VHDL

Wikibooks.org

16. Februar 2012

Inhaltsverzeichnis

Dieses Buch steht im Regal Elektrotechnik[1].

0.1 Einführung

VHDL (VHSIC (Very High Speed Integrated Circuits) Hardware Description Language) ist in Europa die verbreitetste Hardware-Beschreibungssprache. Daneben gibt es VERILOG. Ursprünglich wurde sie entwickelt, um Testumgebungen für die Simulation integrierter Schaltungen zu entwickeln. Daher ist VHDL eine relativ komplexe Programmiersprache, deren Konstrukte nicht zwangsläufig synthetisierbar sind. Das führt bereits zum üblichen Ablauf der Hardwareentwicklung: Nach der Festlegung der Funktionalität wird diese mittels VHDL beschrieben. Der entstandene "Source-Code" wird compiliert und anschließend simuliert. Nachdem die einwandfreie Funktion sichergestellt ist wird der VHDL-Code direkt mit Hilfe eines Syntheseprogramms in eine Gatternetzliste umgesetzt. Dazu benötigt das Synthesewerkzeug eine von der Zielhardware abhängige Elementbibliothek, die der Chiphersteller zur Verfügung stellt.

0.1.1 Zum Wesen von VHDL

Im Innersten besteht jede digitale Hardwareschaltung aus kombinatorischer Logik und speichernden Elementen.

Unter Kombinatorik versteht man "NICHT/UND/ODER/Exclusiv-ODER"-Gatter und deren Kombinationen. Also jede Art von Verknüpfungen von einem oder mehreren Eingängen zu einem oder mehreren Ausgängen. Eine Änderung eines Eingangs bewirkt eine unmittelbare Wirkung auf den Ausgang. Eine wirkliche Hardwareschaltung benötigt dazu jedoch geringe

1 http://de.wikibooks.org/wiki/Regal%3AElektrotechnik

Laufzeiten. Diese Laufzeiten kombinatorischer Logik werden aber in einer reinen "RTL"-Beschreibung nicht im VHDL-Code modelliert, obwohl dies an sich möglich wäre.

Speichernde Elemente sind "Flipflops" oder "Latches". Latches sollte man als Zustandsspeicher in synchronen Schaltungen vermeiden. Flipflops sind Grundelemente die einen Dateneingang und einen Takteingang haben. Der Ausgang übernimmt üblicherweise den Zustand des Eingangs mit der steigenden Taktflanke. Es gibt auch Flipflops die zur fallenden Flanke schalten. Zusätzlich können Flipflops auch einen asynchronen Reset-Eingang haben. Dieser setzt im Normalfall zu Beginn, nach Anlegen der Versorgungsspannung, das Flipflop in den Grundzustand (Ausgang hat den Pegel "0").

Um diese zwei Arten der realen Hardware nachzubilden ist die grundsätzliche Denkweise für ein VHDL-Programm deutlich anders, als es für den seriellen Ablauf beispielsweise eines C-Programms nötig ist. VHDL ist im Grunde eine Aneinanderreihung von Prozessen, die quasi simultan abgearbeitet werden. Der VHDL-Simulator wird zwar den Code in irgendeiner Weise in eine serielle Software umwandeln. Die Wirkungsweise der Prozesse ist jedoch so, als würden sie wirklich völlig gleichzeitig bearbeitet.

Zu welchen Zeitpunkt ein Prozess abgearbeitet wird, bestimmt die "Sensitivity-List". Dies ist im Prozess-Header eine Liste von vereinbarten Signalen. Ändert eines dieser Signale den Wert so wird der Prozess angestoßen. Führt dieses zu einer Änderung eines Ausgangs und ist dieser wiederum in der Sensitivity-List eines anderen Prozesses, so wird auch dieser mit dem neuen Wert des Signals angestoßen.

Man unterscheidet zwei Typen von Prozessen: Kombinatorische und synchron getaktet Prozesse, analog zu der eingangs besprochenen realen Hardware. Kombinatorische Prozesse haben in der Sensitivity-List alle Eingangs-Signale und beschreiben im Inneren deren Verknüpfung. Synchron getaktet Prozesse haben in der Sensitivity-List nur "reset" und "clock". Im Inneren wird beschrieben welches Signal oder auch welche Verknüpfung von Signalen zur Taktflanke am Ausgang übernommen werden soll.

Beispiel für einen kombinatorischen Prozess:

```
procname: process(a,b,c)
begin
  x <= (a and b) or c;
end process;
```

-- Ausgang "x" ist eine Verknüpfung von "a","b","c"

Beispiel für einen synchron getakteten Prozess:

```
procname: process(nres,clk)
begin
  if (nres='0') then
     q <= '0';
  elsif (clk'event and clk='1') then
     q <= x;
  end if;
end process;
```

-- FlipFlop mit Ausgang "q" schaltet mit steigender "clk"-Flanke und übernimmt den Wert von "x" -- x stammt aus kombinatorischem Prozess!!

Zur Beschreibung kombinatorischer Vorgänge gibt es die aus vielen Sprachen bekannte Konstrukte:

```
n: process(a,b)
begin
  x <= a and b;    -- Und-Verknüpfung von "a" und "b"
end;
```

--dazu gleichwertig:

```
n: process(a,b)
begin
  if (a='1') and (b='1') then
     x <= '1';
  else
     x <= '0';
  end if;
end;
```

--dazu gleichwertig:

```
n: process(a,b)
variable ab : std_logic_vector(1 downto 0);
begin
  ab(0) := a;
  ab(1) := b;
  case ab is
    when "00" => x <= '0';
    when "01" => x <= '0';
    when "10" => x <= '0';
    when "11" => x <= '1';
    when others => null;
  end case;
end;
```

Im Weiteren baut VHDL einen begrenzten Rahmen von logischen Elementen als ein Bauelement zusammen, das wiederum mit der diskreten Schaltungstechnik vergleichbar ist. So ist es ähnlich wie in der altbekannten TTL-Schaltungstechnik, dass ein solches Bauteil Eingänge, Ausgänge und ein Innenleben hat. VHDL definiert dieses Element oder diesen Block mit seinen "Pins" in der "Entity". Die "Architecture" beschreibt dann mit den oben gezeigten Prozessen das Innenleben.

Formal:

```
entity 74LS00 is
  port (i0,i1 : in bit;
        i2,i3 : in bit;      -- Eingänge
        o0    : out bit;     -- Ausgang
        o1    : out bit);
end 74LS00;



architecture behv of 74LS00 is
begin
signal z : bit;      -- Vereinbarung internen Signale
```

```
comb_1: process(i0,i1,i2,i3)
  begin
    o0 <= not(i0 and i1);    -- nand nr.1
    z  <= not(i2 and i3);    -- nand nr.2
  end comb_1;
comb_2: process(z)
  begin
    o1 <= z;
  end comb_2;
end behv;
```

Man sieht: In der "entity" wird eine Portliste vereinbart, die alle nach außen führenden Signale beinhalten muss. Interne Signale werden wie oben gezeigt vereinbart. Üblicherweise führen diese nach der Synthese zu realen "Drähten" in der Schaltung, können aber auch, wie in diesem Fall "z", wegoptimiert werden. Besonderheit: Signale, die den Block verlassen, können *nicht* in der "architecture" verschaltet werden. Das heißt, alle in der entity als "out" deklarierten Signale können nirgends in der "sensitivity list" eines Prozesses oder als Zuweisungswert erscheinen. Sie sollen ebenfalls nur in einem einzigen Prozess des VHDL-Files zugewiesen werden, so wie ein "Draht" ebenfalls zu einem Zeitpunkt immer nur einen Pegel führen kann.

Ein File mit einer Entity und einer Architecture ist bereits eine kompilierbare, vollständige VHDL-Komponente. Ähnlich wie auf einer Platine können auch mehrere VHDL-Bauteile miteinander quasi verdrahtet werden. In diesem Fall wird in einem anderen oder auch gleichen VHDL-File diese Komponente als "component" eingebunden.

Beispiel einer "Component"-Deklaration:

```
component 74LS00
  port (i0,i1,i2,i3 : in bit;
        o0,o1       : out bit);
end component;
```

Beispiel für eine "Component"-Instanziierung:

```
architecture behv of test is
  -- signal deklarierungen:
  signal a,b,c,d : bit;
  -- componenten declarierungen:
  component adder
    port(a,b       : in bit;
         sum,carry : out bit);
  end component;
begin
  teil_a: adder port map (a,b,c,d);  -- Komponente "adder" hat den Namen "teil_a"
                                     -- a,b,c,d ist angeschlossen
end behv;
```

0.2 Basis-Konstrukte

Nach dieser kurzen Einführung über die grundsätzlichen Gedanken von VHDL sollen im Folgenden in alphabetischer Reihenfolge die sprachlichen Basis-Konstrukte beschrieben werden:

0.2.1 aggregates

Ein Aggregat ist ein Klammerausdruck, der mehrere Einzelelemente zu einem Vektor zusammenfasst, wobei die Elemente durch Kommata getrennt werden.

(wert_1,wert_2,...)

(element_1 => wert_1,element_2 => wert_2,...)

Beispiele:

```
signal databus : bit_vector(3 downto 0);
signal d1,d2,d3,d4 : bit;
...
databus <= (d1,d2,d3,d4);
```

identisch mit:

```
databus(3) <= d1;
databus(2) <= d2;
databus(1) <= d3;
databus(0) <= d4;
```

```
--
type zustand is (idle,run,warte,aktion);  -- enumerated type
signal state : zustand;
--
type packet is record
  flag   : std_logic;
  nummer : integer range 0 to 7;
  daten  : std_logic_vector(3 downto 0);
end record;
signal x : packet;
...
x <= ('1',3,"0011");
--
type vierbit is array(3 downto 0) of std_ulogic;
type speicher is array(0 to 7) of vierbit;
variable xmem : speicher := (others=>'0');   -- mit '0' vorbelegen
--
variable dbus : std_logic_vector(15 downto 0) := (others=>'Z');
```

0.2.2 alias

Ein Alias bezeichet einen Teil eines Signals.

```
alias ”alias_name” : ”alias_type”(range) is ”signal_name”(range);
```

Beispiel:

```
signal daten_in : bit_vector(11 downto 0);
alias opcode : bit_vector(3 downto 0) is daten_in(3 downto 0);
alias daten  : bit_vector(7 downto 0) is daten_in(11 downto 4);
```

0.2.3 architecture

Eine Designeinheit in VHDL, die das Verhalten oder die Struktur einer Entity beschreibt.

```
architecture ”architecture_name” of ”entity_name” is
  declarations
begin
  concurrent statements
end architecture;
```

0.2.4 arrays

```
type ”type_name” is array (range) of ”element_type”;
```

Beispiele:

```
type Speicher is array (0 to 1023) of integer range 0 to 255;
signal sram : speicher;
type mem8 is array (natural range <>) of std_logic_vector(7 downto 0);
signal sram8_1024 : mem8(1023 downto 0);
signal sram8_192  : mem8(191 downto 0);
```

0.2.5 assert

```
assert ”bedingung” report ”string” severity ”severity_level”;
```

Überwache, dass *bedingung* erfüllt ist, wenn NICHT report *string* mit severity *severity_level*

```
severity_level ist "error" (default), "note", "warning" oder "failure"
```

Beispiel:

```
assert (a > c) report "a muss grösser c sein" severity note;
assert (true)  report "diese Meldung wird nie ausgegeben" severity note;
assert (false) report "diese Meldung wird immer ausgegeben" severity note;
```

0.2.6 attributes

Beispiele:

```
  signal'LEFT       signal(7) bei std_logic_vector(7 downto 0);
                    signal(0) bei std_logic_vector(0 to 7);
  signal'RIGHT      signal(0) bei std_logic_vector(7 downto 0);
                    signal(7) bei std_logic_vector(0 to 7);
  signal'HIGH       signal(7) bei std_logic_vector(7 downto 0);
                    signal(7) bei std_logic_vector(0 to 7);
  signal'LOW        signal(0) bei std_logic_vector(7 downto 0);
                    signal(0) bei std_logic_vector(0 to 7);
  signal'RANGE           7 downto 0   bei std_logic_vector(7 downto 0);
                         0 to 7       bei std_logic_vector(0 to 7);
  signal'REVERSE_RANGE   0 to 7       bei std_logic_vector(7 downto 0);
                         7 downto 0   bei std_logic_vector(0 to 7);
  signal'LENGTH          8            bei std_logic_vector(7 downto 0);
                         8            bei std_logic_vector(0 to 7);
  signal'EVENT      if (clk'event and clk='1') then
```

0.2.7 block statements

```
"block_name": block
  declarations
begin
  concurrent statements
end block;
```

0.2.8 case

```
case "expression" is
  when "fall_1" => "sequential statement"
  when "fall_2" => "sequential statement"
  when others     => "sequential statement"
```

```
end case;
```

Beispiel1:

```
case wert is
  when 0      => w <= '1';
  when 1      => w <= '0';
  when 2 | 3  => w <= a;
  when 4 to 7 => w <= b;
  when others => w <= 'X';
end case;
```

Beispiel2:

```
wert <= '0';
case din is
  when "00"   => wert <= '1';
  when others => null;          -- "when others" soll immer vorhanden sein
end case;                       -- durch default-Zuweisung is "null"-statement
 möglich!
```

Beispiel3:

```
    type state_type is ( IDLE, DO_SOMETHING );
    ...
    case state is
      when IDLE         => tx_line <= '0';
      when DO_SOMETHING => tx_line <= '1';
      when others =>
        assert false report "case defaulted!" severity failure;
     end case;
```

0.2.9 component declaration

Deklaration zur Festlegung des Namens und der Schnittstelle einer Komponente, die einer Entitydeklaration und Architecture zugeordnet sein muss.

```
component "component_name"
  generic ("generic_liste");
  port    ("port_liste");
end component;
```

0.2.10 component instantiation

```
label: "component_name"
generic map ( "generic1" => " generic1_entity" )
```

```
port map (
          ”component_port1” => ”entity_port1”,
          ”component_port2” => ”entity_port2”,
          ...
          ”component_portx” => ”entity_portx”
          );
```

0.2.11 constant

```
constant ”constant_name” : type := value;
```

Beispiel:

```
constant festwert : std_logic_vector(7 downto 0) := "10101100";
constant zeitwert : time := 50 ns;
type rdatum is array (0 to 3) of bit_vector(7 downto 0);
constant rom : rdatum :=
  ("00000001",
   "00000010",
   "00000011",
   "00000100");
```

0.2.12 entity

Eine Struktureinheit in einem VHDL- Entwurfssystem. Beschreibt die Schnittstellen eines VHDL-Funktionsblocks nach außen. Mit Hilfe von Port-Anweisungen erfolgt die Deklaration der Anschlüsse innerhalb der Entity. Zu jeder Entity gehört eine Architecture.

```
entity ”entity_name” is
  generic (generic_list);
  port    (port_list);
end ”entity_name”;
```

0.2.13 exit-Anweisung

mit der exit-Anweisung wird die "innerste" Schleife verlassen und mit der Anweisung, die direkt auf die Schleife folgt, fortgefahren.

Beispiel: Bestimmen der Anzahl der führenden Nullen

```
for i in signal'range loop
 exit when signal(i)='1';
 null_v := null_v + 1;
end loop;
```

0.2.14 file declaration

Beispiel:

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_textio.all;
library std;
  use std.textio.all;
```

```
architecture sim of dut is
  file my_file : text open write_mode is "my_file.dat";
begin
  proc: process(clk)
    variable outline : line;
    variable counter : integer := 0;
  begin
    if rising_edge(clk) then
      write(outline, string'("Takt: "));
      write(outline, counter);
      writeline(my_file, outline);
      counter := counter + 1;
    end if;
  end process proc;
end sim;
```

0.2.15 for loop

```
"eventuell_label": for "parameter" in "range" loop
  sequential statements
end loop "eventuell_label";
```

Beispiel:

```
for i in 0 to 7 loop
  w(i) <= a(i) and b;   -- 8bit bus "a" mit Einzelsignal "b" verunden
end loop;
```

0.2.16 functions

Eine der beiden Möglichkeiten in VHDL, Code mittels eines einfachen Aufrufmechanismus wiederverwertbar zu machen. Functions werden gewöhnlich mit ihrem Namen und einer in Klammern stehenden Liste der Eingangsparameter aufgerufen und können nur ein Ausgangsargument liefern- vgl. auch Procedure.

```
function ”funktion_name” (parameter_list) return ”type” is
  declarations
begin
  sequential statements
end function_name;
```

Beispiel:

```
function parity_generator (din : std_ulogic_vector)
                           return std_ulogic is
  variable t : std_ulogic := ’0’;   -- variable mit default Zuweisung
begin
  for i in din’range loop           -- ganze Busbreite
     t := t xor din(i);
  end loop;
  return t;
end parity_generator;
```

Aufruf der Funktion als "concurrent" oder "sequential statement":

```
  sig_pary <= paritiy_generator(data_bus);
```

Achtung: keine "signal assignments" oder "wait"

0.2.17 generate

```
”label”: for ”parameters” in ”range” generate
  concurrent statements
end generate ”label”;
```

oder

```
”label”: if ”condition” generate
  concurrent statements
end generate ”label”;
```

Beispiel:

```
architecture gen of test is
  component volladdierer
    port (x,y,ci : in bit;
          s,co   : out bit);
  end component;
```

```
  component halbaddierer
    port (x,y  : in bit;
          s,co : out bit;
  end component;
  signal carry : bit_vector(0 to 7);
begin
  gen_addierer: for i in 0 to 7 generate
     niedrigstes_bit: if i=0 generate
       w0: entity halbaddierer port map
           (x(i),y(i),s(i),carry(i));
     end generate niedrigstes_bit;
     hoeheres_bit: if i>0 generate
       wi: entity volladdierer port map
           (x(i),y(i),carry(i-1),s(i),carry(i));
     end generate hoeheres_bit;
  end generate gen_addierer;
  co <= carry(7);
end gen;
```

Beispiel2:

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;
...
    dft_mem_in      : in  std_logic_vector(const1 downto 0);
    dft_mem_out     : out std_logic_vector(const2 downto 0);
    pwr_mem_ctrl_in : in  std_logic_vector(3 downto 0)
...
  gen0: for i in dft_mem_out'range generate
    signal tmp : std_logic_vector(dft_mem_out'range);
  begin
    tmp <= std_logic_vector(resize(unsigned(dft_mem_in), dft_mem_out'length));
    dft_mem_out(i) <= tmp(i) xor (pwr_mem_ctrl_in(0) xor pwr_mem_ctrl_in(1) xor
 pwr_mem_ctrl_in(2) xor pwr_mem_ctrl_in(3));
  end generate gen0;
```

0.2.18 generic

```
entity "entity_name" is
  generic (generic_list)
  port    (port_list)
end "entity_name";
```

Beispiel: (wichtig für skalierbare Blöcke!!)

```
 entity test is
   generic (n : integer := 15); -- hierbei ist 15 der Default-Wert, falls kein
Generic
                               -- bei der Initialisierung angegeben wird
   port    (a : in std_ulogic_vector(n-1 downto 0));
 end test;
```

0.2.19 if

```
if    condition_a then
  sequential statements
elsif condition_b then
  sequential statements
else
  sequential statements
end if;
```

Beispiel:

```
if nreset='0' then
   count <= 0;
elsif clk'event and clk='1' then
  if count=9 then
    count <= 0;
  else
    count <= count+1;
  end if;
end if;
```

0.2.20 library

0.2.21 names

0.2.22 next-Anweisung

Die next-Anweisung beendet den aktuellen Schleifendurchlauf vorzeitig; das bedeutet, dass die Anweisungen bis zur end-loop-Anweisung übersprungen werden und mit dem nächsten Schleifendurchlauf fortgefahren wird.

Beispiel: Bestimmen der Anzahl der Nullen in einem Vektor

```
for i in signal'range loop
 next when signal(i)='1';
 null_v := null_v + 1;
end loop;
```

0.2.23 notations

```
hex_var := 16#8001#;
binary_s <= b"000_111_010";
```

```
octal_s  <= o"207";
hex_s    <= x"01_FB";
```

0.2.24 null statement

Falls durch die Syntax ein Statement erforderlich ist, kann das "Null"-Statement verwendet werden, um anzuzeigen, dass nichts zu tun ist. Vgl. hierzu beispielsweise 'when others => null;' einer case-Anweisung.

```
proc: process(clk)
begin
  if rising_edge(clk) then
    case select is
      when "00" => reg <= '1';
      when "11" => reg <= '0';
      when others => null;
    end case;
  end if;
end process proc;
```

0.2.25 operators

Logische Operatoren für Typen: bit,boolean,bit_vector,std_logic,std_logic_vector

```
and      -- und
or       -- oder
nand     -- nicht und
nor      -- nicht oder
xor      -- exclusive oder
not      -- nicht (invertierung)
```

Vergleichs Operatoren Ergebnis: boolean

```
=        -- Gleichheit
/=       -- Ungleichheit
<        -- kleiner
>        -- groesser
<=       -- kleiner gleich (Achtung bei Type int: Speichern)
>=       -- grösser gleich
```

Arithmetische Operatoren für Typen: integer,real

```
a  <= a + 7;       -- Addition
r1 <= r2 - 3.1414  -- Subtraktion (real)
m  <= x * y        -- Multiplikation
```

```
d  <= m / 2        -- Division
```

VHDL93:

```
sll   -- shift left  logical
srl   -- shift right logical
sla   -- shift left arith.
sra   -- shift right arith.
rol   -- rotate left
ror   -- rotate right
```

0.2.26 package

```
package ”package_name” is
  declarations
end package;
```

Beispiele:

```
package demo is
  constant nullwert : bit_vector := "00000000";
  function foo ( v : std_ulogic ) return std_ulogic;
  component adder          -- Dessen Implementierung ist vielliecht in
 irgendeiner Library vorcompiliert,
    port(x,y,ci : in bit;  -- da ein Component niemals in der package body
 definiert werden kann.
         s,co   : out bit);
  end component;
end demo;
package body demo is
  function foo ( v : std_ulogic ) return std_ulogic is
  begin
    return v;
  end function;
end package body;
```

Package-Aufruf lautet dann:

```
use work.demo.all;
 entity xx is
 port
   ( wert : out bit_vector(7 downto 0));
```

```
 end xx;

 architecture behv of xx is
 begin
   wert <= nullwert;
 end behv;
```

0.2.27 procedures

```
procedure ”procedure_name” (paramter_list) is
  declarations
begin
  sequential statements
end ”procedure_name”;
```

Beispiel:

```
procedure parity_generator
  (signal din : in  std_ulogic_vector;
   signal par : out std_ulogic) is
variable t : std_ulogic := '0';
begin
  for i in 0 to din'range loop
    t := t xor din(i);
  end loop;
  par <= t;
end parity_generator;
```

Proceduren in Packages:

```
package my_procedures is
  procedure parity_generator
    (signal din : in  std_ulogic_vector;
     signal par : out std_ulogic);
end my_procedures;
package body my_procedures is
  procedure parity... (procedure code siehe oben!)
end my_procedures;
```

Aufruf:

```
...
parity_generator(databus,par_bit);
```

...

0.2.28 process

```
"optionales_label": process (optionale sensitivity liste)
  declarations
begin
  sequential statements;
end process optionales_label;
```

0.2.29 records

```
  type my_record_t is record
    element1 : std_logic;
    element2 : std_logic;
    element3 : std_logic_vector(1 downto 0);
    element4 : std_logic_vector(4 downto 0);
  end record;
  type my_array_t is array (3 downto 0) of my_record_t;



  signal my_signal_s : my_array_t;
  my_signal_s <= (others => ('0','0',(others => '0'),(others => '0')));
```

0.2.30 type conversion

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

```
  std_logic_vector   ->   unsigned          | unsigned(arg)
  std_logic_vector   ->   signed            | signed(arg)
  std_logic_vector   ->   integer           |
  std_logic_vector->signed/unsigned->integer
  integer            ->   unsigned          | to_unsigned(arg,size)
  integer            ->   signed            | to_signed(arg,size)
  integer            ->   std_logic_vector  |
  integer->signed/unsigned->std_logic_vector
  unsigned           ->   std_logic_vector  | std_logic_vector(arg)
  unsigned           ->   integer           | to_integer(arg)
  signed             ->   std_logic_vector  | std_logic_vector(arg)
  signed             ->   integer           | to_integer(arg)
```

```
unsigned          ->   unsigned          | resize(arg,size)
signed            ->   signed            | resize(arg,size)
```

```
std_ulogic        ->   bit               | to_bit(arg)
std_ulogic_vector ->   bit_vector        | to_bitvector(arg)
std_ulogic_vector ->   std_logic_vector  | to_stdlogicvector(arg)
bit               ->   std_ulogic        | to_stdulogic(arg)
bit_vector        ->   std_logic_vector  | to_stdlogicvector(arg)
bit_vector        ->   std_ulogic_vector | to_stdulogicvector(arg)
std_logic_vector  ->   std_ulogic_vector | to_stdulogicvector(arg)
std_logic_vector  ->   bit_vector        | to_bitvector(arg)
```

Beispiele:

```
signal value_i   : integer range 0 to 15;
signal value_u   : unsigned(3 downto 0);
signal value_slv : std_logic_vector(3 downto 0);
...
value_i   <= 14;
value_u   <= to_unsigned(value_i, value_u'length); -- conversion
value_slv <= std_logic_vector(value_u);            -- cast (types are closely
related)
```

0.2.31 type declaration

```
type <memory_type> is array(0 to 9) of std_logic_vector(7 downto 0);
type <fsm_type> is (idle, run, ready);
type <hour_range_type> is range 1 to 12;
subtype <vector_type> is std_logic_vector(5 downto 0);
```

0.2.32 use

Verwendung von Bibliotheken, oder Teilen daraus.

Beispiel:

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

0.2.33 variable declaration

```
variable "Variablenname" : "Typ" := "Initialisierungswert"
```

Beispiel:

```
variable MeinBool : boolean := false;
```

0.2.34 variable assignment

```
"Variablenname" := "Wert";
```

0.2.35 wait

- wait until *condition*
- wait on *signal list*
- wait for *time* -- nicht synthetisierbar
- wait;

Beispiel:

```
wait until din="0010";
```

oder:

```
stimulus: process
begin

  loop
    clk <= '0';
    wait for 50 ns;
    clk <= '1';
    wait for 50 ns;
  end loop;

end process;
```

0.2.36 when

signal_name <= *expression* when *condition* else
expression when *condition* else
expression;

Beispiel (Tristate Port):

```
data <= data_out when data_enable = '1' else 'Z';
```

0.2.37 while

```
while condition loop
  sequential statements
end loop;
```

Beispiel (Register rechts schieben):

```
if clk'event and clk='1' then
  i:=0;
  while (i<7) loop
    buswert(i) <= buswert(i+1);
    i:=i+1;
  end loop;
  buswert(7) <= din;
end if;
```

0.2.38 with select

```
with select_signal select
  dst_signal <= src_signal when select_value1,
                src_signal when select_value2,
                src_signal when others;
```

Selektive Signalzuweisung außerhalb eines Prozesses, als Alternative zu case:

```
signal sel_s                : std_logic_vector(1 downto 0);
signal monitor_s,one_s,two_s : std_logic_vector(3 downto 0);
```

```
with sel_s select
  monitor_s <= one_s when "00",
               two_s when "11",
               "0000" when others;
```

0.3 Operator precedence

VHDL operators in order of precedence (Highest first)

Note: The concatenate operator & has a lower order of precedence than some arithmetic operators +/-

**

abs

not

*

/

mod

rem

+

-

+

-

```
&
sll
srl
sla
sra
rol
ror
=
/=
<
<=
>
>=
and
or
nand
nor
xor
xnor
```

0.4 Beispielprogramme

0.4.1 Zustandsmaschinen

Die im Folgenden beschriebene Codierung einer Zustandsmaschine erhebt nicht den Anspruch die eleganteste Lösung darzustellen, sie soll vielmehr einen Eindruck der Sprache und der grundsätzlichen Abläufe vermitteln.

```
architecture demo of zustandsmaschine is
  constant vectorbreite : integer := 3;

  type zustandsvector is std_logic_vector(vectorbreite downto 0);
  signal zustand : zustandsvector;
  -- "one hot" Codierung !!
  constant warte_zustand   : zustandsvector := "0001";
  constant a_zustand       : zustandsvector := "0010";
  constant b_zustand       : zustandsvector := "0100";
  constant c_zustand       : zustandsvector := "1000";
  signal gehe_zu_warte_zustand : std_logic;
  signal gehe_zu_a_zustand     : std_logic;
  signal gehe_zu_b_zustand     : std_logic;
  signal gehe_zu_c_zustand     : std_logic;
  signal timer : std_logic_vector(3 downto 0);
begin
-------------------------------------------
zustaende : process(nres,clk)
```

```
begin
   if    (nres='0') then
       zustand <= warte_zustand;
   elsif (clk'event and clk='1') then
     if    (gehe_zu_warte_zustand='1') then
       zustand <= warte_zustand;
     elsif (gehe_zu_a_zustand='1') then
       zustand <= a_zustand;
     elsif (gehe_zu_b_zustand='1') then
       zustand <= b_zustand;
     elsif (gehe_zu_c_zustand='1') then
       zustand <= c_zustand;
     else                      -- der "else" Pfad ist optional!
       zustand <= zustand;     -- ohne diese Zeilen identische Funktion!!
     end if;
end process zustaende;
-------------------------------------------
zustandsweiterschaltung : process(eingang1,eingang2,zustand,timer)
begin
  gehe_zu_warte_zustand <= '0';   -- default assignments
  gehe_zu_a_zustand     <= '0';
  gehe_zu_b_zustand     <= '0';
  gehe_zu_c_zustand     <= '0';
  case zustand is
    when warte_zustand =>
                          if    (eingang1='1') then    -- Priorisierung von
                            gehe_zu_a_zustand <= '1';  -- "eingang1" und
 "eingang2"
                          elsif (eingang2='1') then
                            gehe_zu_c_zustand <= '1';
                          end if;
    when a_zustand     =>
                          if (timer="0000") then         -- Maschine bleibt im
 "a_zustand"
                            gehe_zu_b_zustand <= '1';    -- bis Timer abgelaufen
                          end if;
    when b_zustand     =>
                          gehe_zu_warte_zustand <= '1';
    when c_zustand     =>
                          gehe_zu_a_zustand <= '1';
    when others        =>                                -- diese Zeile sichert
 Vollständigkeit
                          gehe_zu_warte_zustand <= '1';   -- der Auscodierung!!!
  end case;
end process zustandsweiterschaltung;
-------------------------------------------
wartezeit : process (nres,clk)
begin
```

```
  if    (nres='0') then
      timer <= "1010";
  elsif (clk'event and clk='1') then
      if (zustand=a_zustand) then     -- Zahler steht auf "1010"
         timer <= timer-1;            -- nur im "a_zustand" läuft er rückwärts
      else
         timer <= "1010";
      end if;
  end if;
end process wartezeit;
-------------------------------------------
-------------------------------------------
getakteter_ausgang : process (nres,clk)
begin
  if    (nres='0') then
      ausgang1 <= '0';
  elsif (clk'event and clk='1') then
      if (zustand=b_zustand) then
         ausgang1 <= '1';          -- eine "clk"-Periode langer Pulse
      else                         -- Achtung: erscheint einen Takt nach
 "b_zustand"!!
         ausgang1 <= '0';
      end if;
  end if;
end process getakteter_ausgang;
-------------------------------------------
-------------------------------------------
asynchroner_ausgang : process (gehe_zu_b_zustand)
begin
  ausgang2 <= gehe_zu_b_zustand;    -- eine "clk"-Periode langer Pulse (könnte
 spiken!!)
end process asynchroner_ausgang;
-------------------------------------------
-------------------------------------------
```

0.5 Fehlervermeidung

0.5.1 Unbeabsichtiges Erzeugen eines "Latch"

In der Synthese wird unbeabsichtigt ein "Latch" implementiert. Die Ursache ist meist, dass in einem kombinatorischen Prozess die Zuweisungen auf ein Signal nicht vollständig auscodiert wurden:

Beispiel:

```
if    (a='1') and (b='0') then
    x <= '1';
elsif (a='0') and (b='1') then
    x <= '0';
end if;
```

Die Fälle a=1 und b=1 ebenso wie a=0 und b=0 wurden nicht definiert. Die Folge ist, dass die Synthese versucht in diesen Fällen den aktuellen Zustand beizubehalten. Dies erfolgt durch die Implementierung eines "Latch".

0.5.2 Vergessen von Signalen in der "sensitivity list" eines kombinatorischen Prozesses

Die Folge ist ein Verhalten in der Simulation das von der realen Gatterschaltung abweicht! Es gibt vhdl-Editoren die die "sensitivity list" prüfen. Spätestens in der Synthese erscheint eine Warnmeldung.

EN:PROGRAMMABLE LOGIC/VHDL[2] FR:CONCEPTION ET VHDL[3]

0.5.3 Verwenden von Reset-Signalen in einem Design

Asynchrone Reset-Signale gehören bei VHDL-Designs unbedingt synchronisiert:

```
library      ieee;
use  ieee.std_logic_1164.all;
entity myEnt is
  port(
    rst_an : in std_logic;
    clk: in std_logic;
    rst: in std_logic;
    sigIn: in std_logic_vector(3 downto 0);
    sigOut: out std_logic_vector(3 downto 0));
end entity myEnt;
architecture myArch of myEnt is
  signal sync_rst_r : std_logic_vector(1 downto 0);
  signal mySig: std_logic_vector(sigIn'range);
begin
  process(clk)
  begin
    if (rising_edge(clk)) then
      sync_rst_r <= sync_rst_r(0) & rst_an;
    end if;
```

2 HTTP://EN.WIKIBOOKS.ORG/WIKI/PROGRAMMABLE%20LOGIC%2FVHDL
3 HTTP://FR.WIKIBOOKS.ORG/WIKI/CONCEPTION%20ET%20VHDL

```
  end process;
  process(clk, sync_rst_r)
  begin
    if sync_rst_r(1) = '0' then
        mySig <= (others => '0');
    elsif (rising_edge(clk)) then
      if (rst = '1') then
        mySig <= (others => '0');
      else
        mySig <= sigIn;
      end if;
    end if;
  end process;
sigOut <= mySig;
end architecture;
```

Für detailierte Information zum Thema Reset im FPGAs siehe das Whitepaper von Xilinx "Get Smart About Reset: Think Local, Not Global" (englischsprachig).

0.5.4 Generieren von "Clock"-Signalen in einem Design

Die "Clock"-Signale werden mit Hilfe einer speziellen Verdrahtung auf dem FPGA verteilt. Ein "Clock"-Signal darf nie durch Logik erzeugt werden. Im Folgenden wird ein schlechtes Beispiel gezeigt, in welchem ein neues Clock Signal erzeugt wird:

```
library ieee;
use ieee.std_logic_1164.all;
entity clock_divider_ent is
port (
    clk               : in    std_logic;
    clkDiv            : out   std_logic);

end clock_divider_ent;
architecture synth of clock_divider_ent is

    signal counter:   integer range 1023 downto 0;
    signal clkDivInt: std_logic := '0';

begin
  process(clk)
  begin
      if (rising_edge(clk)) then
          if (counter = 0) then
              counter <= 1023;
              clkDivInt <= not clkDivInt;
          else
              counter <= counter - 1;
```

```
            end if;
        end if;
    end process;
end synth;
```

In diesem Beispiel wird eine bessere Implementierung für den oben gezeigten Block dargestellt. Das Signal clkDivInt wird hier zu einem kurzen Puls: jedes Mal wenn der Zähler den Wert null erreicht, bleibt dieser Puls hoch nur während eines einzigen Taktzyklus vom clk.

```
library ieee;
use ieee.std_logic_1164.all;
architecture synth of clock_divider_ent is

    signal counter:   integer range 2047 downto 0;
    signal clkDivInt: std_logic := '0';

begin
  process(clk)
  begin
      if (rising_edge(clk)) then
          counter <= counter - 1;
          clkDivInt <= '0';
          if (counter = 0) then
              counter   <= 2047;
              clkDivInt <= '1';
              end if;
          end if;
      end process;
end synth;
```

```
...
process(clk)
begin
  if(rising_edge(clk) then
    if (clkDiv = '1') then
      ...
    end if;
  end if;
end process;
...
```

0.5.5 Vergleich von Zahlen

Nach Möglichkeit sollte immer auf "=" und nicht auf "<", ">", "<=" oder ">=" verglichen werden, da diese aufwendiger in Hardware zu implementieren sind.

0.5.6 std_logic_arith vs numeric_std

std_logic_arith wurde nicht standarisiert und ist nicht überall gleich implementiert.
Stattdessen, nur **numeric_std** verwenden:

```
library ieee;
use ieee.std_logic_arith.all;  -- Vermeiden, nicht standarisiert.
use ieee.numeric_std.all;      -- Die beiden zusammen gibt nur Ärger.
```

1 Autoren

Edits	User
2	Brunowe[1]
1	Gagosoft[2]
1	Jakob.wilhelm[3]
1	Klaus Eifert[4]
3	MichaelFrey[5]
7	Rdiez[6]
7	Robertix[7]
1	Stefan Majewsky[8]
4	Walliatwiki[9]

1 http://de.wikibooks.org/w/index.php?title=Benutzer:Brunowe
2 http://de.wikibooks.org/w/index.php?title=Benutzer:Gagosoft
3 http://de.wikibooks.org/w/index.php?title=Benutzer:Jakob.wilhelm
4 http://de.wikibooks.org/w/index.php?title=Benutzer:Klaus_Eifert
5 http://de.wikibooks.org/w/index.php?title=Benutzer:MichaelFrey
6 http://de.wikibooks.org/w/index.php?title=Benutzer:Rdiez
7 http://de.wikibooks.org/w/index.php?title=Benutzer:Robertix
8 http://de.wikibooks.org/w/index.php?title=Benutzer:Stefan_Majewsky
9 http://de.wikibooks.org/w/index.php?title=Benutzer:Walliatwiki

Abbildungsverzeichnis

- GFDL: Gnu Free Documentation License. http://www.gnu.org/licenses/fdl.html
- cc-by-sa-3.0: Creative Commons Attribution ShareAlike 3.0 License. http://creativecommons.org/licenses/by-sa/3.0/
- cc-by-sa-2.5: Creative Commons Attribution ShareAlike 2.5 License. http://creativecommons.org/licenses/by-sa/2.5/
- cc-by-sa-2.0: Creative Commons Attribution ShareAlike 2.0 License. http://creativecommons.org/licenses/by-sa/2.0/
- cc-by-sa-1.0: Creative Commons Attribution ShareAlike 1.0 License. http://creativecommons.org/licenses/by-sa/1.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/
- cc-by-2.0: Creative Commons Attribution 2.0 License. http://creativecommons.org/licenses/by/2.0/deed.en
- cc-by-2.5: Creative Commons Attribution 2.5 License. http://creativecommons.org/licenses/by/2.5/deed.en
- cc-by-3.0: Creative Commons Attribution 3.0 License. http://creativecommons.org/licenses/by/3.0/deed.en
- GPL: GNU General Public License. http://www.gnu.org/licenses/gpl-2.0.txt
- PD: This image is in the public domain.
- ATTR: The copyright holder of this file allows anyone to use it for any purpose, provided that the copyright holder is properly attributed. Redistribution, derivative work, commercial use, and all other use is permitted.
- EURO: This is the common (reverse) face of a euro coin. The copyright on the design of the common face of the euro coins belongs to the European Commission. Authorised is reproduction in a format without relief (drawings, paintings, films) provided they are not detrimental to the image of the euro.
- LFK: Lizenz Freie Kunst. http://artlibre.org/licence/lal/de
- CFR: Copyright free use.
- EPL: Eclipse Public License. http://www.eclipse.org/org/documents/epl-v10.php

1		