



V 1.0.20250217

Inhaltsverzeichnis

1	Realisierungsformen	1
1.1	Programmierbare Bauteile	1
2	Digitaler Design-Flow	1
2.1	Y-Modell von Gajski	1
2.2	Designprozess	1
2.3	Simulation	1
3	Hierarchie & Konnektivität	1
3.1	Grundelemente	1
3.2	Strukturbeschreibung	2
4	Nebenläufige Prozesse & Prozessinteraktionen	2
4.1	Nebenläufige Signalzuweisungen	2
4.2	Nebenläufige Prozesse	3
4.3	Modellierung sequenzieller Schaltungen	3

4.4	Finite State Machines (FSM)	3
4.5	Modellierung von Registern für FPGAs	4
5	Diskreter Ersatz für elektrische Signale	4
5.1	IEEE 1164 Logik System	4
6	Arithmetik in VHDL	5
6.1	ieee.numeric_std	5
6.2	Datentypkonversion	5
7	Simulation / Test	5
7.1	Hierarchisches Vorgehen beim Design	5
7.2	Design vs. Test	5
7.3	Event-based concept of time	5
7.4	Concurrency	6
7.5	Ablaufsteuerung einer Simulation	6
7.6	Modellierung von Verzögerungszeiten	6
7.7	Testbenches	6
8	Modellparametrisierung	7
8.1	Zähler	7
8.2	Parametrisierung	7
9	Anhang	8
9.1	Übersicht Realisierungsformen	8
9.2	Kriterien zur Auswahl der Realisierungsform	8

1 Realisierungsformen

1.1 Programmierbare Bauteile

1.1.1 Read Only Memory (ROM)

Konjunktionen fix, Disjunktionen programmierbar. In Logikstrukturen meist als LUT.

Varianten:

- | | |
|--|---|
| <ul style="list-style-type: none">• PROM (Programmable ROM)<ul style="list-style-type: none">– Einmalig programmierbar– aka OTP (One Time Programmable)• Flash<ul style="list-style-type: none">– Spez. EEPROM mit gr. Speicherdichte– Löschen und schreiben nur in Blöcken | <ul style="list-style-type: none">• EPROM (Erasable PROM)<ul style="list-style-type: none">– Programmier- und löschbar– Löschen durch UV-Licht• EEPROM (Electrically EPROM)<ul style="list-style-type: none">– Programmier- und löschbar– Löschen durch Spannung |
|--|---|

1.1.2 Programmable Logic Device (PLD)

Konjunktionen (AND) programmierbar, Disjunktionen (OR) fix.

Varianten:

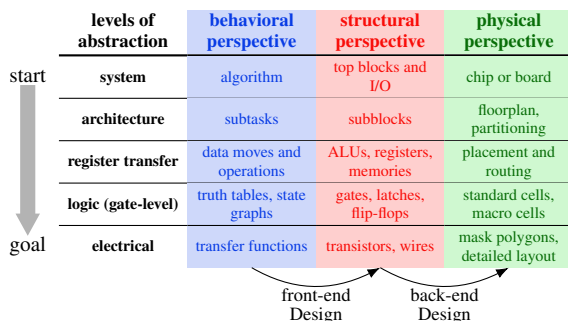
- | | |
|--|--|
| <ul style="list-style-type: none">• PAL (Programmable Array Logic)• Erw. PLD: PLA (Programmable Logic Array)<ul style="list-style-type: none">– Kon- und Disjunktionen programmierbar | <ul style="list-style-type: none">• CPLD (Complex PLD)• GAL (Generic Array Logic) |
|--|--|

1.1.3 Field Programmable Gate Array (FPGA)

- Andere Grundstruktur als PLD, ähnliche Funktionalität
- Besteht aus regelmässig angeordneten Logikblöcken und I/O-Blöcken
 - Benennungen können variieren (z.B. CLB und IOB bei AMD Xilinx)
- Blöcke über Routing-Kanäle und programmierbare Schaltmatrizen verbunden
- Logikblöcke enth. Flip-Flops & Teile zur Realisierung kombinatorischer Funkt.
- SRAM basierte FPGAs verlieren Konfiguration bei power-off

2 Digitaler Design-Flow

2.1 Y-Modell von Gajski



2.2 Designprozess

- Startet auf Systemebene, meist bei **Verhaltenssicht**
- Folgende Fragen müssen beantwortet werden:
 - Welches Verhalten aus Sicht des Benutzers soll das System aufweisen?
 - Welche Schnittstellen hat das System und wie sehen diese aus?
 - Welche Randbedingungen sind zu beachten? (Grösse, Stromverbrauch, etc.)
 - Wie soll das System realisiert werden? (HW, SW, oder beides)
 - Gibt es bereits existierende Lösungen oder muss alles neu entwickelt werden?
- Falls HW nötig ist:
 - **Verhaltenssicht** verfeinern, in einzelne Algorithmen zerlegen und in strukturelle Ebene übersetzen (Front-End-Design)
 - Prozess verläuft hierarchisch, top-down
 - Übergang in geometrische Ebene: Back-End-Design

2.2.1 Designbeschreibung (Technologie-unabhängig)

- Mittels VHDL (oder Verilog)
 - Meist textuell. Auch grafische Beschreibung möglich (Blockdiagramme)

2.2.2 Synthese und Optimierung (Technologie-unabhängig)

- Designbeschreibung $\Rightarrow$ generische Netzliste
 - Rein funktionale Beschreibung
 - Nicht von Zieltechnologie abhängig

2.2.3 Technology Mapping

- Ordnet generische Baublöcke auf reale Baublöcke der Zieltechnologie zu
- Timing noch immer nicht bekannt

2.2.4 Place & Route

- Möglich mittels FPGA (sogenanntes Place & Route)
- Möglich mittels ASIC
 - Standardzellen-Design: Grundzellen auf Chip verteilt
 - Lohnt sich erst ab hohen Stückzahlen
 - Umsetzung Schema $\Rightarrow$ Chip: Layout
 - Parallel zwingend Verifikationsprozess nötig (durch Simulation)

2.3 Simulation

- Konkrete Funktionalität des Designs überprüfen

2.3.1 Simulation mittels Testbench & DUT

Siehe 7.7 Testbenches (S. 6)

3 Hierarchie & Konnektivität

Syntax Info:

- < > Platzhalter für Bezeichner (Identifier)
- [] Optional, kann weggelassen werden
- { } Optional, kann beliebig oft wiederholt werden

Regeln Identifier:

- A..Z, a..z, 0..9, _
- Start mit Buchstaben
- _ nicht am Ende oder doppelt (d.h. in Serie)
- caseinsensitive
- Keine Schlüsselwörter (**in**, **out**, ...)

3.1 Grundelemente

3.1.1 Library

- Wiederverwendbare Code-Blöcke in kompilierter Form
- Kann Komponenten und/oder Packages enthalten
 - **Komponenten:** Beschreibung von konkreten Schaltungen
 - **Packages:** Sammlung von Funktionen, Prozeduren & Deklarationen
- Standardbibliotheken:
 - work: Immer vorhanden, enthält selbst erstellte Blöcke
 - std: Spezifiziert in IEEE 1076. Enthält wichtigste Datentypen
 - ieee: Enthält praktische Packages für Signalverarbeitung

Deklaration: **library** <library_name>; (std & work standardmässig verfügbar)

Verwendung:

- | | |
|------------------|--|
| Einzel: | <code><variable_name> := <library_name>.<const_name>;</code> |
| Alle: | <code>use <library_name>.all;</code> |
| Lib-Element: | <code>use <library_name>.<elem_name>;</code> |
| Package: | <code>use <library_name>.<package_name>.all;</code> |
| Package-Element: | <code>use <library_name>.<package_name>.<elem_name>;</code> |

3.1.2 Entity

Beschreibt Schnittstelle einer Schaltung.

Syntax:

```
1 entity <entity_name> is
2   port (
3     {<port_name> : <mode> <type>;}
4   );
5 end <entity_name>;
```

Mögliche Modi:

- in** Eingang (aus Sicht d. Entity); Nur auf rechter Seite von Signalzuweisung
- out** Ausgang; Nur auf linker Seite einer Signalzuweisung
- inout** Bidirektionales Signal. Für bidirektionale Bus-Signale
- buffer** Ausgang; Darf auch auf rechter Seite stehen. **Nicht empfohlen**

(Einige) mögliche Typen:

		Spezieller Syntax
bit	'1' oder '0'	–
bit_vector	1D Array von Bits	bit_vector (3 downto 0)
boolean	true oder false	–
integer	Ganzzahl (Standard: 32 bit)	integer range 0 to 255

3.1.3 Architecture

- Beschreibt innerer Aufbau eines Moduls.
- Verhaltens**- oder **Struktur**beschreibung

Aufbau: **architecture** <architecture_name> **of** <entity_name> **is**

- Bezeichnung <architecture_name> frei wählbar
- Häufig verwendete Bezeichnungen:
 - behavioral: **Verhaltensbeschreibung**
 - structural: **Strukturbeschreibung**
 - rtl: Register Transfer Level
 - dataflow: Netzliste mit boolschen Operatoren
 - tb: Testbench

Syntax:

```
1 architecture <architecture_name> of <entity_name> is
2   [type_declaration]
3   [subtype_declaration]
4   [constant_declaration]
5   [signal_declaration]
6   [component_declaration]
7 begin
8   {architecture body: concurrent actions}
9 end [architecture_name];
```

Behavioral:

- Häufigster Modellierungsstil
- Beschreibung des Schaltungsverhaltens
- Ähnlich zu Sprachelementen gebräuchlicher Programmiersprachen
 - Verwendung von Prozeduren, Funktionen, if-then-else, case, ...

Structural:

- Schaltschema in Textform beschrieben
 - In einigen CADs auch grafisch möglich
- Ausgänge von Komponenten werden miteinander verbunden

3.2 Strukturbeschreibung

3.2.1 Deklaration von Komponenten und Signalen

Alle verwendeten Komponenten (Verwendung einer **entity** innerh. **architecture**) müssen deklariert werden nach folgendem Muster:

Komponenten:

```
1 component <component_name>
2   port (
3     {<port_name> : <mode> <type>;}
4   );
5 end component;
```

Für mögliche Modi siehe 3.1.2 Entity.

Signale: **signal** <internal_signal_name> : <type>;

3.2.2 Instanziierung von Komponenten

```
1 <instance_name> : <component_name>...
```

Implizite Zuordnung:

```
2 ... port map (
3   <external_signal_name> {,
4   <external_signal_name>});
```

Explizite Zuordnung:

```
2 ... port map (
3   <instance_port_name> => <external_signal_name> {,
4   <instance_port_name> => <external_signal_name>});
```

Beispiel: Strukturelle Architektur

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity top_component is
5   port(
6     in_top  : in std_logic;
7     out_top : out std_ulogic);
8 end entity top_component;
9
10 architecture strucural of top_component is
11   signal int_sig : std_ulogic;
12   -- declaration of component 'sub_component'
13   component sub_component
14     port(
15       inp : in std_logic;
16       oup : out std_ulogic);
17 end component;
18 begin
19   -- two instances of 'sub_component' named 'sub1' and 'sub2'
20   sub1 : component sub_component
21     port map(
22       inp => in_top,
23       oup => int_sig);
24
25   sub2 : component sub_component
26     port map(
27       inp => int_sig,
28       oup => out_top);
29 end architecture structural
```

4 Nebenläufige Prozesse & Prozessinteraktionen

4.1 Nebenläufige Signalzuweisungen

Initialisierung eines Signals bei der Deklaration ist für Synthese **nicht** erlaubt. Verwendung von Initialwerten vorwiegend nur in Testbenches.

4.1.1 Unbedingte Signalzuweisung

- Für einfache Kombinatorische Logik
- Fixe Verdrahtung von Hardware ohne Verzweigungen

Syntax: [<label>:] <signal> <= expression {, expression};

Mögliche expressions:

- Bit Konstante: '0', '1'
- Bit-Vektor Konstante: "1010" etc.
- Logische Ausdrücke: <signal_a> **and**|**or**|**xor**|**not** <signal_b>

4.1.2 Bedingte Signalzuweisung

- Ähnlich wie if-then-else in prozeduralen Sprachen
- Einzelne Bedingungen werden **nacheinander** abgearbeitet
- Kann zu Schachtelung von Logikstufen führen
 - Signallaufzeiten können daher stark variieren
- Ist keine Bedingung **true**, behält Signal seinen Wert

Syntax:

```
1 [<label>:] <dest_signal> <= expression_1 when condition_1 else
2     {expression_x when condition_x else}
3     expression_n;
```

4.1.3 Selektive Signalzuweisung

- Zuweisung abhängig von Bedingung
- Bedingungen **parallel** ausgeführt
- Alle möglichen Bedingungen müssen angegeben werden

Syntax:

```
1 [<label>:] with <select_signal> select <dest_signal> <=
2   <source_signal_1> when <select_value_1>,
3   {<source_signal_x> when <select_value_x>, ...}
4   {<source_signal_n> when others};
```

4.1.4 Aggregate

- Klammerausdruck, der mehrere Signale zu einem Vektor zusammenfasst

Implizit: (Reihenfolge muss zwingend eingehalten werden)

```
1 <vector> <= (value_n, ..., value_0);
```

Explizit: (Reihenfolge kann beliebig sein)

```
1 <vector> <= (elem_n => value_n, ..., elem_0 => value_0);
```

- value_x**: Signal, Konstante oder Ausdruck; **kein** Vektor
- elem_x**: Elementname/-Index des Vektors

Füllen mit Fixwert: <vector> <= (**others** => <value>);

4.1.5 Concatenate

```
1 <vector_3> <= <vector_1> & <vector_2>;
```

- Werden hintereinander aneinandergereiht

Do's and Don'ts:

```
1 <vector> <= "1010";
2 <vector> <= "0011";
3 <vector_implicit_nogo> <= ("000000", x1, x0);
4 <vector_explicit_nogo> <= (1 => x1, 0 => x0, others => '0');
5 <vector_concat> <= ("000000" & x1 & x0);
```

4.2 Nebenläufige Prozesse

4.2.1 Eigenschaften

- Alle Prozesse innerhalb einer Architektur nebenläufig
- Jeder Prozess kann als individueller HW-Block betrachtet werden
- Prozesse legen Signalwerte fest
 - Kommunikation geschieht über Signale (Lokale- & Port-Signale)
- Wird aktiviert, wenn Signal aus Sensitivityliste sich ändert
- Beliebige Ein- und Ausgänge
- In sequenziellem Anweisungsteil **nur** unbedingte Signalzuweisungen und spezielle sequenzielle Anweisungen
- Zuweisung eines Signalwerts erfolgt erst am Ende des Prozesses

Syntax:

```
1 [<label>:] process [<sensitivity_list>]
2 {declarations}
3 begin
4 {sequential_statements}
5 end process [<label>];
```

[<label>:] / [<label>] Prozessname (Optional)
[<sensitivity_list>] Änderungen dieser Signale aktivieren den Prozess
{declarations} Deklarationsteil für Konstanten, lokale Signale, Variablen, Typen etc.
{sequential_statements} Sequenzieller Anweisungsteil (Variablen- & Signalzuweisungen, Kontrollstrukturen)

4.2.2 Sequenzielle Anweisungen

Bedingte Signalzuweisung ⇒ if-then-else-Struktur:

```
1 if condition_a then
2 {sequential_statements}
3 elsif condition_b then
4 {sequential_statements}
5 else
6 {sequential_statements}
7 end if;
```

- Aufzählung der Bedingungen muss nicht abschliessend sein
 - (Ist jedoch empfehlenswert für Synthese)

Selektive Signalzuweisung ⇒ case-Struktur:

```
1 case expression is
2 when choice_a => {sequential_statement}
3 {when choice_b => {sequential_statement}}
4 when others => {sequential_statement}
5 end case;
```

- case-Struktur ist **immer** abschliessend
- Alle werte der **expression** müssen abgedeckt sein
 - Kann mit **others** => {...} abgefangen werden

4.2.3 Variablen in nebenläufigen Prozessen

Syntax:

```
1 variable <name_var_1> {, <name_var_n>}: <type> [:= expression];
2 <name_var> := expression; -- Zuweisung
```

Benötigt, weil Signalzuweisung erst am Ende des Prozesses erfolgt.

4.2.4 Kombinatorische Schleifen

- Rückführung eines Ausgangssignals einer komb. Schaltung auf sich selbst
 - Schaltung wird instabil, schwingt mit Frequenz abh. von Propagation-delay
 - Ausgangssignale dürfen nicht in der eigenen Sensitivityliste stehen
 - Synthese warnt, funktioniert jedoch

4.2.5 Latches

- Können Zustände speichern
- Eher nicht zu verwenden in taktsynchronen Systemen

Unerwünschte Latches & Massnahmen:

- Timing meist schwierig zu erfüllen
- In rein komb. Prozessen alle gelesene Signale in Sensitivitätsliste
- Keine Ausgangssignale in Sensitivitätsliste
- Zuweisung eines aktuellen Wertes an Variablen vor der ersten Verwendung
- Wertzuweisung an jedes Ausgangssignal in jedem Zweig und explizit einen **else** Zweig einfügen
 - Alternativ (noch sicherer): Default-Wert an Ausgangssignal zuweisen vor erster **if** Anweisung

4.3 Modellierung sequenzieller Schaltungen

4.3.1 Unterschiede Kombinatorisch ↔ Sequenziell

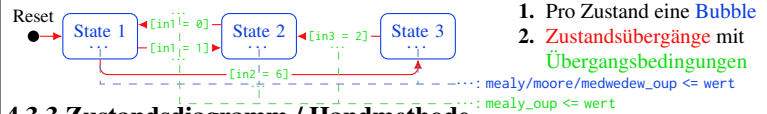
Kombinatorisch:

- Zustandsfrei
- Ausgänge funktional von Eingängen abhängig

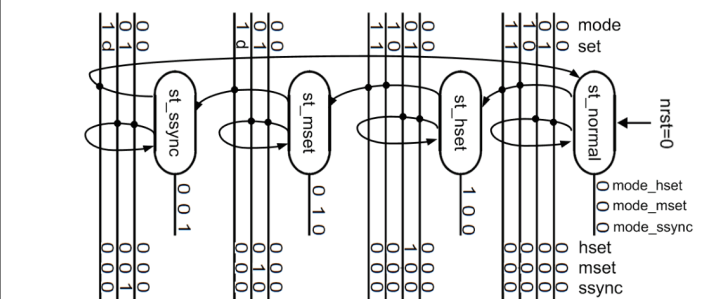
Sequenziell:

- Zustandsbehaftet
- Vergangenes Verhalten & Eingänge bestimmen momentanen Zustand & Ausgänge
- Besitzen Speicher um aktuellen Zustand zu merken

4.3.2 Bubble Diagramm



4.3.3 Zustandsdiagramm / Handmethode



4.3.4 Taktgesteuerte Schaltung

Flanke	bit	std_logic / std_ulogic
Positiv ⌈	<signal>'event and <signal> = '1'	rising_edge(<signal>)
Negativ ⌋	<signal>'event and <signal> = '0'	falling_edge(<signal>)

Wichtig: **bit**-Syntax funktioniert auch mit **std_(u)logic**, jedoch **nicht** umgekehrt. Beim **bit**-Syntax wird für **std_(u)logic** jede Flanke erkannt, nicht nur von 0 auf 1 oder umgekehrt.

4.3.5 D-Flip-Flop

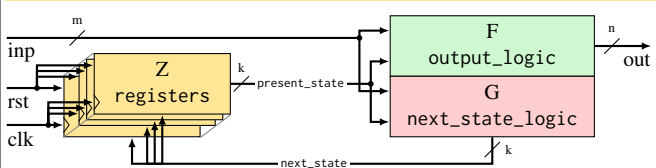
```
1 reg: process(clk)
2 begin
3   if rising_edge(clk) then
4     if reset = '1' then
5       q <= '0';
6     elsif ena = '1' then
7       q <= d;
8     end if;
9   end if;
10 end process;
```

- In Sensitivityliste generell nur **clk** (und **rst** bei asynchronem Reset) aufnehmen
- **Kein else**-Block, da Speicherverhalten gewünscht ist!
- Auch **enable** meist nur synchron (nicht in Sensitivityliste)

4.3.6 Regeln

- Strikte Trennung der Register vom kombinatorischen Teil
- Signalzuweisungen in taktsynchronem Umfeld werden zu D-FF synthetisiert
- **elsif/if**-Anweisung, die Takt definiert muss **letzte** Anweisung sein
- **if**-Anweisung muss nicht abschliessend sein, um Latches zu vermeiden

4.4 Finite State Machines (FSM)



4.4.1 Zustandsdarstellung

```
1 type <state_type> is (s0, s1, ..., sn);
2 signal <present_state>, <next_state>: <state_type>;
```

- Tool initialisiert **s0, s1, ..., sn** aufsteigend von links nach rechts
- Bei expliziter Vergabe der Codierung **muss** in Synthesis Settings **fsm_extraction = off** oder im Code Attribut **fsm_encoding** gesetzt werden

Enum Encoding:

```
1 type <state_type> is (s0, s1, ..., sn);
2 attribute <enum_encoding> : string;
3 attribute <enum_encoding> of <state_type>:
4 type is "0001 0010 ...";
5 signal <present_state>, <next_state> : <state_type>;
```

Konstanten Encoding:

```
1 subtype <state_type> is bit_vector(3 downto 0);
2 constant s0: <state_type> := "0001";
3 constant s1: <state_type> := "0010";
4 ...
5 constant sn: <state_type> := "1000";
6 signal <present_state>, <next_state> : <state_type>;
```

4.4.2 3-Prozess Darstellung

Prozess Z (register):

```
1 registers: process(rst, clk)
2 begin
3   if rst = '1' then -- Asynchroner Reset
4     present_state <= st_reset;
5   elsif clk'event and clk = '1' then -- Synchroner Zustandswechsel
6     present_state <= next_state;
7   end if;
8 end process;
```

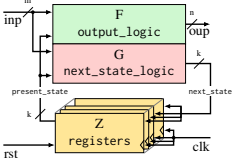
Prozess G (next_state_logic):

```
1 next_state_logic: process(present_state, inp)
2 begin
3   next_state <= st_reset; -- Default value
4   case present_state is
5     when s0 =>
6       if (Bedingung 1 als Fkt. von inp) then
7         next_state <= s1;
8       end if;
9     when s1 =>
10      if (Bedingung 2 als Fkt. von inp) then
11        next_state <= s2;
12      end if;
13     when ...
14     when others => null;
15   end case;
16 end process;
```

4.4.3 Ausgabeberechnung

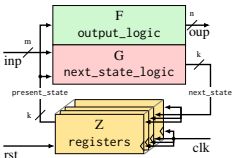
Mealy Automat: Ausgänge abhängig von aktuellem Zustand & Eingängen

```
1 output_logic: process(present_state, inp)
2 begin
3   oup <= '0'; -- Default Wert
4   if(present_state = s0) and (inp = XYZ) then
5     oup <= ...; -- Abh. von Zustand und inp
6   elsif ...
7   end if;
8 end process;
```



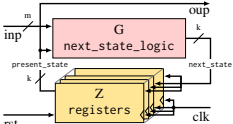
Moore Automat: Ausgänge nur Funktion des aktuellen Zustands

```
1 output_logic: process(present_state)
2 begin
3   oup <= '0'; -- Default Wert
4   if(present_state = s0) then
5     oup <= ...; -- Ausgang abh. von Zustand
6   elsif ...
7   end if;
8 end process;
```



Medwedew Automat: Ausgang entspricht Zustand

```
1 output_logic: oup <= present_state;
```



4.4.4 Huffman Modell – 2-Prozess Darstellung

```
1 entity huffman_fsm is
2   port(rst, clk: in bit;
3         inp: in bit;
4         oup: out bit);
5 end huffman_fsm;
6
7 architecture fsm of huffman_fsm is
8   type state_type is (s0,s1,...);
9   signal st_present : state_type;
10  signal st_next : state_type;
11 begin
12  comb_logic: process(
13    st_present, inp)
14  begin
15    -- default values
16    st_next <= s0;
17    oup <= '0';
18    case st_present is
19      when s0 =>
20        st_next <= s1;
21        oup <= '1';
22      when others =>
23        null;
24      end case;
25    end process;
26  registers: process(clk)
27  begin
28    if rising_edge(clk) then
29      if rst = '1' then
30        st_present <= s0;
31      else
32        st_present <= st_next;
33      end if;
34    end if;
35  end process;
36 end architecture fsm;
```

4.5 Modellierung von Registern für FPGAs

- Modellierung mittels D-Flip-Flop mit Chip Select / Chip Enable (CS / CE)
- **Wichtig:** rst muss synchron sein.
- Alle Eingänge high-aktiv.
- VHDL Umsetzung, Siehe D-Flip-Flop 4.3.5

5 Diskreter Ersatz für elektrische Signale

5.1 IEEE 1164 Logik System

5.1.1 Darstellung elektrischer Signale

	Logikwert	Logischer Zustand		
		Low	Unknown	High
Treiberstärke	Strong	0	X	1
	Weak	L	W	H
	High-Impedance	Z	Z	Z

5.1.2 9-Wertiges Logik System

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

5.1.3 Auflösungsfunktion (std_logic_1164)

Resolve	Treiber A (Quelle)								
	U	X	0	1	Z	W	L	H	-
Treiber B (Quelle)	U	U	U	U	U	U	U	U	U
	X	U	X	X	X	X	X	X	X
	0	U	X	0	X	0	0	0	X
	1	U	X	X	1	1	1	1	X
	Z	U	X	0	1	Z	W	L	H
	W	U	X	0	1	W	W	W	X
	L	U	X	0	1	L	W	L	X
	H	U	X	0	1	H	W	W	X
	-	U	X	X	X	X	X	X	X

5.1.4 std_logic

Akzeptiert mehrere Treiber pro Signal. (Bidirektionale Busse)

Nachteile:

- Gefahr der Mehrfachzuweisung (Kurzschluss in HW) wird nicht vom Compiler erkannt.
- Detektion nur bei Treiberkonflikt in Simulation möglich.
- Simulationsaufwand höher, da immer Resolve-Funktion ausgewertet werden muss

Vorteile (gilt für beide):

- Erkennt nicht initialisierte Signale in Simulation
- Erlaubt Modellierung von Tri-State-Signalen durch Signalwert 'Z'
- Ermöglicht Modellierung von schwachen Treibersignalen in Simulation
- Erlaubt don't care Modellierung

5.1.5 std_ulogic

Akzeptiert nur einen Treiber pro Signal. (Signalkonflikte von Compiler erkannt => Fehlermeldung)

Nachteile:

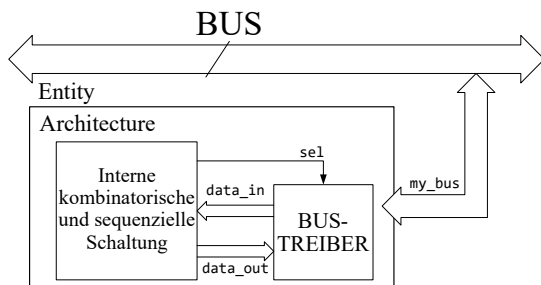
- Erst mit VHDL-2008 sauber definiert
- VHDL-2008 nicht von allen Tools vollständig unterstützt

Generell: Vorzugsweise std_ulogic verwenden. std_logic sollte grundsätzlich nur bei bidirektionalen Bussystemen verwendet werden.

5.1.6 Pull-Up & Pull-Down Widerstände in VHDL

```
1 -- innerhalb architecture:
2 pull_up : y <= 'H';
3 P1_logic : process(en1, d1)
4 begin
5   y <= 'Z';
6   if en1 = '1' and d1 = '1' then
7     y <= '0';
8   end if;
9 end process;
10
11 P2_logic : process(en2, d2)
12 begin
13   y <= 'Z';
14   if en2 = '1' and d2 = '1' then
15     y <= '0';
16   end if;
17 end process;
```

5.1.7 Bidirektionaler Bus-Driver in VHDL



```
1 library ieee;
2 use ieee.std_logic_1164.all;
3
4 entity bus_driver is
5 port(
6     sel      : in    std_ulogic;
7     data_out : in    std_ulogic_vector(7 downto 0);
8     my_bus   : inout std_ulogic_vector(7 downto 0);
9     data_in  : out   std_ulogic_vector(7 downto 0);
10 end bus_driver;
11
12 architecture behavioral of bus_driver is
13 begin
14     -- paralleler Lesevorgang:
15     read_logic : data_in <= to_stdulogicvector(my_bus);
16
17     write_logic : process(sel, data_out)
18     begin
19         if sel = '1' then
20             my_bus <= to_stdlogicvector(data_out); -- write to my_bus
21         else
22             my_bus <= (others => 'Z'); -- inactive (high impedance)
23         end if;
24     end process;
25 end behavioral;
```

6 Arithmetik in VHDL

bit_vector stellt keine Zahlen dar, sondern lediglich einen Haufen Bits.

6.1 ieee.numeric_std

6.1.1 Verfügbare arithmetische (Vergleichs-)Operationen

Operator	Bedeutung	Beispiel
=	gleich	... when A = B ...
/=	ungleich	... when A /= B ...
<	kleiner	... when A < B ...
<=	kleiner gleich	... when A <= B ...
>	grösser	... when A > B ...
>=	grösser gleich	... when A >= B ...

Operator	Bedeutung	Beispiel	Synthesefähigkeit
+	Addition	Y <= A + B	Ja
-	Subtraktion	Y <= A - B	Ja
*	Multiplikation	Y <= A * B	Ja
/	Division	Y <= A / B	Nur wenn B ∈ 2 ⁿ
**	Potenzbildung	Y <= A ** B	Nur wenn A ∈ 2 ⁿ
mod	Rest von A / B, Vorzeichen entspricht dem von B	Y <= A mod B	Nur wenn B ∈ 2 ⁿ
rem	Rest von A / B, Vorzeichen entspricht dem von A	Y <= A rem B	Nur wenn B ∈ 2 ⁿ

6.1.2 Einbindung

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
```

6.1.3 Anwendung

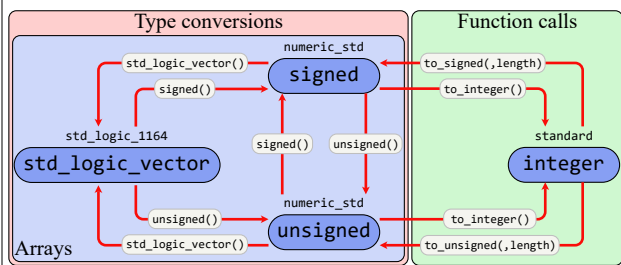
```
1 signal A_unsig: unsigned(3 downto 0);
2 signal B_sig  : signed(3 downto 0);
3 A_unsig <= "1111"; -- = 15 decimal
4 B_sig   <= "1111"; -- = -1 decimal
```

6.2 Datentypkonversion

VHDL verlangt gleiche Datentypen **und** Längen auf beiden Seiten bei Signalzuweisungen.

Konversionsfunktion	Argumenttyp (s)	Ergebnistyp (a)
a <= to_bit(s)	std_ulogic	bit
a <= to_stdulogic(s)	bit	std_ulogic
a <= to_bitvector(s)	std_ulogic_vector	bit_vector
	std_logic_vector	
a <= to_stdulogicvector(s)	bit_vector	std_ulogic_vector
	std_logic_vector	
a <= to_stdlogicvector(s)	bit_vector	std_logic_vector
	std_ulogic_vector	

6.2.1 Typecast und Typeconversion



- **Type-Cast:** Konversion zwischen abgeleiteten Typen (gleicher Basistyp)
- **Type-Conversion:** Konversion zwischen unterschiedlichen Typen

6.2.2 Vektoren erweitern und kürzen

Resize unsigned:

```
1 function resize(arg: unsigned; new_size: natural)
2 return unsigned;
```

- **Vergrößerung:** höherwertige Bits links werden mit 0 aufgefüllt.
- **Verkleinerung:** höherwertige Bits (links) werden abgeschnitten.

Resize signed:

```
1 function resize(arg: signed; new_size: natural)
2 return signed;
```

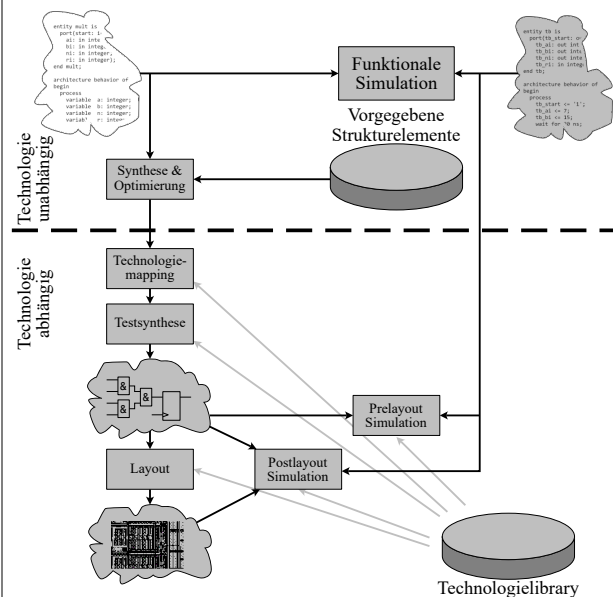
- **Vergrößerung:** höherwertige Bits links werden mit **sign Bit** aufgefüllt.
- **Verkleinerung:** sign Bit wird an MSB Position gesetzt.

Beispiel:

	Funktionsaufruf	Input (A)	Dec	Output (A')	Dec
signed	resize(A, 8);	1011	-5	1111'1011	-5
signed	resize(A, 4);	1111'1011	-5	1011	-5
unsigned	resize(A, 8);	0101	5	0000'0101	5
unsigned	resize(A, 4);	0000'0101	5	0101	5

7 Simulation / Test

7.1 Hierarchisches Vorgehen beim Design



7.2 Design vs. Test

7.2.1 Design

- Schaltungsdetails bis zum Gatterlevel
- Zeitliches Verhalten
- enable- & reset Signale

⇒ Beschränkt auf synthetisierbares Subset von VHDL.

7.2.2 Test

- Denkt in Kategorie v. Schaltungsverhalten & Funktionalität
- I/O Verhalten
- Quantisierungsfehler
- Datenübertragung

⇒ Im Testbench-Design **nicht** auf synthetisierbares Subset von VHDL beschränkt.

7.3 Event-based concept of time

7.3.1 Zeitablauf

- Simulation ⇒ zeitliches Verhalten eines digitalen Systems
- Reale Zeit entspricht in Simulation angezeigter Zeit
 - Hat **nichts** zu tun mit Rechenzeit der Simulation

7.3.2 Modelle

- Wie gut stimmt Simulation und Realität überein?
- Hersteller liefern Modelle mit Timinginfos bezüglich Basiszellen.
- Werden in Simulation **nach** der Synthese verwendet

7.4 Concurrency

7.4.1 Parallele Abläufe

VHDL-Simulationsresultate: Verhalten sich, als ob Parallel. In Wirklichkeit meiste Prozesse nicht parallel

7.4.2 Anforderungen an VHDL

VHDL braucht einen Mechanismus, der Prozesse für sequenzielle Ausführung aufbereitet, sodass perfekte Parallelität imitiert wird. Gleichzeitig müssen Signalverzögerungen abgebildet werden können.

7.5 Ablaufsteuerung einer Simulation

7.5.1 Event Queue

- Ereignisse in Liste eingetragen
- Simulation schreitet fort
- Jeder Zeitpunkt mit eingetragenen Ereignissen in Ereignisliste, wird bearbeitet

7.5.2 Simulationszeitpunkt (Event)

- Jeder Zeitpunkt, bei welchem in einer Schaltung ein Signalwert ändert
- Jedes event ereignet sich zu genau definierten Simulationszeit

7.5.3 Simulationsablauf

- Jeder Simulationszeitpunkt besteht aus mehreren Delta-Zyklen, die um infinitesimal kleine Zeiteinheit Δ verschoben sind.

7.5.4 Delta-Zyklus

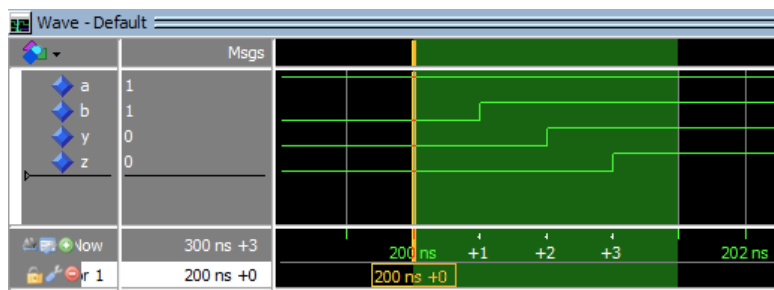
Stellt Ursache-Wirkung dar.

1. Warten bis Prozess durch Signalevent in sensitivity list getriggert (*update request*)
2. **Prozessausführung**: Alle aktiven Prozesse bis zum Ende / nächsten wait ausgeführt (Δ -Zyklus)
3. **Signalzuweisung**: Nach Ausführung aktiver Prozesse werden entsprechende Signalzuweisungen ausgeführt. Kann nun weitere Prozesse aktivieren für nächsten Delta-Zyklus
4. Simulation zum neuen Event vorrücken

7.5.5 Event driven Simulation

Simulation rechnet nur bei einem event. In der Zeit zwischen zwei events wird keine Rechenleistung benötigt. Simulation wird beendet, wenn keine Einträge mehr in der event queue sind oder wenn die vordefinierte Endzeit im Simulator erreicht wird.

7.5.6 Event queue sichtbar machen



7.6 Modellierung von Verzögerungszeiten

7.6.1 Functional Simulation: delta time

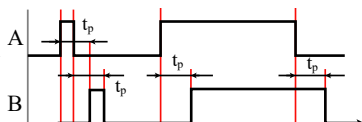
Erlaubt zeitliche Reihenfolge der Abarbeitung der Event queue einzuhalten $\Rightarrow$ Konsistenz Ursache/Wirkung

7.6.2 Timing Simulation

Transport delay:

Signal Assignment `y <= a or b` wird mit `transport...after` ergänzt

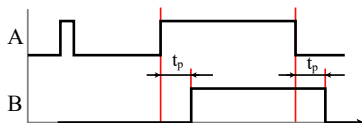
```
1 y <= transport a or b after 10 ns;
```



Inertial Delay Model:

- Signal Assignment wird mit `inertial...after` oder nur `after` ergänzt (gleichbedeutend)
- `y` ändert Wert nur, wenn Änderung von `a` mindestens `t1` lang ist

```
1 y <= a or b after 10 ns;  
2 y <= inertial a or b after 10 ns;
```

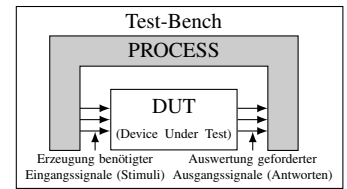


7.7 Testbenches

7.7.1 Aufbau

- Entity: Leere Entity, da keine Schnittstelle nach aussen
- Architecture: Testbench instanziiert DUT
 - Prozess für zeitlichen Ablauf (clock)
 - Prozess für Anlegen der Stimuli
 - Prozess für Prüfung der Systemantwort (response) mit Asserts

7.7.2 Bestandteile



7.7.3 Allgemeine Eigenschaften

- VHDL-Beschreibung, **nicht** synthesefähig $\Rightarrow$ voller VHDL-Sprachumfang darf verwendet werden
- Keine Schnittstelle nach aussen (in der Regel)
- Testvektor: einzelner Test durch Anlegen eines Stimulus an DUT (Eingangsvektor = Testvektor)
- Response: Systemantwort auf einen Testvektor
 - Wird mit erwarteter Antwort im Response-Monitor verglichen

7.7.4 Device Under Test (DUT)

Prüfling:

- DUT Unverändert aus Bibliothek übernommen
- DUT wird als interne Komponente der Test-Bench deklariert und instanziiert.

Konfiguration:

- Testbench muss verschiedene Architekturen des Prüflings einfach einbinden können
- Mit einer Konfigurationsangabe kann eine von mehreren vorhandenen Architekturen ausgewählt werden:

```
1 for all: exor use entity work.exor(behavioral);
```

- Oder gar pro Instanz je eine unterschiedliche Architektur:

```
1 for dut1: exor use entity work.exor(structural);  
2 for dut2: exor use entity work.exor(behavioral);
```

Signale: Um DUT mit Testbench zu verdrahten

7.7.5 Stimuli

- Zur Simulation benötigte Eingangssignale (Testvektoren)
 - $\Rightarrow$ Kombinieren Wert- & Zeitinformation
 - Entweder: In Prozess ausprogrammiert (unflexibel, aber für kleinere Projekte vorwiegend verwendet)
 - Oder: Separat in ASCII-File abgelegt (kompliziert, aber zur Wiederverwendung besser geeignet)

Beispiel: Clock Prozess

- In Testbenches kommen häufig Prozesse ohne Sensitivitätsliste zum Einsatz
 - $\Rightarrow$ Erlaubt, wenn im Prozess ein `wait`-Statement steht.

```
1 --Frequency f in Hz  
2 constant f: integer := 1000;  
3 --PeriodeT in sec  
4 constant T: time := 1 sec/f;
```

Variante 1:

```
1 signal clk1 : std_ulogic;  
2 clock1 : process  
3 begin  
4   clk1 <= '1';  
5   loop  
6     wait for (T / 2);  
7     clk1 <= not clk1;  
8   end loop;  
9 end process;
```

Variante 2:

```
1 signal clk0 : std_ulogic;  
2 clock0 : process  
3 begin  
4   clk0 <= '1'; wait for (T / 2);  
5   clk0 <= '0'; wait for (T / 2);  
6 end process;
```

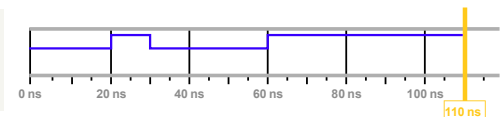
Variante 3 (Lazy):

```
1 signal clk2 : std_ulogic := '1';  
2 clock2: clk2 <= not clk2 after (T / 2);
```

Beispiel: Signalerzeugung

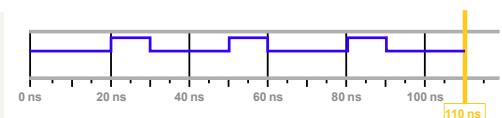
Einfacher Signalverlauf:

```
1 tb_s0 <= '0';  
2 '1' after 20 ns,  
3 '0' after 30 ns,  
4 '1' after 60 ns;
```



Wiederkehrende Signalverläufe:

```
1 sequenz : process  
2 begin  
3   tb_s1 <= '0';  
4   wait for 20 ns;  
5   tb_s1 <= '1';  
6   wait for 10 ns;  
7 end process sequenz;
```



7.7.6 Response (Geforderte Ausgangssignale)

- Erzeugt von "golden device" (in unseren einfachen Beispielen häufig Kontrolle von Hand. In grösseren Beispielen z.B. eine funktionale Realisierung des Designs.)
- Extern gespeichertes ASCII-File mit den erwarteten Systemantworten, erzeugt von Hand oder durch Automated Test Equipment (ATE)

Assert:

```
1 [<label>:] assert (condition) report ["message"]
2 severity (note | warning | error | failure);
```

- **assert** (condition): prüft, ob Systemantwort der **erwarteten Condition** entspricht
- **report "message"**: (**Optional**) wird ausgegeben, falls condition = false
- **severity**: (**Optional**) Schweregrad des Fehlers (Absturz meist bei **error** und **failure**), je nach Simulator-Einstellungen

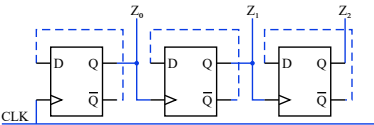
Beispiel: Testbench

```
1 -- Libraries
2 library ieee;
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 -- Entity declaration (must be empty)
7 entity TestBench is
8 end entity TestBench;
9
10 architecture RTL of TestBench is
11     -- component declaration
12     component comp_a
13     port(
14         inp      : in std_ulogic_vector(7 downto 0);
15         clk, rst  : in std_logic;
16         oup1, oup2 : out std_ulogic_vector(3 downto 0));
17     end component comp_a;
18     -- Internal signal declarations
19     signal inp_tb      : std_ulogic_vector(7 downto 0);
20     signal oup1_tb, oup2_tb : std_ulogic_vector(3 downto 0);
21     signal clk_tb, rst_tb : std_ulogic;
22
23     -- Constant declarations
24     constant test_frequency : integer := 1e6; --1'000'000;
25     constant test_period   : time := 1 sec / test_frequency;
26     constant sample_before_clk : time := 1 ns;
27 begin
28     -- Component instantiation
29     dut : component comp_a
30     port map(
31         inp => inp_tb,
32         clk => clk_tb,
33         rst => rst_tb,
34         oup1 => oup1_tb,
35         oup2 => oup2_tb);
36     -- Clock generator
37     clk_generator : process
38     begin
39         clk_tb <= '0'; wait for test_period / 2;
40         clk_tb <= '1'; wait for test_period / 2;
41     end process;
42     -- reset generator: '1' for 8 cycles, then '0'
43     reset_generator : rst_tb <= '1', '0' after 8*test_period;
44     -- Generate all possible stimuli
45     stimuli_generator : process
46     begin
47         inp_tb <= (others => '0'); -- default value
48         wait until rst_tb = '0';
49         wait until rising_edge(clk_tb);
50         -- generate all possible stimuli for signal inp_tb
51         for i in 0 to 2**8 - 1 loop
52             inp_tb <= std_ulogic_vector(to_unsigned(i, 8));
53             wait for test_period;
54         end loop;
55     end process;
56
57     -- Check system response with asserts
58     response_monitor : process
59     begin
60         wait until (rising_edge(clk_tb) and (rst_tb = '0'));
61         for i in 0 to 2**8 - 1 loop
62             wait for (test_period - sample_before_clk);
63             assert (oup1_tb = oup1_expected)
64                 report "mismatch at vector " & integer'image(i)
65                 severity warning;
66             assert (oup2_tb = oup2_expected)
67                 report "mismatch at vector " & integer'image(i)
68                 severity warning;
69             wait for sample_before_clk;
70         end loop;
71     end process;
72 end architecture RTL;
```

8 Modellparametrisierung

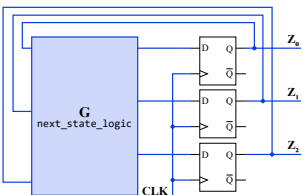
8.1 Zähler

8.1.1 Asynchron (Kaskade)



- Vorteile:**
- Einfache Hardware
 - Minimale Anzahl an Flip-Flops
- Nachteile:**
- Asynchrone Ausgänge

8.1.2 Synchron (FSM)



- Vorteile:**
- Alle Ausgänge stehen gleichzeitig zur Verfügung => keine Timingprobleme
 - Systematischer Entwurf für unterschiedlichste Zähler

Beispiel: Counter2

- Ausgangspunkt:**
- Synchroner 2-Bit Zähler der Kaskadiert werden kann
 - Nicht optimal, gibt bessere Varianten
- Entwurf:**
1. Zustandsdiagramm
 2. Zustandstabelle
 3. VHDL Beschreibung
 4. Synthese

Realisierung in VHDL als FSM:

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4
5 entity counter_sync is
6 port(
7     rst, clk, ena : in std_ulogic;
8     carry         : out std_ulogic;
9     count         : out std_ulogic_vector(1 downto 0));
10 end entity;
11
12 architecture sync_ena of counter_sync is
13     signal cnt_present, cnt_next : unsigned(1 downto 0);
14     signal carry_next            : std_ulogic;
15     constant maxcount           : unsigned(1 downto 0) := "11";
16 begin
17     Next_state_logic: process(cnt_present, ena)
18     begin
19         carry_next <= '0';
20         cnt_next <= cnt_present; -- Defaults
21         if ena = '1' then
22             if cnt_present = maxcount then -- range exceeded?
23                 cnt_next <= "00";
24                 carry_next <= '1';
25             else cnt_next <= cnt_present + "01"; -- also accepted: + 1;
26             end if;
27         end if;
28     end process;
29     FF2: process(clk)
30     begin
31         if rising_edge(clk) then
32             if rst = '1' then
33                 cnt_present <= (others => '0');
34                 carry <= '0';
35             else
36                 cnt_present <= cnt_next;
37                 carry <= carry_next;
38             end if;
39         end if;
40     end process;
41     output_logic: count <= std_ulogic_vector(cnt_present);
42 end architecture;
```

8.2 Parametrisierung

8.2.1 Generics

- VHDL kann Parameter an ein Modell übergeben
- Deklariert in Schnittstellenbeschreibung
- **generic** muss in **entity** und **component** Deklaration vorkommen

Deklaration: **generic**(<param_name>: <param_type> [:= default_value]);

Beispiel: Deklaration & Instanziierung

Deklaration Entity:

```
1 entity counter_generic is
2     generic(CNT_MAX : natural := 127);
3     port(
4         clk, rst, ena : in std_ulogic;
5         count         : out natural range 0 to CNT_MAX - 1);
6 end counter_generic;
```

Deklaration Komponente:

```
7 architecture structural of counter_array is
8   constant maxcount0 : natural := 14;
9   constant maxcount1 : natural := 16;
10  signal clk, rst, ena : std_ulogic;
11  signal count0 : natural range 0 to maxcount0;
12  signal count1 : natural range 0 to maxcount1;
13  signal carry0, carry1 : std_ulogic;
14
15  component counter_generic
16    generic(CNT_MAX: natural);
17    port(
18      clk, rst, ena : in std_ulogic;
19      count : out natural range 0 to CNT_MAX - 1;
20      carry : out std_ulogic);
21  end component;
```

Instanziierung Komponente:

```
22 begin
23   cnt0: component counter_generic
24     generic map(CNT_MAX => maxcount0)
25     port map( -- clk, rst, ena implizit
26       count => count0,
27       carry => carry0);
28   cnt1: component counter_generic
29     generic map(CNT_MAX => maxcount1)
30     port map( -- clk, rst, ena implizit
31       count => count1,
32       carry => carry1);
33 end architecture structural;
```

8.2.2 Unterschiede der Konzepte: Parameter fixiert zu...

...Designtime:

Konstanten legen Zählbereich in VHDL-Design fest (designtime). Macht Code übersichtlich, verständlich. Beispiel: Zählbereich 0 ... 4.

...Compiletime:

Generic Konzept übermittelt Information zu compiletime Macht Code wiederverwendbar. Beispiel: Wiederverwendbarer Zähler mit Zählbereich 0 ... maxcount.

...Runtime:

Signale übermitteln Information zu runtime. Beispiel: Zählerbereich kann über Schalter eingestellt werden (siehe Praktikum)

8.2.3 Anpassen der Wortbreite

Vorsicht bei variablem Interface:

Signale an Interfaces sollten grundsätzlich statisch sein. Modifikationen der Interfaces erzwingen in der Regel manuelle Anpassungen in hierarchischen Designs.

In Spezialfällen und Modulen:

Bei parametrisierten Zellen ist Anpassung zur Zeit der Instanziierung möglich. Intern ist Berechnung der Wortbreite aus Parameter sogar erwünscht und unproblematisch, da Berechnung der Wortbreite vor der Synthese erfolgt.

Beispiel: Generic Counter

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 use ieee.numeric_std.all;
4 use ieee.math_real.log2;
5 use ieee.math_real.ceil;
6 entity counter_generic is
7   generic(CNT_MAX : natural := 127);
8   port(
9     clk, rst, ena : in std_logic;
10    count : out std_ulogic_vector
11      (natural(ceil(log2(real(CNT_MAX + 1)))) - 1 downto 0));
12 end counter_generic;
```

Beispiel: Clockdivider

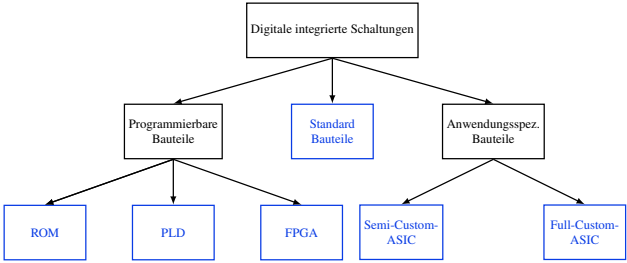
Ausgang kippt, wenn Maximalwert erreicht wird.

```
1 library ieee;
2 use ieee.std_logic_1164.all;
3 entity clockdivider is
4   generic(DIV_FACTOR : natural := 128);
5   port(
6     clk : in std_ulogic;
7     rst : in std_ulogic;
8     ena : in std_ulogic;
9     clk_out : out std_ulogic);
10 end entity clockdivider;
11 architecture behavioral of clockdivider is
12   -- pro Register ein present- und next-signal
13   signal cnt_present, cnt_next : natural range 0 to (DIV_FACTOR - 1);
14   signal tick_present, tick_next : std_ulogic;
15
16 begin
```

```
17   output_logic : clk_out <= tick_present;
18   next_state_logic : process(cnt_present, tick_present)
19   begin
20     if (cnt_present = div_factor - 1) then
21       tick_next <= not tick_present;
22       cnt_next <= 0;
23     else
24       cnt_next <= cnt_present + 1;
25       tick_next <= tick_present;
26     end if;
27   end process;
28   registers : process(clk)
29   begin
30     if rising_edge(clk) then
31       if (rst = '1') then
32         cnt_present <= 0;
33         tick_present <= '0';
34       elsif ena = '1' then -- Flip-Flop with enable input
35         cnt_present <= cnt_next;
36         tick_present <= tick_next;
37       end if;
38     end if;
39   end process;
40 end architecture behavioral;
```

9 Anhang

9.1 Übersicht Realisierungsformen



9.2 Kriterien zur Auswahl der Realisierungsform

Kriterien	Std-Bauteile	ROM	PLD	FPGA	Semi-Custom	Full-Custom
Machbarkeit	++	--	--	+	+	+++
Zeit Realisierungs	+	++	++	++	-	--
Iterationszeit	-	++	++	++	-	--
NRE	++	+	+	+	-	---
Stückpreis	--	+	+	-	+	+++