an jedes Ausgangssignal in jedem Zweig und explizit einen **else** - Zweig einfügen. Oder noch sicherer: Zuweisung eines Default-Wertes an die Ausgangssignale vor der ersten **if**-Anweisung.
4. Zuweisung eines aktuellen Wertes an Variable vor ihrer ersten Verwendung.

3.3 Modellierung sequenzieller Schaltungen

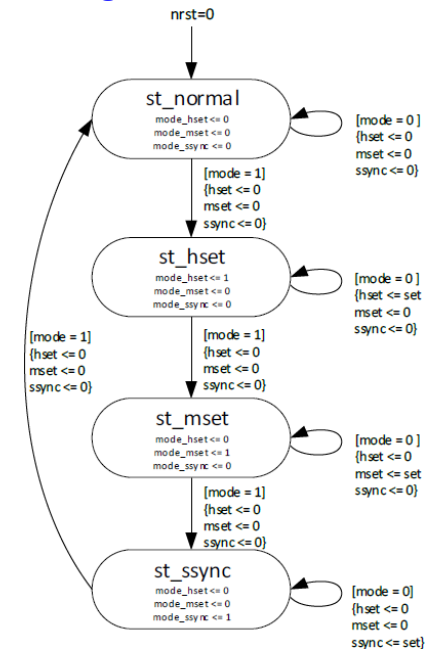


Von links:

1. Einzelner Zustand als Bubble mit Übergangsbedingungen und Übergangsausgängen
2. Allgemeines Bubble-Diagramm-Beispiel
3. Beispiel "Watch Controller"



3.3.1 Bubble Diagram



3.4 Endliche Zustandsautomaten mit VHDL (3 Prozess-Modell)

3.4.1 Allgemeine Struktur

```
library ieee;

entity fsm_prototype is
port(
    clk: in bit;
    rst: in bit;
    inp: in bit_vector(m-1 downto 0);
    oup: out bit_vector(n-1 downto 0)
);
end;

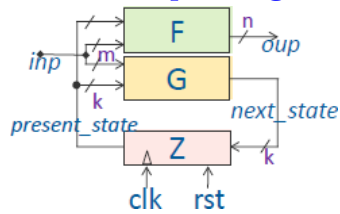
architecture behavioral of fsm_prototyp is
--type declaration (automatische State-Codierung)
type state_type is (st_reset, st_1, st_2, ...);
--registers
signal present_state, next_state: state_type;
begin
--Prozess F = Output Logic: Kombinatorischer
--Prozess für Ausgangsberechnung
    output_logic: process(...) ...

--Prozess G = Next_State Logic: Kombinatorischer
--Prozess für Zustandsaktualisierung
    next_state_logic: process(inp, present_state)
    begin
        --Default Folgezustand setzen
        next_state <= st_reset; -- / present_state
```

```
--Folgezustand berechnen
case present_state is
when st_1 =>
    if (inp = "01") then
        next_state <= st_1;
    else
        next_state <= st_2;
    end if;
when ... =>
    if (...) then
        ...
    end if;
when others => -- Error State
    null; -- Falls Default Zustand gesetzt
end case; -- wird kann null = no action
end process; -- verwendet werden
```

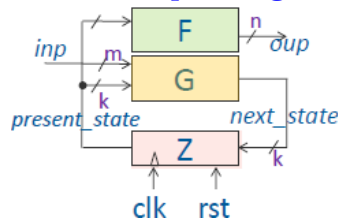
```
--Prozess Z = State Register: Sequenzieller
--Prozess für Zustandsaktualisierung
state_register: process(clk)
begin
    if clk'event and (clk = '1') then
        if (rst = '1') then --Synchroner Reset
            present_state <= st_reset;
        --Zustandswechsel bei steigender Clockflanke
        else --immer synchron zum Clock
            present_state <= next_state;
        end if;
    end if;
end process;
end behavioral;
```

3.4.2 Output Logic Mealy



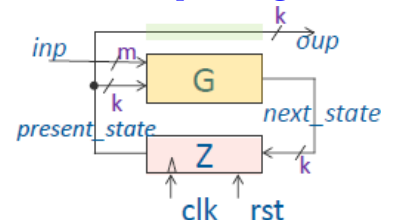
```
output_logic :
process (inp, present_state)
begin
  --Defaultwert für Ausgang
  oup <= "00";
  --Ausgang abhängig von
  --Zustand und Eingang
  if (present_state = st_1
    and inp = "10") then
    oup <= "10";
  elsif (present_state = ...
    and inp = ...) then
    oup <= ...;
  ...
end if;
end process;
```

3.4.3 Output Logic Moore



```
output_logic :
process (present_state)
begin
  --Defaultwert für Ausgang
  oup <= "00";
  --Ausgang abhängig von Zustand
  if (present_state = st_1) then
    oup <= "10";
  elsif (present_state = ...) then
    oup <= ...;
  ...
end if;
end process;
```

3.4.4 Output Logic Medw.



```
output_logic :
oup <= present_state
```

3.4.5 Manuelle State-Codierung

```
subtype state_type is bit_vector(2 downto 0);
signal present_state, next_state : state_type;
```

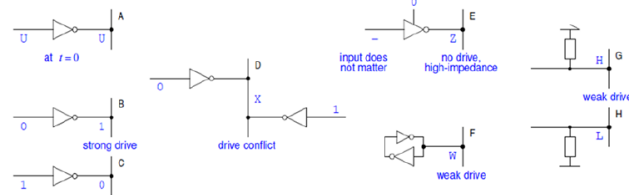
```
constant st_1 : state_type := "000";
constant st_2 : state_type := "010"; -- ...
```

4 Key Concept 3: Diskreter Ersatz für elektrische Signale

4.1 Das IEEE 1164 Logik System

Um elektrische Signale modellieren zu können, definiert der IEEE Standard 1164 ein neunwertiges Logiksystem. Dies wird benötigt, um z.B. Bussysteme zu beschreiben.

Logikwert		Logischer Zustand		
		low	unknown	high
uninitialized		U		
Treiberstärke	strong	0	X	1
	weak	L	W	H
	high impedance	Z	Z	Z
Don't care		-		



Logikwert	Bedeutung	Verwendung	Geeignet für Synthese
'U'	uninitialized	Signal ist im Simulator noch nicht initialisiert. In der Hardware ist das Signal unbekannt. Typisch für nicht initialisierte Speicherstelle nach Start-up.	nein
'X'	undefined	Signal ist dem Simulator unbekannt. Typisch für Treiberkonflikte, d.h. ein Signal wird von mehr als einer Quelle getrieben. In der Welt der Hardware kommt dies vor allem bei Buskonflikten vor.	nein
'0'	strong zero	Low Pegel eines Standardausganges. Regelfall für Hardware '0' Pegel.	ja
'1'	strong one	High Pegel eines Standardausganges. Regelfall für Hardware '1' Pegel.	ja
'Z'	high impedance	Signalausgang von Leitung entkoppelt. Typisch für Hochimpedanz-Ausgang eines Three-State Treibers bei bidirektionalen Bussen.	ja
'L'	weak zero	Low Pegel eines schwachen Treiberausgangs. Typisch für ein Signal, das mit einem pull-down Widerstand auf Low Potenzial gezogen wird.	nein
'H'	weak one	High Pegel eines schwachen Treiberausgangs. Typisch für ein Signal, das mit einem pull-up Widerstand auf High Potenzial gezogen wird.	nein
'W'	weak unknown	Simulator erkennt Treiberkonflikt. Dieser Fall ist analog zu 'X' mit dem Unterschied, dass der Treiberkonflikt hier durch schwache Treiber verursacht wird.	nein
'-'	don't care	Logikzustand des Ausgangssignals ist bedeutungslos. Existiert nur in der Schaltungssynthese und wird für Logikminimierung gebraucht.	nein

4.1.1 std_ulogic (_vector)

Löst ein Kompilationsfehler aus, wenn ein Treiberkonflikt besteht.

Vorteile von std_ulogic zu bit:

- Erlaubt Modellierung von Three-State Ausgängen durch den Signalwert 'Z'. Es darf allerdings kein zweiter Treiber auf das Signal wirken. Somit sind bidirektionale Busse mit std_ulogic nicht möglich.
- Erkennt nicht initialisierte Signale während der Simulation und bezeichnet sie mit dem Signalwert 'U'.
- Erlaubt Modellierung schwacher Treibersignale in der Simulation durch die Signalwerte 'H' und 'L'.
- Erlaubt don't care Modellierung durch den Signalwert '-'.

4.1.2 std_logic (_vector)

Behebt den Treiberkonflikt automatisch.

Vorteile und Nachteile von std_logic zu std_ulogic:

- Bidirektionale Busse können so simuliert werden, dass mehrere Treiber erlaubt sind. Treiberkonflikte werden automatisch vom Simulator aufgelöst.
- Gefahr von Mehrfachzuweisungen wird nicht vom Compiler, sondern erst in der Simulation – wenn überhaupt – erkannt.
- Simulationsaufwand wird höher, da immer Resolve – Funktion ausgewertet werden muss.
- **Achtung** : Es sollte immer std_ulogic bevorzugt werden! std_logic nur einsetzen, wenn unbedingt nötig (z.B. für Bidirektionale Busse).

4.2 Bidirektionale Busse in VHDL

```
architecture tri_state_out of busdriver is
begin
  write : process(enable, out_i)
  begin
    if enable = '1' then
      bus_i_o <= out_i; — Schreibvorgang
    else
      →
```

```
— Hochimpedanz
      bus_i_o <= (others => 'Z');
    end if;
  end process write;
  — paralleler Lesevorgang
  read : in_i <= bus_i_o;
end tri_state_out;
```

5 Key Concept 4: Ereignisbasiertes Konzept zur zeitlichen Darstellung bei Sim.

5.1 Modellierung des zeitlichen Verhaltens in VHDL 5.1.1 Event Queue

Im Gegensatz zu Simulatoren die zeitliche Abläufe mit Hilfe von Differentialgleichungen und linearer Algebra (Mod. Knotenpotentialmethode) simulieren, verarbeiten VHDL-Simulatoren Events in einer Event Queue. Events können wieder neue Events auslösen und die Simulation läuft so lange, bis alle Events (der eingestellten Simulationszeit) abgearbeitet sind. Diese Methode ist sehr viel recheneffizienter für den digitalen Anwendungszweck.

5.1.2 Delta Time Modell

Die Simulation einer kombinatorischen Schaltung zeigt auf Verhaltensebene keinerlei Verzögerung zwischen Ein- und Ausgang. Der Simulator muss trotzdem zwischen Ursache und Wirkung an den einzelnen Gattern unterscheiden können, damit die Simulation korrekt ablaufen kann. Zwischen den einzelnen Ereignissen wird in der Event Queue eine "Delta Time" eingefügt, das ist eine infinitesimale Verzögerung, die in der Waveform View aber wegen ihrer Winzigkeit gleich Null erscheint.

5.1.3 Transport Delay Modell

Modellierung der "Propagation Delay Time". Die Zeit von der Ursache bis zur Wirkung an einer Komponente (Eingang zu Ausgang).

```
B <= transport A after tp; tp ist vom Typ time
```

5.1.4 Inertial Time Modell

Modellierung der "Setup and Hold Time". Die Zeit die ein Signal anliegen muss, bis eine Reaktion ausgelöst wird. Mit diesem Modell sollen ganz kurze Glitches auf den Eingangssignalen ausgefiltert werden.

```
B <= inertial A after tp oder nur B <= A after tp
```

5.2 Simulationsaufbau

Die Überprüfung einer Bedingung erfolgt mit assert <condition> report <message> severity <note|warning|error|failure> Eigenschaften einer Test-Bench:

- 1. Muss nicht synthesefähig sein → ganzer Sprachumfang von VHDL steht tzur Verfügung.
- 2. Test-Bench entity hat (in der Regel) keine Schnittstellen nach aussen.
- 3. Ein Testvektor beschreibt einen einzigen Vektor von Eingangssignalen, die als Stimulus ans DUT (Device Under Test) zum Zeitpunkt t angelegt werden. Die Response beschreibt die Systemantwort darauf.

5.2.1 Beispiel einer Test-Bench für die Komponente "xor2"

```
— library declarations
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

— entity declarations
entity exor_tb is
end;

— architecture declarations
architecture tb of exor_tb is

— constant declarations
constant sim_cyc : time := 10 ns;

— signal declarations
signal tb_a, tb_b : std_ulogic;
signal tb_y : std_ulogic;

— component declarations
component xor2
  port( xor2_a, xor2_b : in std_ulogic;
        xor2_y       : out std_ulogic );
end component;

— configuration
for all : xor2 use entity work.xor2 (behavioral);
begin

— instance assignment
dut : xor2
port map(
→
```

```
xor2_a => tb_a,
xor2_b => tb_b,
xor2_y => tb_y );

— signal assignments for stimuli
Stimuli : process
variable tb_in_vec :
std_ulogic_vector(2 - 1 downto 0);
begin
  for i in 0 to (2 ** 2) - 1 loop
    tb_in_vec :=
std_ulogic_vector(to_unsigned(i, 2));
    tb_a <= tb_in_vec(1);
    tb_b <= tb_in_vec(0);
    wait for sim_cyc;
  end loop;
  wait;
end process;

— evaluation of responses
Response : process
begin
  wait for (sim_cyc - 1 ns);
  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; ...
end process;
end tb;
```

6 **Arithmetik und Datentypen in VHDL**

Für die Typen `bit_vector` und `std_logic_vector` sind nur bit-weise Operationen definiert. Eine Bitfolge alleine erklärt den Zahlenwert aber nicht. Um auf Signalen von den vorher genannten Typen arithematische Operationen ausführen zu können, gibt es das Paket `ieee.numeric_std`.

6.1 **ieee.numeric_std**

Das Paket führt zwei neue Datentypen ein (auf Basis von `std_(u)logic_vector`): `(u_)signed` und `(u_)unsigned`. **Achtung** : Division ist nicht synthesesfähig, bei Potenzen nur mit Basis 2 (Schiebeoperation) und Modulo nur mit Divisor 2.

6.1.1 **Type Casting**

target_type (T)	source_type (S)	Type Cast Operator	Beispiel
(u_)unsigned	std_(u)logic_vector	(unresolved_)unsigned (S)	T <= unsigned(S)
(u_)signed	std_(u)logic_vector	(unresolved_)signed (S)	T <= signed(S)
std_(u)logic_vector	(u_)unsigned	std_(u)logic_vector(S)	T <= std_(u)logic_vector(S)
std_(u)logic_vector	(u_)signed	std_(u)logic_vector(S)	T <= std_(u)logic_vector(S)

6.1.2 **Type Conversion**

target_type (T)	source_type (S)	Type Conversion Operator	Beispiel
integer	signed	to_integer (S)	T <= to_integer(S)
integer	unsigned	to_integer (S)	T <= to_integer(S)
signed	integer	to_signed(S, length)	T <= to_signed(S,length)
unsigned	integer	to_unsigned(S, length)	T <= to_unsigned(S,length)

6.2 **Diverse weitere Funktionen und Operatoren der 1164-Paket-Familie**

→ Siehe Quick-Ref-Card Seite 3

7 **Key Concept 5: Parametrisierbarkeit von Modellen**

<code>generic param_name : param_type := initial_value</code>	(wird zur Compile-Zeit festgelegt)
---	------------------------------------

7.1 **Beispiel: counter_generic**

<pre>entity counter_generic is generic (CNT_MAX : integer := 128); port(clk, rst, ena : in std_ulogic; count : out integer range 0 to CNT_MAX - 1); end counter_generic; architecture structural of counter_array is constant CNT_MAX0 : integer := 14; constant CNT_MAX1 : integer := 16; signal clk, rst, ena : std_ulogic; signal count0 : integer range 0 to CNT_MAX0 - 1; signal count1 : integer range 0 to CNT_MAX1 - 1;</pre>	<pre>component counter_generic generic(CNT_MAX : integer := 127); port(clk, rst, ena : in std_ulogic; count : out integer range 0 to CNT_MAX - 1); end component; begin cnt0 : counter_generic generic map(CNT_MAX => CNT_MAX0) port map(clk => clk, rst => rst, ena => ena, count => count0); cnt1 : ... end structural;</pre>
--	--

8 **Beispiele** 8.1 **FSM asynchroner Reset**

```
state_register : process(clk, rst)
begin
  if (rst = '1') then
    present_state <= reset_state;
  elsif (clk'event and clk = '1') then
    present_state <= next_state;
  end if;
end process;
```

8.2 **Clock-Generator**

```
clock_generator : process
begin
  tb_clk <= '0';
  wait for sim_cyc/2;
  tb_clk <= '1';
  wait for sim_cyc/2;
end process;
```