Digitaldesign mit VHDL

Prof. H. W. Wagner

Digitaldesign mit VHDL

Ablauf

2 SWS Vorlesung in den ersten Semesterwochen anschließend Belegarbeit

2 SWS Praktikum

Abschluss - benotete Belegarbeit

Digitaldesign mit VHDL

Literatur

Molitor	VHDL eine Einführung	Pearson
Zwolinski	Digital System Design with VHDL	Pearson
Yalamanchili	Introductory VHDL	Pearson
Skahill	VHDL for program.	Addison

Digitaldesign mit VHDL

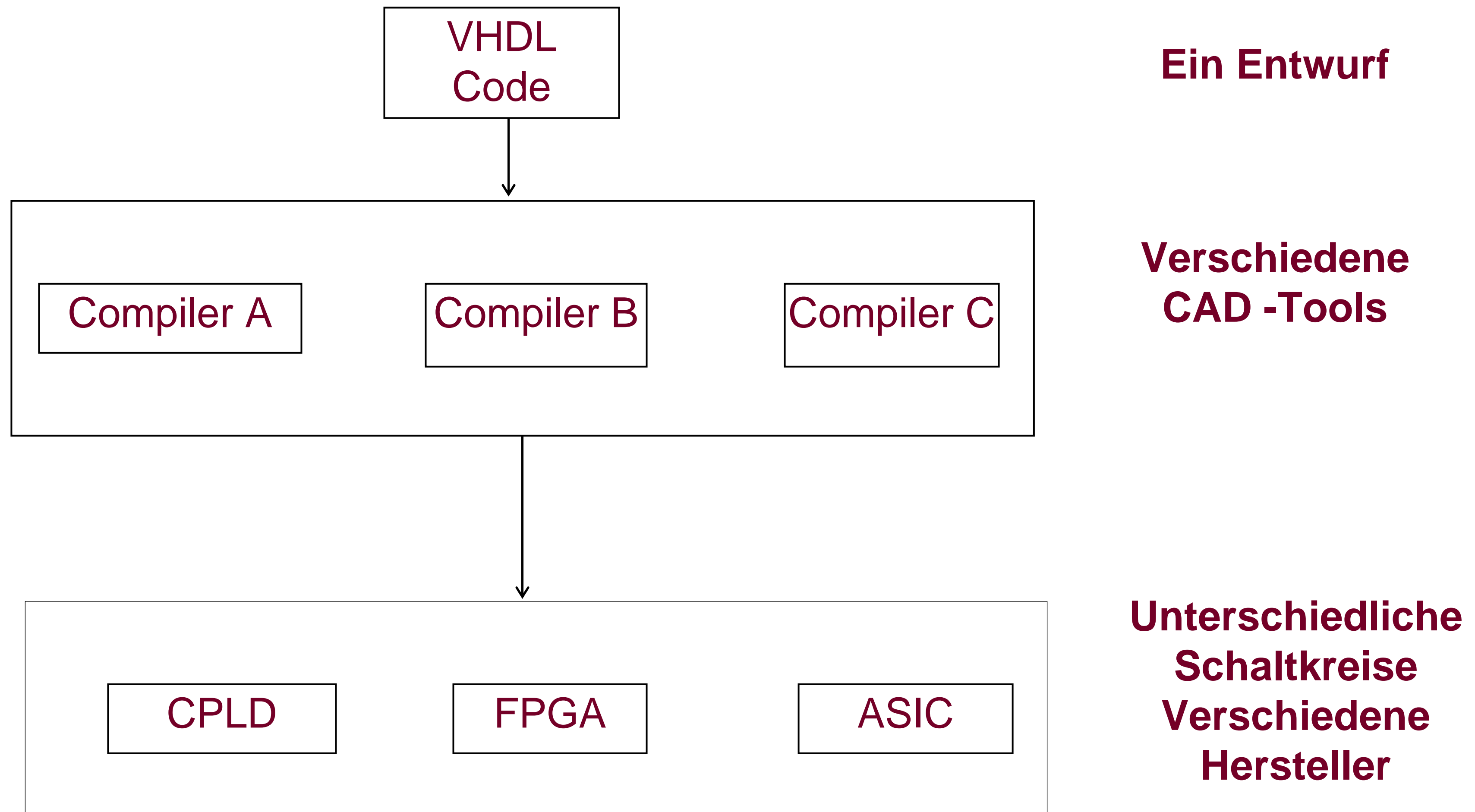
Literatur

Reichhardt	VHDL-Synthese	Oldenbourg
Jorke	Rechnergestützter Entwurf digitaler Schaltungen	Fachbuch-verlag Leipzig
Ten Hagen	Abstrakte Modellierung	Springer
Hamblen	Rapid Prototyping	Kluwer

Digitaldesign mit VHDL

VHDL ist einsetzbar für

- die Funktionsbeschreibung vorhandener digitaler Schaltkreise und Systeme
- die Simulation existierender bzw. zu entwerfender Systeme
- die Implementierung einer Funktionsbeschreibung in einen digitalen Schaltkreis


VHDL
Code
Ein Entwurf
Compiler A
Compiler B
Compiler C
Verschiedene
CAD -Tools
CPLD
FPGA
ASIC
Unterschiedliche
Schaltkreise
Verschiedene
Hersteller

Digitaldesign mit VHDL

Probleme beim Entwurf mit VHDL

- Der Einfluss auf die Implementierung der Funktion in der Gatterebene geht verloren.
- Die Realisierung der Logik durch automatische Schaltungssynthese kann uneffektiv sein.
- Die Syntheseergebnisse hängen sehr stark von der Qualität der CAD Tools ab.

Digitaldesign mit VHDL

Probleme beim Entwurf mit VHDL

- VHDL ist eine sehr umfangreiche Sprache
- Die Einarbeitungszeit für einen effektiven Einsatz ist sehr hoch
- Die Effektivität des Entwurfs hängt stark von den Erfahrungen des Bearbeiters ab

Digitaldesign mit VHDL

Beschreibung kombinatorischer Logik

- **Verhaltensbeschreibung**
- **Datenflussbeschreibung**
- **Strukturbeschreibung**

1. Design

```
ENTITY design1 IS
  PORT(
        xa,xb,xc        : IN       BIT;
        ya,yo           : OUT      BIT
        );
END design1;
```

1. Design

```
ARCHITECTURE adesign1 OF design1 IS
BEGIN
  ya<=xa and xb and xc;
  yo<=xa or xb or xc;
END adesign1;
```

Entity

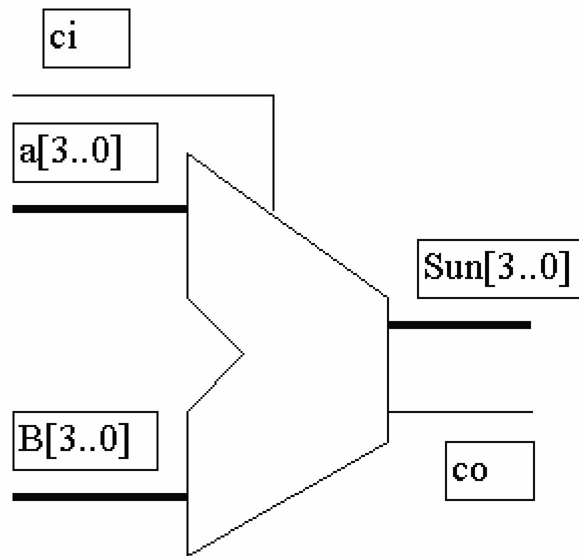
Die Entity Deklaration beschreibt die Ein- und Ausgänge einer Funktionseinheit.

Bsp.:

```
ENTITY adder4 IS
  PORT(
      a,b        : IN  STD_LOGIC_VECTOR(3 downto 0);
      ci         : IN  STD_LOGIC;
      sum        : OUT STD_LOGIC_VECTOR(3 downto 0);
      co         : OUT STD_LOGIC);
END adder4;
```

Diese Beschreibung ist äquivalent dem Symbol bei der Funktionsbeschreibung mit Hilfe von Schematics.

Entity



ENTITY

- **PORT ist ein Datenobjekt.**
- **Datenobjekten:**
  - **können Werte zugewiesen werden**
  - **können in Ausdrücken verwendet werden**
- **Alle Ports eines Entity bilden die PORT - Deklaration**
- **PORT wird beschrieben durch**
  - **Name bzw. Identifyer als erster Teil der Deklaration**
  - **mode bestimmt die Übertragungsrichtung**
  - **Typ**

ENTITY

Modes

- **Die Datenübertragungsrichtung wird durch mode angegeben. Die Übertragungsrichtung wird aus der Sicht des Entity festgelegt.**

ENTITY

Modes können sein:

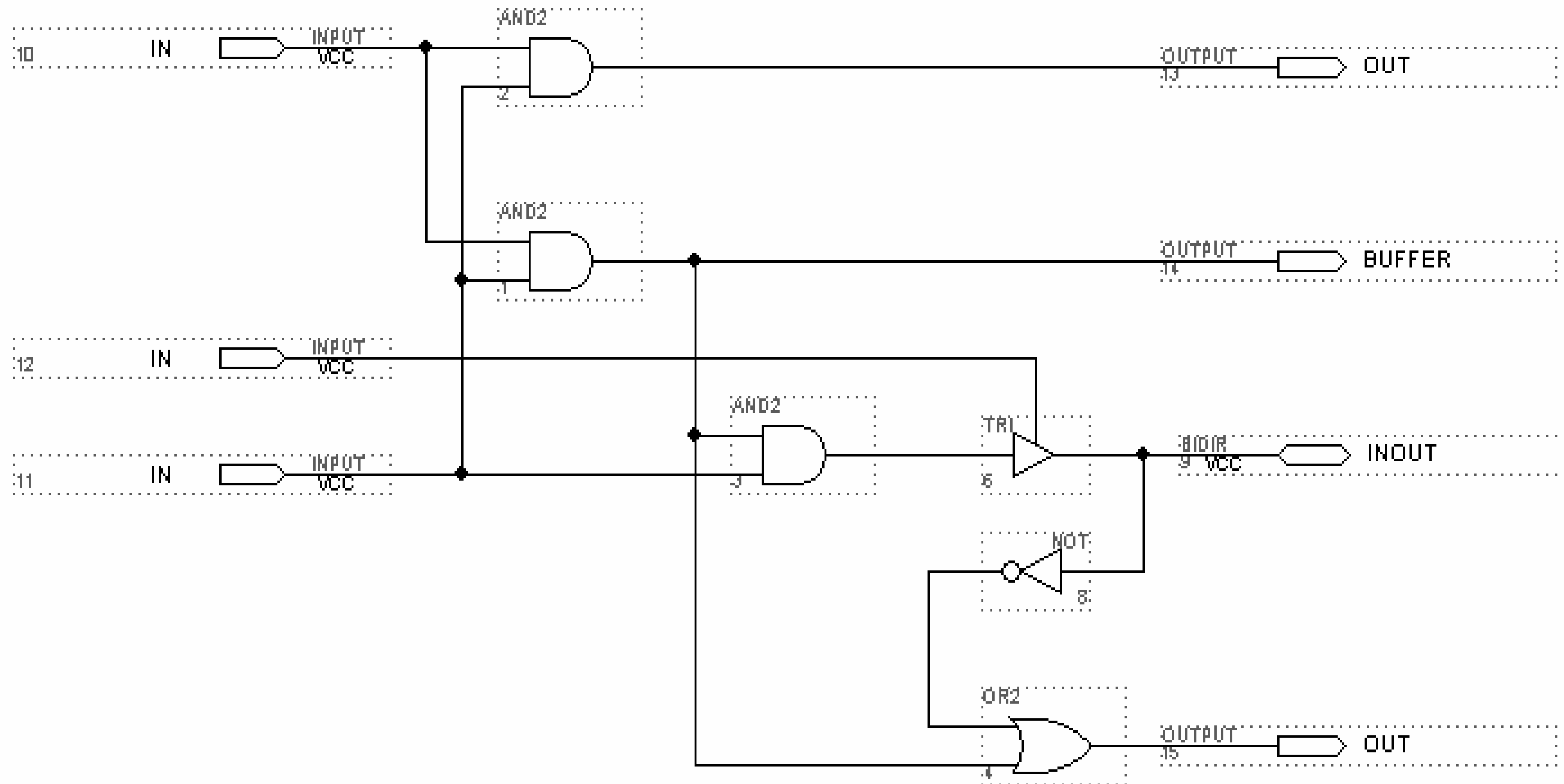
- **IN Der Datenfluß erfolgt von außen in das Entity. IN wird verwendet für unidirektionale Eingänge**
- **OUT Datenquelle liegt im Entity. Out kann nicht für Rückkopplung verwendet werden.**

ENTITY

Modes können sein:

- **BUFFER Buffer wird für Signale verwendet, die Ausgänge sind und auch intern genutzt werden.**
  - **Buffer kann nicht für bidirektionale Signale verwendet werden. Buffer können nicht zur Erhöhung der Treiberfähigkeit verwendet werden**
  - **Buffer können nur Intern und zum Anschluß an einen Buffer eines anderen Entity genutzt werden.**

ENTITY



ENTITY

Modes können sein:

- **INOUT wird für bidirektionale Signale verwendet. Inout wird auch für interne Rückkopplungen verwendet. Inout kann alle anderen Modes ersetzen.**

ENTITY

```
ENTITY __entity_name IS
GENERIC(__parameter_name : string :=  __default_value;
   __parameter_name : integer:=  __default_value);
   PORT(
   __input_name, __input_name : IN   STD_LOGIC;
   __input_vector_name  : IN
        STD_LOGIC_VECTOR(__high downto __low);
   __bidir_name, __bidir_name  : INOUT        STD_LOGIC;
   __output_name, __output_name      : OUT STD_LOGIC);
END __entity_name;
```

ENTITY

```
ARCHITECTURE a OF __entity_name IS
   SIGNAL __signal_name : STD_LOGIC;
   SIGNAL __signal_name : STD_LOGIC;
BEGIN
   --  Process Statement
   --  Concurrent Procedure Call
   --  Concurrent Signal Assignment
   --  Conditional Signal Assignment
   --  Selected Signal Assignment
   --  Component Instantiation Statement
   --  Generate Statement
END a;
```

Backus Naur Form BNF

entity_declaration::=
entity identifier **is**
entity_header
entity_declarative_part
[**begin**
entity_statement_part]
end [**entity**] [*entity* _simple_name] ;

Backus Naur Form BNF

identifier::=

basic_identifier | extended_identifier

basic_identifier ::=

letter { [underline] letter_or_digit }

extended_identifier ::= \ graphic_character { graphic_character } \

Backus Naur Form BNF

letter_or_digit ::=
letter | digit

letter ::=
upper_case_letter | lower_case_letter

extended_identifier ::=
**** graphic_character { graphic_character } ****

Backus Naur Form BNF

Comments

A basic identifier must not contain any blanks!

Examples

count

x

c_out

store_next_item

\74S05

\a\\b

Backus Naur Form BNF

graphic_character ::=
_basic_graphic_character_ |
lower_case_letter | other_special_character

basic_character ::=
_basic_graphic_character_ | format_effector

basic_graphic_character ::=
upper_case_letter | digit | special_character
| space_character

Backus Naur Form BNF

Specification

(a) upper_case_letter
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
(b) digit
0 1 2 3 4 5 6 7 8 9
(c) special_character
" # & ' () * + , - . / : ; < = > _ |
(d) space_character
(e) lower_case_letter
a b c d e f g h i j k l m n o p q r s t u v w x y z

Backus Naur Form BNF

Specification

(f) other_special_character

! $ % @ ? [\ ] ^ ` {} ~

(g) Format effectors (***format_effectors*** **) are**

Tab (horizontal and vertical),

Carriage return,

Line feed and

Form feed.

Backus Naur Form BNF

entity_header::=

[*formal* _generic_clause]

[*formal* _port_clause]

Backus Naur Form BNF

generic_clause ::=

generic (generic_list) ;

generic_list ::= *generic* _interface_list

Backus Naur Form BNF

port_clause ::=
port (port_list) ;

port_list ::= *port* _interface_list

interface_list ::=
interface_element { ; interface_element }

interface_element ::= interface_declaration

Backus Naur Form BNF

interface_declaration ::=
interface_constant_declaration
| interface_signal_declaration
| interface_variable_declaration
| interface_file_declaration

Backus Naur Form BNF

interface_signal_declaration ::=
[signal] identifier_list : [mode] subtype_indication [bus] [:= *static* _expression]

mode ::= in | out | inout | buffer | linkage

Backus Naur Form BNF

subtype_indication ::=

[*resolution_function* _name]

type_mark

[constraint]

type_mark ::=

***type* _name**

| *subtype* _name

Backus Naur Form BNF

constraint ::=
range_constraint
| index_constraint

range_constraint ::= range range

range ::=
***range* _attribute_name**
| simple_expression direction simple_expression

Backus Naur Form BNF

index_constraint ::= (discrete_range { , discrete_range })

discrete_range ::= *discrete* _subtype_indication | range

Syntax

- VHDL ist nicht case-sensitiv
- Kommentar begint mit - - und geht bis zum Zeilenende
- Anweisungen enden mit ´;´ und können über mehrere Zeilen gehen
- ´,´Trennzeichen zwischen Listenelementen
- Wertzuweisung an Signale <=

Entity - Minimalform

```
ENTITY __entity_name IS
  PORT(
  __in_name, __in_name              : IN
  BIT;
  __out_name, __out_name  : OUT  BIT);
END __entity_name;
```

Identifier, Namen, Bezeichner

Normale Identifier

- **Maximale Länge: 32 Zeichen**
- **Zeichensatz: a-z, A-Z, 0-9, und underscore (_)**
- **Das erste Zeichen muss ein Buchstabe sein**
- **Das letzte Zeichen darf nicht underscore sein**
- **Underscore darf nicht zweimal hintereinander stehen**
- **Name darf kein reserviertes Wort sein**
- **Namen sind nicht case-sensitiv**

Identifier, Namen, Bezeichner

Graphical / erweiterte Identifier in VHDL´93

- **Werden eingeschlossen in back slashes \name**
- **Namen sind case-sensitiv**
- **Graphikzeichen sind erlaubt**
- **Zwischenräume und mehrere aufeinanderfolgende Unterstriche sind erlaubt**
- **VHDL vordefinierte Worte sind erlaubt**

Ports

- **Stellen die Ein- und Ausgänge dar**
- **Sie werden beschrieben durch**
  - **Identifier / Namen**
  - **Modus / Signalübertragungsrichtung**
    - **IN** **Eingang**
    - **OUT** **Ausgang**
    - **INOUT** **Ein- Ausgang**
    - **Buffer** **für interne Rückkopplung**
- **Typ**

Vordefinierte Typen

BIT

- **Werte (´0´,´1´)**
- **Vordefinierte Funktionen für Bit**
  - **Funktion Ergebnis**
    - **and BIT**
    - **or BIT**
    - **nand BIT**
    - **nor BIT**

Vordefinierte Typen

BIT

- **Vordefinierte Funktionen für Bit**
  - **Funktion — Ergebnis**

Funktion	Ergebnis
xor	BIT
xnor	BIT
not	BIT

Vordefinierte Typen

BIT

- **Vordefinierte Funktionen für Bit**

Funktion	Ergebnis
=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Vordefinierte Typen

BIT_VECTOR

BIT_VECTOR (7 DOWNTO 0) oder (0 TO 7)

Werte:

“011001“

B“011001“

B“011_001“

O“31“

X“19“

Vordefinierte Typen

BIT_VECTOR

- **Vordefinierte Funktionen für BIT_VECTOR**

Funktion	Ergebnis
and	BIT_VECTOR
or	BIT_VECTOR
nand	BIT_VECTOR
nor	BIT_VECTOR

Vordefinierte Typen

BIT_VECTOR

- **Vordefinierte Funktionen für BIT_VECTOR**

Funktion	Ergebnis
• xor	BIT_VECTOR
• xnor	BIT_VECTOR
• not	BIT_VECTOR

Vordefinierte Typen

BIT_VECTOR

- **Vordefinierte Funktionen für Bit**

Funktion	Ergebnis
=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Vordefinierte Typen

INTEGER

- **zahl : INTEGER RANGE –255 to 255**
- **Vordefinierte Funktionen für INTEGER**
  - **Funktion — Ergebnis**

Funktion	Ergebnis
+	**INTEGER**
-	**INTEGER**

Vordefinierte Typen

INTEGER

- **Vordefinierte Funktionen für INTEGER**
  - **Funktion** — **Ergebnis**

Funktion	Ergebnis
=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Entwurfssichten

- **Strukturmodellierung**

- **Verhaltensmodellierung**

Beschreibungsformen

- **Strukturbeschreibung**
- **Datenflussbeschreibung**
- **Verhaltensbeschreibung**

Beschreibungsformen

Strukturbeschreibung

- **Beschreibung der Zusammenschaltung**
  - **von Gatter**
  - **Funktionsblöchen**
- **Beschreibung der Hierarchie**

Beschreibungsformen

Datenflussbeschreibung

- **Beschreibung arithmetischer Operationen**
- **Register – Tranfer**
- **Concurrent Assignments**

Beschreibungsformen

Verhaltensbeschreibung

- **Sequentielle Beschreibung**
- **Automatenbeschreibung**
- **Testbench**
- **Entwurfsspezifikation**

Beispiel für Beschreibungsformen

Verhaltensbeschreibung

- **Sequentielle Beschreibung**
- **Automatenbeschreibung**
- **Testbench**
- **Entwurfsspezifikation**

2.Beispiel

```
-- 4 zu 1 Multiplexer
-- Beschreibung mit Logikgleichungen
ENTITY mux_log IS
  PORT(
      da,db,dc,dd          : IN     BIT;
      aa,ab                : IN     BIT;
      daus                 : OUT    BIT);
END mux_log;
```

2.Beispiel

```
-- 4 zu 1 Multiplexer
-- Beschreibung mit Logikgleichungen
ARCHITECTURE amux_log OF mux_log IS
BEGIN
daus<=     (da and not aa and not ab) or
        (db and aa and not ab) or
        (dc and not aa and ab) or
        (dd and aa and ab) ;
END amux_log;
```

2.Beispiel

```
-- 4 zu 1 Multiplexer
-- c
ENTITY mux_when IS
  PORT(
     da,db,dc,dd          : IN  BIT;
     ad                   : IN  BIT_VECTOR(1
  downto 0);
     daus                 : OUT     BIT);
END mux_when;
```

2.Beispiel

```
-- 4 zu 1 Multiplexer
-- Beschreibung mit when - else
ARCHITECTURE amux_when OF mux_when IS
BEGIN
multiplexer:
daus <= da WHEN ad="00" ELSE
        db WHEN ad="01" ELSE
        dc WHEN ad="10" ELSE
        dd;
END amux_when;
```

2.Beispiel

```
-- 4 zu 1 Multiplexer
-- Beschreibung mit select - when
ENTITY mux_select IS
  PORT(
      da,db,dc,dd          : IN  BIT;
      ad                   : IN  BIT_VECTOR(1
  downto 0);
      daus                 : OUT     BIT);
END mux_select;
```

2.Beispiel

```
ARCHITECTURE amux_select OF mux_select IS
BEGIN
multiplexer:
WITH ad SELECT
   daus <= da WHEN "00",
           db WHEN "01",
           dc WHEN "10",
           dd WHEN "11";
END amux_select;
```

Die Library IEEE

Signaltypen

- **std_logic**
- **std_logig_vector**

Definiert in der Library IEEE

Einfügen der Library in ein

Die Library IEEE

Einfügen der Library in ein Design

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

std_logic Signalwerte

```
TYPE std_ulogic IS ( 'U',  -- Uninitialized
                     'X',  -- Forcing  Unknown
                     '0',  -- Forcing  0
                     '1',  -- Forcing  1
                     'Z',  -- High Impedance
                     'W',  -- Weak     Unknown
                     'L',  -- Weak     0
                     'H',  -- Weak     1
                     '-'   -- Don't care
                   );
```

std_logic Signalwerte

```
TYPE std_ulogic_vector IS ARRAY (
  NATURAL RANGE <> ) OF std_ulogic;

SUBTYPE std_logic IS resolved
  std_ulogic;

TYPE std_logic_vector IS ARRAY (
  NATURAL RANGE <>) OF std_logic;
```

std_logic Signalwerte

SUBTYPE X01 IS resolved std_ulogic RANGE 'X' TO '1'; -- ('X','0','1')

SUBTYPE X01Z IS resolved std_ulogic RANGE 'X' TO 'Z'; -- ('X','0','1','Z')

SUBTYPE UX01 IS resolved std_ulogic RANGE 'U' TO '1'; -- ('U','X','0','1')

SUBTYPE UX01Z IS resolved std_ulogic RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')

std_logic logische Funktionen

FUNCTION "and" (l : std_ulogic; r : std_ulogic) RETURN UX01;

FUNCTION "nand" (l : std_ulogic; r : std_ulogic) RETURN UX01;

FUNCTION "or"   (l : std_ulogic; r : std_ulogic) RETURN UX01;

FUNCTION "nor" (l : std_ulogic; r : std_ulogic) RETURN UX01;

std_logic logische Funktionen

```
FUNCTION "xor"  ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "xnor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;
FUNCTION "not"  ( l : std_ulogic ) RETURN UX01;
```

std_logic logische Funktionen

```
FUNCTION "and"  ( l, r : std_logic_vector  )
RETURN std_logic_vector;
FUNCTION "and"  ( l, r : std_ulogic_vector )
RETURN std_ulogic_vector;

FUNCTION "nand" ( l, r : std_logic_vector  )
RETURN std_logic_vector;
FUNCTION "nand" ( l, r : std_ulogic_vector )
RETURN std_ulogic_vector;
```

std_logic logische Funktionen

```
FUNCTION "or"   ( l, r : std_logic_vector  )
  RETURN std_logic_vector;
  FUNCTION "or"   ( l, r : std_ulogic_vector )
  RETURN std_ulogic_vector;

  FUNCTION "nor"  ( l, r : std_logic_vector  )
  RETURN std_logic_vector;
  FUNCTION "nor"  ( l, r : std_ulogic_vector )
  RETURN std_ulogic_vector;
```

std_logic logische Funktionen

```
FUNCTION "xor"  ( l, r : std_logic_vector  )
  RETURN std_logic_vector;
  FUNCTION "xor"  ( l, r : std_ulogic_vector )
  RETURN std_ulogic_vector;

  FUNCTION "xnor"  ( l, r : std_logic_vector  )
  RETURN std_logic_vector;
  FUNCTION "xnor"  ( l, r : std_ulogic_vector )
  RETURN std_ulogic_vector;
```

std_logic logische Funktionen

FUNCTION "not" (l : std_logic_vector) RETURN std_logic_vector;

FUNCTION "not" (l : std_ulogic_vector) RETURN std_ulogic_vector;

std_logic Umwandlungsfunktionen

FUNCTION To_bit (s : std_ulogic; xmap : BIT := '0') RETURN BIT;

FUNCTION To_bitvector (s : std_logic_vector ; xmap : BIT := '0') RETURN BIT_VECTOR;

FUNCTION To_bitvector (s : std_ulogic_vector; xmap : BIT := '0') RETURN BIT_VECTOR;

std_logic Umwandlungsfunktionen

FUNCTION To_StdULogic (b : BIT) RETURN std_ulogic;

FUNCTION To_StdLogicVector (b : BIT_VECTOR) RETURN std_logic_vector;

FUNCTION To_StdLogicVector (s : std_ulogic_vector) RETURN std_logic_vector;

std_logic Umwandlungsfunktionen

FUNCTION To_StdULogicVector (b : BIT_VECTOR) RETURN std_ulogic_vector;

FUNCTION To_StdULogicVector (s : std_logic_vector) RETURN std_ulogic_vector;

Multiplexer

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

ENTITY mux_4zu1 IS
   PORT
   (
   din         : IN    STD_LOGIC_VECTOR(3 DOWNTO 0);
   adr         : IN    STD_LOGIC_VECTOR(1 DOWNTO 0);
   daus        : OUT STD_LOGIC
   );
END mux_4zu1;
```

Multiplexer

```
ARCHITECTURE amux_4zu1 OF mux_4zu1 IS
BEGIN
multiplexer:
WITH adr SELECT
   daus     <=    din(0)     WHEN     "00",
                  din(1)     WHEN     "01",
                  din(2)     WHEN     "10",
                  din(3)     WHEN     "11";

END amux_4zu1;
```

Multiplexer mit Strukturbeschreibung

```
library altera;
use altera.maxplus2.all;
library ieee;
use ieee.std_logic_1164.all;
```

Multiplexer mit Strukturbeschreibung

```
entity mux_struktur is
  port(
     ad   :in std_logic_vector(1 downto 0);
     di   :in std_logic_vector(3 downto 0);
     da   :out std_logic
     );
end mux_struktur;
```

Multiplexer mit Strukturbeschreibung

architecture structur of mux_struktur is

signal a0n,a1n : std_logic;

signal zu : std_logic_vector(3 downto 0);

Multiplexer mit Strukturbeschreibung

```
begin
   u0: a_7404 port map (ad(0),a0n);        -- negator
   u1: a_7404 port map (ad(1),a1n);
   u2: a_7410 port map (a0n,a1n,di(0),zu(0));
   u3: a_7410 port map (ad(0),a1n,di(1),zu(1));
   u4: a_7410 port map (a0n,ad(1),di(2),zu(2));
   u5: a_7410 port map (ad(0),ad(1),di(3),zu(3));
   u6: a_7420 port map (zu(0),zu(1),zu(2),zu(3),da);
end structur;
```

Multiplexer mit PROCESS

```
__process_label:
PROCESS (__signal_name, __signal_name...)
  VARIABLE __variable_name   :      STD_LOGIC;
BEGIN
  -- Signal Assignment Statement (optional)
  -- Variable Assignment Statement (optional)
  -- Procedure Call Statement (optional)
  -- If Statement (optional)
  -- Case Statement (optional)
  -- Loop Statement (optional)
END PROCESS __process_label;
```

Multiplexer mit PROCESS

```
CASE __expression IS
  WHEN __constant_value =>
      __statement;
      __statement;
  WHEN __constant_value =>
      __statement;
      __statement;
  WHEN OTHERS =>
      __statement;
      __statement;
END CASE;
```

Multiplexer mit PROCESS

```
IF __expression THEN
  __statement;
  __statement;
ELSIF __expression THEN
  __statement;
  __statement;
ELSE
  __statement;
  __statement;
END IF;
```

Multiplexer mit PROCESS und CASE

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

entity mux_proc is
   port(
        ad          :in std_logic_vector(1 downto 0);
        di          :in std_logic_vector(3 downto 0);
        da          :out std_logic
        );
end mux_proc;
```

```
architecture amux_proc of mux_proc is
begin
multiplexer: PROCESS (ad,di)
BEGIN
  CASE ad IS
      WHEN "00" =>   da<=di(0);
      WHEN "01" =>   da<=di(1);
      WHEN "10" =>   da<=di(2);
      WHEN "11" =>   da<=di(3);
      WHEN OTHERS =>     da<='0';
  END CASE;
END PROCESS multiplexer;
end amux_proc;
```

Multiplexer mit PROCESS und IF

```
library ieee;
use ieee.std_logic_1164.all;
entity mux_proc_if is
   port(
        ad          :in std_logic_vector(1 downto 0);
        di          :in std_logic_vector(3 downto 0);
        da          :out std_logic
        );
end mux_proc_if;
```

```
architecture amux_proc_if of mux_proc_if is
begin
multiplexer: PROCESS (ad,di)
BEGIN
   IF ad="00" THEN              da<=di(0);
   ELSIF  ad="01" THEN          da<=di(1);
   ELSIF  ad="10" THEN          da<=di(2);
   ELSIF  ad="11" THEN          da<=di(3);
   ELSE                         da<='0';
   END IF;
END PROCESS multiplexer;
end amux_proc_if;
```

Zusammenfassung Sprachelemente in Kapitel 1

Entity

ARCHITECTURE

VHDL ist nicht case-sensitiv

Kommentar begint mit - - und geht bis zum Zeilenende

Anweisungen enden mit ´;´ und können über mehrere Zeilen gehen

´,´Trennzeichen zwischen Listenelementen

Wertzuweisung an Signale <=

Zusammenfassung Sprachelemente in Kapitel 1

Normale Identifier

- Maximale Länge: 32 Zeichen
- Zeichensatz: a-z, A-Z, 0-9, und underscore (_)
- Das erste Zeichen muss ein Buchstabe sein
- Das letzte Zeichen darf nicht underscore sein
- Underscore darf nicht zweimal hintereinander stehen
- Name darf kein reserviertes Wort sein
- Namen sind nicht case-sensitiv

Zusammenfassung
Sprachelemente in Kapitel 1

Ports

Vordefinierte Typen

BIT

Werte (´0´,´1´)

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte logische Funktionen für Bit

Funktion	Ergebnis
and	BIT
or	BIT
nand	BIT
nor	BIT
xor	BIT
xnor	BIT
not	BIT

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte relationale Funktionen für Bit

Funktion	Ergebnis
=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte Typen

BIT_VECTOR

BIT_VECTOR (7 DOWNTO 0) oder (0 TO 7)

Darstellung von Werten:

“011001“

B“011001“

B“011_001“

O“31“

X“19“

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte logische Funktionen für BIT_VECTOR

Funktion	Ergebnis
and	BIT_VECTOR
or	BIT_VECTOR
nand	BIT_VECTOR
nor	BIT_VECTOR
xor	BIT_VECTOR
xnor	BIT_VECTOR
not	BIT_VECTOR

Zusammenfassung
Sprachelemente in Kapitel 1

Vordefinierte relationale Funktionen für BIT_VECTOR

Funktion	Ergebnis
=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte Typen

INTEGER

zahl : INTEGER RANGE –255 to 255

Vordefinierte arithmetische Funktionen für INTEGER

Funktion	Ergebnis
+	INTEGER
-	INTEGER

Zusammenfassung
Sprachelemente in Kapitel 1

Vordefinierte relationale Funktionen für INTEGER

Funktion	Ergebnis
=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Zusammenfassung
Sprachelemente in Kapitel 1

Funktionsbeschreibung

Beschreibung mit Logikgleichungen

Beschreibung mit when - else

Beschreibung mit select - when

Strukturbeschreibung

Zusammenfassung Sprachelemente in Kapitel 1

Funktionsbeschreibung kombinatorischer Logik

- Beschreibung mit Process
  - case
  - if elseif else

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte Funktionen für Bit

Funktion	Ergebnis
and	BIT
or	BIT
nand	BIT
nor	BIT
xor	BIT
xnor	BIT
not	BIT

Zusammenfassung
Sprachelemente in Kapitel 1

Vordefinierte Funktionen für Bit

=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Zusammenfassung Sprachelemente in Kapitel 1

BIT_VECTOR

BIT_VECTOR (7 DOWNTO 0) oder (0 TO 7)

Darstellung von Werten:

“011001“

B“011001“

B“011_001“

O“31“

X“19“

Zusammenfassung Sprachelemente in Kapitel 1

Vordefinierte Funktionen für BIT_VECTOR

Funktion	Ergebnis
and	BIT_VECTOR
or	BIT_VECTOR
nand	BIT_VECTOR
nor	BIT_VECTOR
xor	BIT_VECTOR
xnor	BIT_VECTOR
not	BIT_VECTOR

Zusammenfassung
Sprachelemente in Kapitel 1

Vordefinierte Funktionen für BIT_VECTOR

=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Zusammenfassung Sprachelemente in Kapitel 1

INTEGER

zahl : INTEGER RANGE –255 to 255

Vordefinierte Funktionen für INTEGER

Funktion	Ergebnis
+	INTEGER
-	INTEGER

Zusammenfassung Sprachelemente in Kapitel 1

=	BOOLEAN
/=	BOOLEAN
<	BOOLEAN
<=	BOOLEAN
>	BOOLEAN
>=	BOOLEAN

Zusammenfassung Sprachelemente in Kapitel 1

Library in einem Design

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;

library altera;
use altera.maxplus2.all;
```