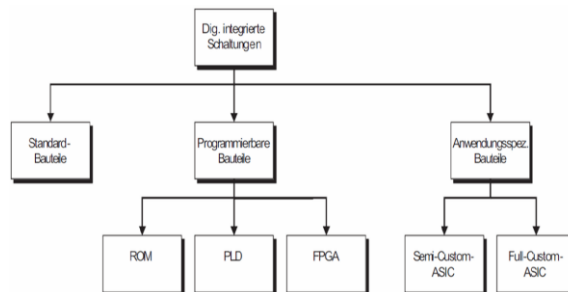
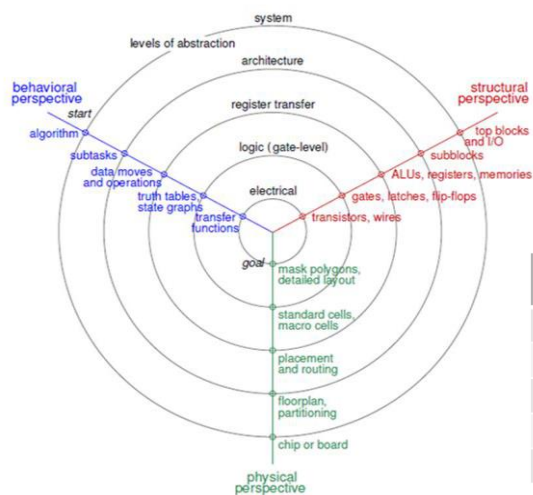
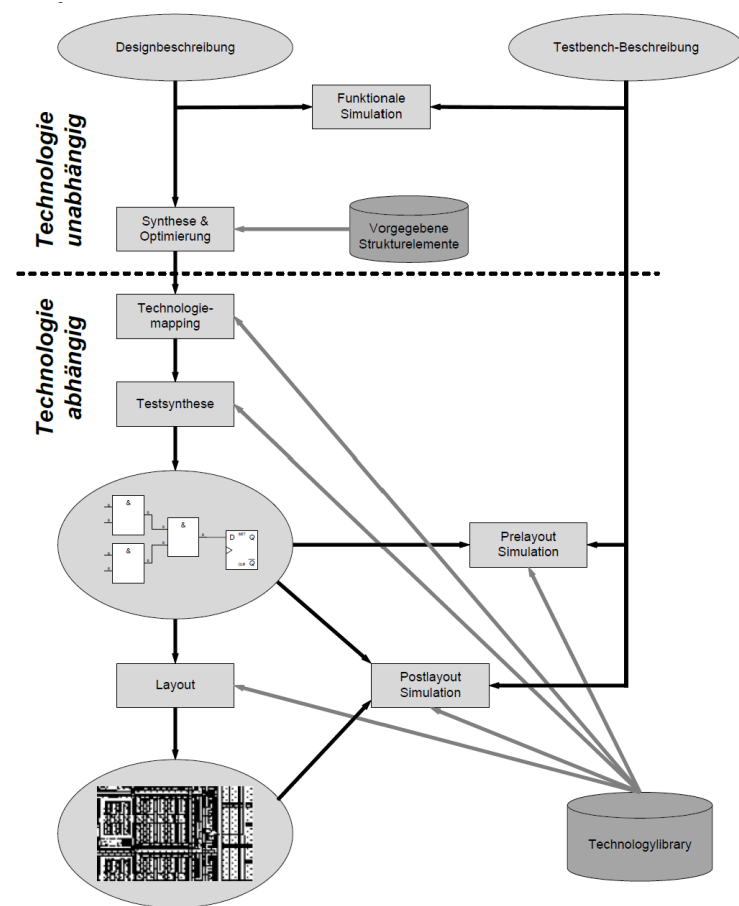


Realisierungsformen

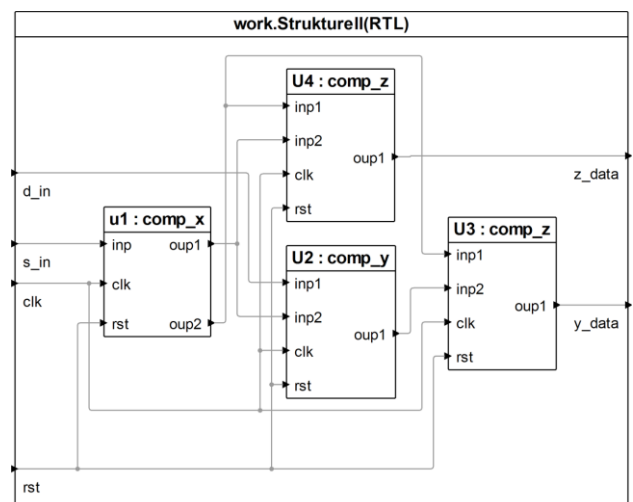


Kriterien	Standard Bauteile	ROM	PLD	FPGA	Semi-Custom	Full-Custom
Machbarkeit	++	--	--	+	+	+++
Zeit Realisierung	+	++	++	++	-	--
Iterationszeit	-	++	++	++	-	--
NRE	++	+	+	+	-	---
Stückpreis	--	+	+	-	+	+++

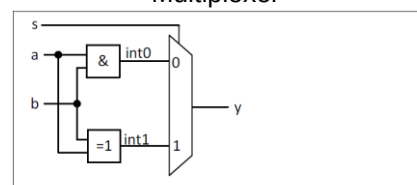
Digital Design-Flow



Blockschemata



Multiplexer



VHDL-Syntax

Beispiel 3.1 Kommentarzeilen

```

-- Dies ist eine ganze Kommentarzeile
-- auch eine zweite Kommentarzeile muss mit "--"
-- eingeleitet werden
entity nand4 is -- Kommentar ab hier bis zum Zeilenende
/* Dies ist ein block comment.
Er wird erst durch
Die folgenden Zeichen abgeschlossen: */

```

Als Bezeichner (Prozess-, Variablen-, Signalnamen, etc.) sind alle alphanumerischen Zeichenketten bestehend aus A..Z, a..z, 0..9, _ mit folgenden Einschränkungen zugelassen:

- Start mit Buchstaben
- _ nicht am Ende oder doppelt (__)
- Case insensitive (Bsp: Input_signal ist identisch mit input_signal)
- Schlüsselwörter dürfen nicht verwendet werden

Entity & Components

```
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;
```

Variablen (nur in Prozessen):

- ➔ **Eingangssignale** eines Prozesses bleiben **während** der internen prozeduralen Berechnung **unverändert** und werden **erst am Ende** des Prozesses aktualisiert. Muss während der Berechnung auf ein Wert zugegriffen werden, bieten sich Variablen an.
- ➔ einer Variable wird bei der Deklaration ein (einmaliger) **Initialisierungswert** zugewiesen (sonst entsteht Latch!)

```
full_adder : process(a, b, ci)
  variable yint1 : bit_vector(2 downto 0) := (others => '0');
  variable yint2 : bit_vector(1 downto 0) := (others => '0');

begin
  yint1 := (ci, b, a); -- Aggregat zur Bündelung
                        -- von einzelnen Bits

  case yint1 is
    when "000" => yint2 := "00";
    when "001" => yint2 := "01";
    when "010" => yint2 := "01";
    when "011" => yint2 := "10";
    when "100" => yint2 := "01";
    when "101" => yint2 := "10";
    when "110" => yint2 := "10";
    when "111" => yint2 := "11";
  end case;

  sum <= yint2(0);
  co <= yint2(1);
end process full_adder;
```

Zeiten zur Parameterfixierung

- ➔ **Konstanten** legen Parameter zu **designtime** fest, dies macht den Code übersichtlicher und verständlicher. (Bsp.: Zählbereich 0...4)
- ➔ **Generic-Konzept** ermöglicht die Festlegung von Parametern zu **compiletime** und macht Code wiederverwendbar. (Bsp.: Zählbereich 0...maxcount)
- ➔ **Signale** übermitteln Information zu **runtime**. (Bsp.: Einstellung des Zählbereichs mit einem Schalter)

```
architecture RTL of countdown_counter is
  component modulo_n_counter
    generic(n : positive);
    port(
      clk : in std_ulogic;
      rst : in std_ulogic;
      count_out : out std_ulogic_vector(natural(ceil(log2(real(n)))) - 1 downto 0)
    );
  end component modulo_n_counter;

begin
  minute_counter : component modulo_n_counter
    generic map(
      n => 10
    )
    port map(
      clk => clk,
      rst => rst,
      count_out => minute
    );
end;
```

Instanzierung generic component

Nebenläufige Signalzuweisungen & Prozesse

Selektive Signalzuweisung

Unbedingte Signalzuweisung

`y <= a and b;`

Bedingte Signalzuweisung

```
architecture conditional of mux4x1 is
begin
  y <= e(0) when s = "00" else
      e(1) when s = "01" else
      e(2) when s = "10" else
      e(3) when s = "11";
end conditional;
```

Aggregate & Auskopplung von Vektorelementen

```
vector_implicite <= ('0', '0', i3, i2);
vector_explicite <= (1 => i3, 0 => i2, others => '0');
i1 <= inp(3);
(i0, i1, i2, i3) <= inp;
int4 <= (in1 & int3);
outr <= int3;
co <= int1(4);
```

msb & lsb

Sequenzielle Anweisungen innerhalb von

Prozessen:

- ➔ **if-then-else** und **case**-Anweisungen können **nur innerhalb von Prozessen** verwendet werden
- ➔ **selektive** und **bedingte** Signalzuweisungen dürfen hingegen **nicht** in Prozessen verwendet werden
- ➔ Prozesse werden innerhalb **prozedural** abgearbeitet (ein Befehl nach dem anderen)

Übersicht Signalzuweisungen

```
-- Unbedingte Signalzuweisungen
x <= (a and b and not c) or b;

-- Bedingte Signalzuweisungen
x <= a when (c = '1') else b;

-- Selektive Signalzuweisungen
with s select
  z <= e(0) when "00",
      e(1) when "01",
      e(2) when others;

-- Parallele Prozesse
p1 : process ( x, y, ... )
begin -- sequenziell innerhalb der Prozesse
  ...
end process p1;
```

Parametrisierbare Modelle (mit generic)

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

entity counter_generic is
  generic(
    MAXCOUNT : integer := 100
  );
  port(
    clk : in std_ulogic;
    reset : in std_ulogic;
    count_en : in std_ulogic;
    count : out integer range 0 to MAXCOUNT - 1
  );
end counter_generic;

architecture behavioral of counter_generic is
  signal present_counter, next_counter : integer range 0 to MAXCOUNT - 1;

begin
  output_logic : count <= present_counter;

  next_state_logic : process(present_counter, count_en)
  begin
    if count_en = '1' then
      if present_counter >= MAXCOUNT - 1 then
        next_counter <= 0;
      else
        next_counter <= present_counter + 1;
      end if;
    else
      next_counter <= present_counter;
    end if;
  end process;

  registers : process(clk)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        present_counter <= 0;
      else
        present_counter <= next_counter;
      end if;
    end if;
  end process;
end behavioral;
```

Beispiel für einen counter mit generic

```
entity N_bit_counter_generic is
  generic(n_bits : natural := 8);
  port(
    clk : in std_ulogic;
    rst : in std_ulogic;
    ena : in std_ulogic;
    clr : in std_ulogic;
    count : out std_ulogic_vector(n_bits - 1 downto 0)
  );
end entity N_bit_counter_generic;

architecture behavioral of N_bit_counter_generic is
  signal present_counter, next_counter : unsigned(n_bits - 1 downto 0);

begin
  output_logic : count <= std_ulogic_vector(present_counter);
end;
```

Variante mit std_ulogic_vector

Finite State Machines (FSM)

Zustandskodierungsarten:

(Bild 1.: Synthesewerkzeug macht's, z.B. mittels Gray-Codierung)

```
type state_typ is (st_normal, st_hset, st_mset, st_ssync);
signal present_state, next_state : state_typ;

-- type declaration
type state_typ is (st_normal, st_hset, st_mset, st_ssync);

-- explizite Codierung mit enum_encoding
attribute enum_encoding : string;
attribute enum_encoding of state_typ :
type is "0001 0010 0100 1000"; -- one hot codierung
signal present_state, next_state : state_typ;

-- Explizite Codierung mit Konstanten:
subtype state_type is bit_vector(3 downto 0);
-- one cold codierung
constant st_normal : state_type := "1110";
constant st_hset : state_type := "1101";
constant st_mset : state_type := "1011";
constant st_ssync : state_type := "0111";
signal present_state, next_state : state_type;
```

Beispiel FSM mit 3 Prozessen

```
architecture behavioral of controller is
-- type declaration
type state_typ is (st_normal, st_hset, st_mset, st_ssync);
-- signal declaration for states
signal present_state, next_state : state_typ;
begin

output_logic : process(present_state, mode, set)
begin
hset <= '0'; -- Defaultwerte für alle Ausgänge
mset <= '0';
ssync <= '0';
mode_hset <= '0';
mode_mset <= '0';
mode_ssync <= '0';
case present_state is
when st_normal =>
-- Teil fehlt!
when st_hset =>
mode_hset <= '1'; -- moore Ausgang
hset <= not mode and set; -- mealy Ausgang
when st_mset =>
mode_mset <= '1';
mset <= not mode and set;
when others =>
mode_ssync <= '1';
ssync <= not mode and set;
end case;
end process;
```

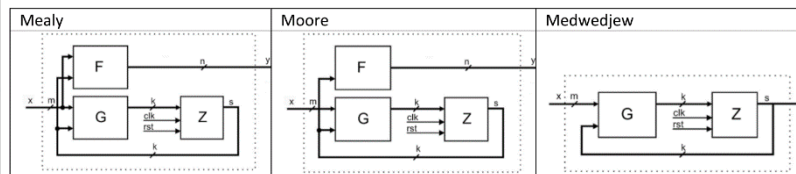
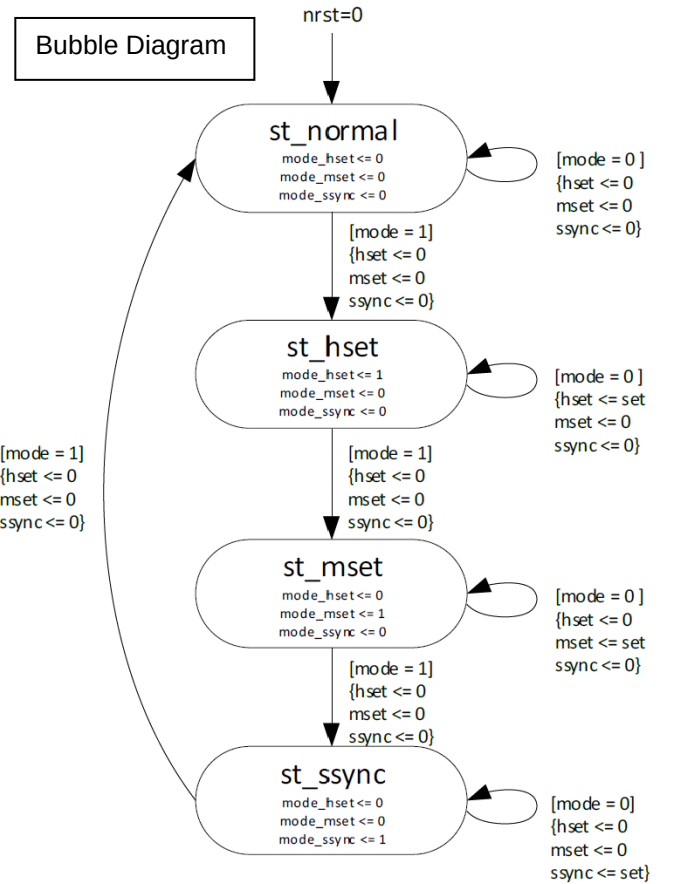
```
next_state_logic : process(present_state, mode)
begin
next_state <= present_state;
case present_state is
when st_normal =>
if (mode = '1') then -- Zustandsübergang
next_state <= st_hset;
end if;
-- Falls mode /= '0'
-- => next_state durch Default gesetzt
when st_hset =>
if (mode = '1') then
next_state <= st_mset;
end if;
when st_mset =>
if (mode = '1') then
next_state <= st_ssync;
end if;
when others =>
if (mode = '1') then
next_state <= st_normal;
end if;
end case;
end process;
```

```
process(clk)
begin
if rising_edge(clk) then
if (reset = '1') then
q <= '0';
elsif (load = '1') then
q <= d;
end if;
end if;
end process;
```

synchroner Reset

```
registers : process(nrst, clk)
begin
if (nrst = '0') then
present_state <= st_normal;
elsif clk'event and (clk = '1') then
present_state <= next_state;
end if;
end process;
```

asynchroner Reset



Testbenches (tb)

Clock-Erzeugung (3 Varianten)

```
-- Frequency f in Hz
constant f : integer := 1000;
-- Periode T in sec
constant T : time := 1 sec/f;

clock0 : process
begin
clk0 <= '1'; wait for (T / 2);
clk0 <= '0'; wait for (T / 2);
end process;

clock1 : process
begin
clk1 <= '1';
loop
wait for (T / 2); clk1 <= not clk1;
end loop;
end process;

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

Einfache(o.r.) & wiederkehrende(andere) 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;
end process;
```

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

sequenz : process
begin
tb_s1 <= '0';
wait for 20 ns;
tb_s1 <= '1';
wait for 10 ns;
end process sequenz;
```

Bsp. Response-Auswertung

```
response_monitor : process -- Resultate werden kurz vor Ende des
begin -- Simulationszyklus' abgefragt
wait for (SIM_CYC - 10 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;
assert (tb_y = '1') report "error at vector 10" severity error;
wait for SIM_CYC;
assert (tb_y = '0') report "error at vector 11" severity error;
wait;
end process;
```

wie schwerwiegend der aufgetretene Fehler ist. Er kann die Werte note, warning, error und failure annehmen, wobei die letzten beiden üblicherweise zum Abbruch der Simulation führen.

Testbench:

- entity hat keine Schnittstelle (nach Aussen)
- DUT wird als Component instanziiert
- ie 1 Prozess für zeitl. Ablauf, Stimuli & Auswertung

```
constant test_frequency : integer := 1e6; --1'000'000;
constant test_period : time := 1 sec / test_frequency;
constant sample_before_clk : time := 1 ns;
```

Elektrische Signale

library ieee;
use ieee.std_logic_1164.all;

- normalerweise **bit** oder **std_ulogic** verwenden
- **std_logic** nur bei bidirektionalem Bus!

Wert	Bedeutung	Verwendung
'U'	Nicht initialisiert	Das Signal ist im Simulator (noch) nicht initialisiert
'X'	Undefinierter Pegel	Simulator erkennt mehr als einen aktiven Signalreiber (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-Drain-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 (don't care)

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

Implementierung Pull-Up/Pull-Down (plus 2 treibende Gatter)

```
begin
  pull_up : y <= 'H';

  P1_logic : process(en1, d1)
  begin
    y <= 'Z';
    if en1 = '1' and d1 = '1' then
      y <= '0';
    end if;
  end process;

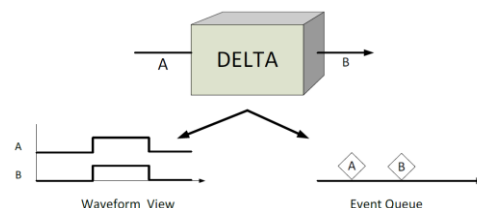
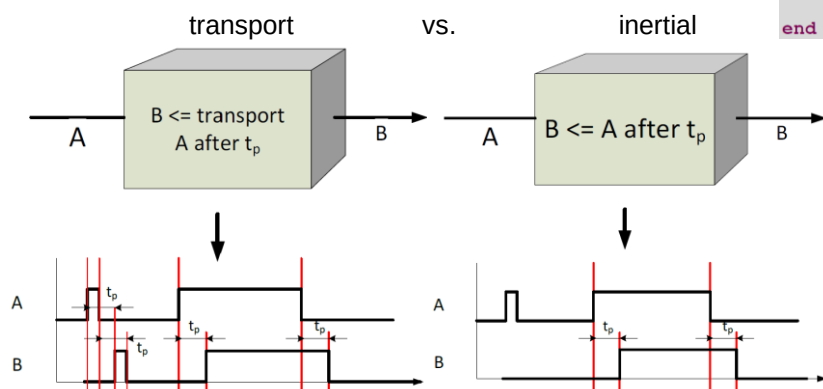
  P2_logic : process(en2, d2)
  begin
    y <= 'Z';
    if en2 = '1' and d2 = '1' then
      y <= '0';
    end if;
  end process;
```

Modellierung bidirektionaler Bus

```
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
      bus_i_o <= (others => 'Z'); -- Hochimpedanz
    end if;
  end process write;

  read : in_i <= bus_i_o; -- paralleler Lesevorgang
end tri_state_out;
```

Das **Delta-Time-Modell** wird bei funktionalen Symolationen angewendet, um mittels einer infinitesimal kurzen Zeit zwischen Ursache und Wirkung unterscheiden zu können.



Arithmetik

math_real-Bibliothek für Bitwerte aus generic-Variablen

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

entity modulo_n_counter is
  generic(n : positive := 100);
  port(
    clk      : in  std_ulogic;
    rst      : in  std_ulogic;
    count_out : out std_ulogic_vector(natural(ceil(log2(real(n)))) - 1 downto 0)
  );
end entity modulo_n_counter;
```

Beispiel 8.5 Physikalische Datentypen

```
type CAPACITANCE is range 0 to 1E30
units
  pf; --picofarad
  nf = 1000 pf; --nanofarad
  uf = 1000 nf; --microfarad
  mf = 1000 uf; --millifarad
end units;
```

Beispiel 8.6 Array of integer

```
type sig_array_typ is array (9 downto 0) of integer;
```

Beispiel 8.7 Resize von signed Signalen bei einer Addition

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

entity resize_bsp is
  port (
    a,b : in  signed(2 downto 0);
    y    : out signed(4 downto 0)
  );
end entity resize_bsp;

architecture behavioral of resize_bsp is
begin
  y <= resize(a, y'length) + resize(b, y'length);
  -- Oder:
  -- y <= resize((a + b), y'length);
end architecture behavioral;
```

