
Digitaltechnik

Kapitel 3, Einführung in VHDL

Prof. Dr.-Ing. M. Winzker

Nutzung nur für Studierende der Hochschule Bonn-Rhein-Sieg gestattet.
(Stand: 20.03.2019)

3.1 Einleitung

VHDL ist eine **Hardwarebeschreibungssprache** (engl. „Hardware Description Language“, HDL)

- Das ‚V‘ in VHDL steht für die Initiative „VHSIC“ (Very High Speed Integrated Circuits), unter der VHDL ursprünglich entstand

VHDL dient für zwei Aufgaben bei der Schaltungsentwicklung

- **Entwurf:**

- Die Funktion einer Schaltung wird mit VHDL beschrieben
- Mit Hilfe von Computerprogrammen wird aus VHDL eine Schaltung erzeugt
Beispiel: Ein Addierer wird beschrieben als $C \leq A+B$;

- **Verifikation:**

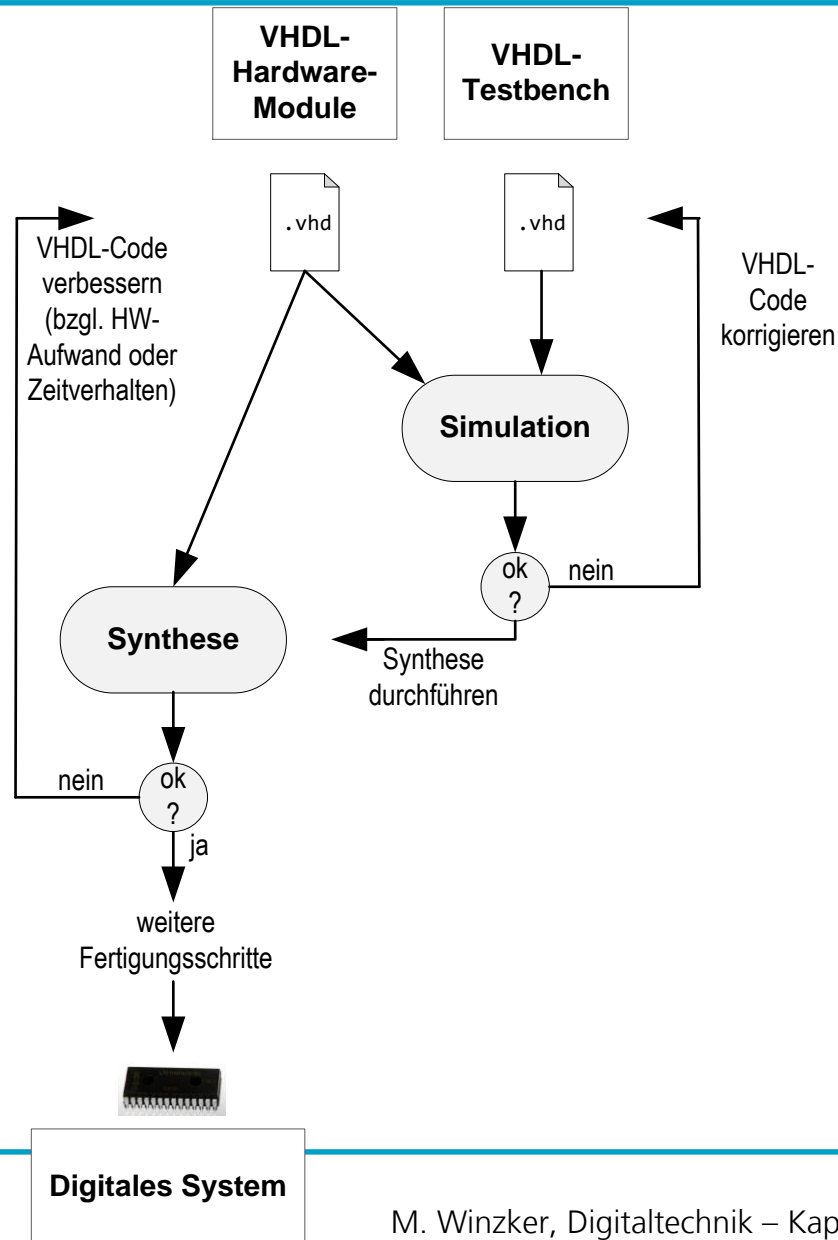
- Das korrekte Verhalten einer Schaltung wird überprüft
Beispiel: Für eine Addierschaltung werden verschiedene Werte für A und B an die Schaltung angelegt und das Ergebnis mit dem erwarteten Wert verglichen

VHDL zum Schaltungsentwurf

- In diesem Abschnitt soll zunächst nur der **Schaltungsentwurf** behandelt werden
 - **Schaltungsverifikation** wird später behandelt
- Schaltungsentwurf mit VHDL nutzt nur einen Teil des Sprachumfangs
- Die VHDL-Beschreibung einer Schaltung wird per Computerprogramm in eine Schaltung umgesetzt
 - Die Umsetzung bezeichnet man als **Synthese**, oder **Schaltungssynthese**
 - Das entsprechende Computerprogramm ist das **Syntheseprogramm**, **Synthese-Tool**
 - Auch als **CAD-Tool** („Computer-Aided-Design“) oder **EDA-Tool** („Electronic Design Automation“)
 - VHDL-Code zur Schaltungssynthese muss **synthesefähig** sein
- Zur Verifikation wird eine **Testbench** (auf deutsch etwa „Prüfstand“) eingesetzt
 - Die Testbench erzeugt eine Folge von Eingangswerten für die Schaltung und überprüft die Ausgangswerte
 - Dies bezeichnet man als **Simulation**. Die **EDA-Tools** sind **Simulatoren**.
 - Hierfür können **nicht-synthesefähige** VHDL-Konstrukte verwendet werden



VHDL zum Schaltungsentwurf (II)



„Golden Rules of VHDL“

VHDL is a hardware-design language

- When you are working with VHDL, you are not programming, you are “designing hardware”. Your VHDL code should reflect this fact.
- If your VHDL code appears too similar to code of a higher-level computer language, it is probably bad VHDL code.

Have a general concept of what your hardware should look like

- Although VHDL is vastly powerful, if you do not understand basic digital constructs, you will probably be unable to generate efficient digital circuits.
- If you are not able to roughly envision the digital circuit you are trying to model in terms of basic digital circuits, you will probably misuse VHDL.
- VHDL is cool, but it is not as magical as it initially appears to be.

Aus: B. Mealy, F. Tappero, “Free Range VHDL,” <http://freerangefactory.org>

Weitere Literatur:

- J. Reichardt, B. Schwarz, „VHDL-Synthese“, Oldenbourg-Verlag.

3.2 Grundstruktur eines VHDL-Moduls

- Ein Modul bezeichnet eine Schaltungseinheit
- Ein Modul kann andere Module als Teilschaltungen enthalten
- Jedes Modul wird in VHDL in zwei Teilen beschrieben:

Entity:

- Die Entity beschreibt die **äußere Schnittstelle** eines Moduls, also die Eingangssignale und Ausgangssignale

Architecture:

- Die Architecture beschreibt die Funktion eines Moduls
- Für ein Modul können mehrere Architecture angelegt werden, z.B.:
 - Eine nicht-synthesefähige Architecture zur Simulation
 - Eine synthesefähige Architecture zur Synthese
 - Eine Architecture aus Logik-Gattern, nach der Synthese

- Außerdem gibt es noch:

Package:

- Ein Package enthält Definitionen und Funktionen (ähnlich .h-Files bei C)



Beispiel: Einfaches Modul in VHDL

```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity or_and is  
  port (  
    a : in    std_logic;  
    b : in    std_logic;  
    c : in    std_logic;  
    y : out   std_logic);  
end or_and;  
  
architecture behave of or_and is  
begin  
  
  y <= (a or b) and c;  
  
end;
```

Aufruf eines Package mit
Definition von
Datentypen

Dies ist die Benutzung,
nicht die Definition
eines Package

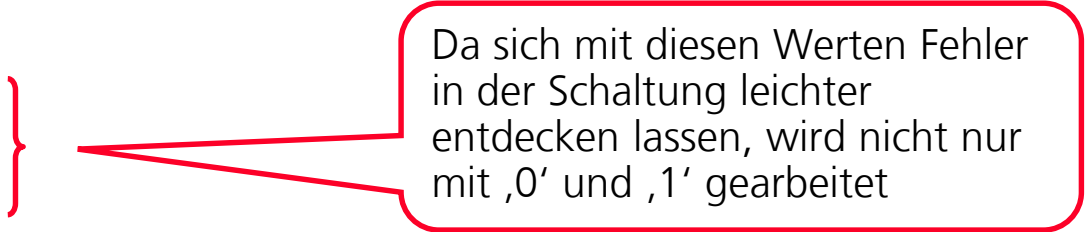
Entity Beschreibung des
Moduls „OR_AND“ mit
drei Eingängen A, B, C
und einem Ausgang Y

Architecture Beschreibung des
Moduls „OR_AND“
Der Ausgang Y ergibt sich als
boolesche Gleichung aus den drei
Eingängen

Schlüsselwörter sind
fett gedruckt

3.3 Datentypen

In VHDL sind verschiedene Datentypen in der Sprache selbst sowie in Packages vordefiniert. Zusätzliche Typen können definiert werden.

- Der allgemeinste Datentyp ist **std_logic**
 - Das Package `ieee.std_logic_1164.all` muss eingebunden sein
 - `std_logic` kann neun verschiedenen Werte einnehmen, darunter:
 - '0' = logischer Wert 0
 - '1' = logischer Wert 1
 - 'X' = unbekannt
 - 'U' = noch nicht initialisiert
 - 'Z' = hochohmig
- 
- Da sich mit diesen Werten Fehler in der Schaltung leichter entdecken lassen, wird nicht nur mit ,0' und ,1' gearbeitet
- Ein 1-dimensionales Feld des Typs `std_logic` heißt **std_logic_vector** und wird z.B. für Datenbusse verwendet. Die Angabe der Wortbreite ist stets erforderlich, etwa **std_logic_vector (7 downto 0)**.

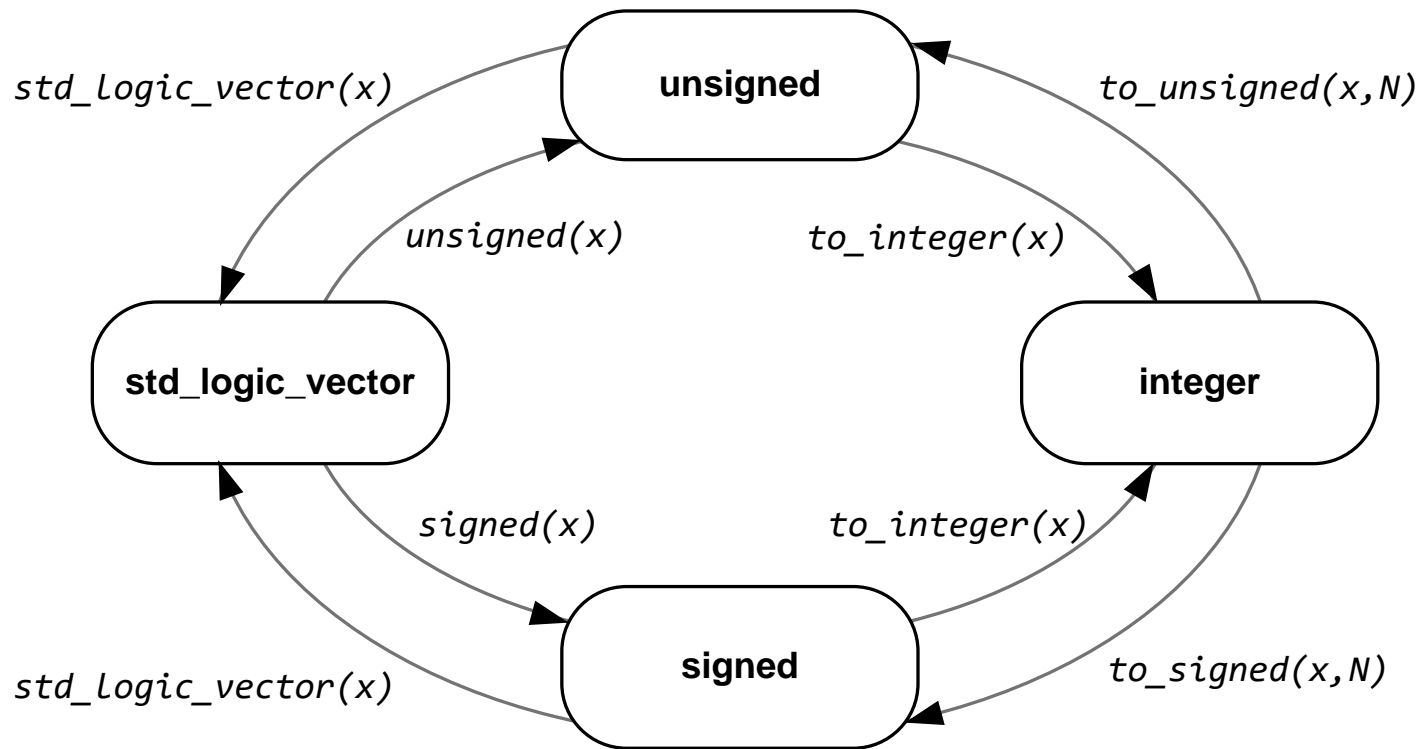
Datentypen (II)

Für Arithmetik wird der **std_logic_vector** in die beiden Datentypen **signed** und **unsigned** umgewandelt

- Das Package `ieee.numeric_std.all` muss eingebunden sein
- Mit **signed** und **unsigned** sind arithmetische Operationen möglich
- Eine abstraktere Darstellung von Zahlenwerten, ohne Betrachtung der einzelnen Bits, ist mit dem Datentyp **integer** möglich
 - Wertebereich -2^{31} bis $+2^{31}-1$
- Weitere Datentypen sind unter anderem:
 - `bit` und `bit_vector`: Kann nur die Werte '0' und '1' annehmen
 - `float`: Floating-Point-Zahlen (nur für die Simulation)

Umwandlung zwischen Datentypen

Für die Umwandlung zwischen den Datentypen stehen verschiedene Funktionen zur Verfügung



x: umzuwandelnder Wert

N: Wortbreite

Umwandlung zwischen Datentypen (II)

Je nach Konvertierungsfunktion muss die Wortbreite angegeben werden

Exemplarische Typumwandlungen

i	<= to_integer (s8);	-- signed -> integer
u8	<= to_unsigned (i,8);	-- integer -> unsigned
s8	<= to_signed (-123,8);	-- Ganzzahlige Konstanten: -- Datentyp Integer
slv8	<= std_logic_vector (u8);	-- unsigned -> std_logic_vector
i	<= to_integer (unsigned (slv8));	-- std_logic_vector -> integer
slv8	<= std_logic_vector (to_signed (i,8));	-- integer -> std_logic_vector

Signale und Ports

- **Signale** in VHDL entsprechen Leitungen einer Schaltung
- Signale müssen mit ihrem Datentyp definiert werden
 - Bei Vektoren muss auch die Wortbreite und die Zählrichtung angegeben werden
 - Zählrichtung heißt, **(7 downto 0)** und **(0 to 7)** sind nicht identisch
 - VHDL prüft die Übereinstimmung von Datentypen, Wortbreite und Zählrichtung sehr penibel
 - Eine Signalzuweisung von (7 downto 0) an (0 to 7) ist ein Fehler
- Ein **Port** ist ein Eingangs- oder Ausgangssignal
 - Eingabesignale können nur lesend verwendet werden
 - Ausgabesignale können nicht gelesen, sondern nur beschrieben werden
 - Ein Port kann auch bidirektional sein. Bidirektionale Ports werden üblicherweise nur an den Grenzen eines Bausteins und nicht innerhalb eines Bausteins verwendet.
 - Der Datenbus zu einem RAM-Speicher ist bidirektional. Je nach Operation (Lesen, Schreiben) gehen Daten zum oder vom RAM.

Definition von Signalen und Ports

- Ports werden in der Entity definiert und in der Architecture verwendet

```
entity sub_module is  
  port (  
    a    : in   std_logic_vector(7 downto 0);  
    b    : in   std_logic_vector(7 downto 0);  
    sel  : in   std_logic;  
    x    : out  std_logic;  
    y    : out  std_logic_vector(7 downto 0) );  
end sub_module;
```

- Signale werden in der Architecture definiert

```
architecture behave of sub_module is  
  signal sum_a : std_logic_vector(7 downto 0);  
  signal sum_b : std_logic_vector(8 downto 0);  
  signal abc   : std_logic_vector(7 downto 0);  
  
begin  
  ...  
  
end;
```

Hier wird das Verhalten des Moduls beschrieben

Weitere Sprachelemente

Kommentare

- Kommentare beginnen in VHDL mit zwei Minus-Zeichen und gehen bis zum Ende der Zeile
 - `a <= b or c; -- Dies ist ein Kommentar`

Bezeichner (Identifizier)

- Bezeichner beginnen mit einem Buchstaben und dürfen Buchstaben, Ziffern und den Unterstrich `_` (Underscore) enthalten
- Reservierte Worte (z.B. „if“, „and“) dürfen nicht als Bezeichner verwendet werden
 - `value_10`, `sum`
 - Ein kontext-sensitiver Editor zeigt reservierte Worte an

Konstanten

- Konstanten für `std_logic` müssen durch einfache Anführungsstriche, für `std_logic_vector` durch doppelte Anführungsstriche begrenzt sein
- Ansonsten wäre eine Verwechslung mit Zahlen möglich
 - `a <= '1';` *-- a ist std_logic*
 - `b <= "0101";` *-- b ist std_logic_vector(3 downto 0);*

Wertzuweisung an Vektoren

- Ein `std_logic_vector` ist ein Feld von `std_logic`
- Eine Wertzuweisung an den Vektor ist auf mehrere Weisen möglich:
 - Annahme: a und b sind `std_logic_vector(3 downto 0)`, c bis f sind `std_logic`
- Zuweisung eines passenden anderen Vectors oder einer Konstante

```
a <= b;
```

```
a <= "0101";
```

- Direkter Zugriff auf die Elemente des Feldes:

```
a(1 downto 0) <= b(3 downto 2);
```

```
a(0) <= c;
```

```
a(1) <= d;
```

```
a(2) <= e;
```

```
a(3) <= f;
```

- Concatenation mit dem `&`-Operator

```
a <= f & e & d & c;
```

```
a <= "10" & b( 1 downto 0 );
```

- Achtung: VHDL überprüft die Zählrichtung. Folgendes ist nicht möglich:

```
a(3 downto 0) <= b(0 to 3); -- ERROR
```



Wertzuweisung an Vektoren (II)

- Den einzelnen Stellen eines Vektors können per direkter Angabe der Stelle Werte zugeordnet werden
- Für nicht genannte Stellen kann eine Zuweisung mit „others“ erfolgen

```
a <= (0 => c, 1 => d, others => e);
```

- Dies ermöglicht auch eine elegante Wertzuweisung für Konstanten
 - Insbesondere für größere Wortbreiten ist die „others“-Schreibweise vorteilhaft

```
zero <= (others => '0');
```

```
one <= (0 => '1', others => '0');
```


Header

- Jede VHDL-Datei sollte an ihrem Anfang einen “**Header**” haben, in dem, als Kommentar, folgende Informationen enthalten sind
 - Name des Moduls (oder Packages)
 - Aufgabe des Moduls
 - Je nach Komplexität auch eine Beschreibung von Randbedingungen
 - Autor, Firma, Datum
- Des weiteren sollten im Header Verifikationen und Änderungen protokolliert werden
 - Art der Änderung oder Verifikation
 - Autor und Datum
- Änderungen können auch durch Software zur Versionskontrolle erfasst werden
- Bedenken Sie, dass an einem Projekt meist mehrere Personen beteiligt sind
 - Auch zu einem kleinen Projekt können plötzlich weitere Personen hinzugezogen werden
 - Manchmal werden Projekte nach monatelangem „Winterschlaf“ von einer anderen Person weitergeführt

Header (II)

Beispiel für einen Header:

```
-- x_delay.vhd
--
-- output Q gives a single pulse of '1' if input A goes from '1'
--      to '0' and stays '0' for 40 clock cycles
--
-- (c) M. Winzker, H-BRS, 29.07.2013
--
--      History:
--      * simulated without errors, M. Winzker, 05.08.2013
--      * optimized for synthesis, F. Gonzales, 12.08.2013
```

Bei von mir begutachteten Abschlussarbeiten geht das Vorhandensein und der Umfang von Headern und Kommentaren in die Beurteilung ein

In der Industrie geht die Dokumentation oft indirekt in Ihre Beurteilung und damit in Ihre Bezahlung ein

Englisch oder Deutsch?

- Die Schlüsselwörter von VHDL sind englisch (z.B. if, then, else)
- Namen für Eingänge, Ausgänge, interne Signale können frei gewählt werden
- In der Praxis sollten englische Worte als Bezeichner gewählt werden. Dafür gibt es mehrere Gründe:
 - Es findet kein Bruch im Lesen zwischen Schlüsselwörtern und Signalnamen statt
 - Englische Namen sind oft etwas kürzer
 - Dateien können an Kollegen/Kolleginnen in anderen Ländern weitergegeben werden
 - Bei Problemen mit EDA-Tools können VHDL-Dateien an den Hersteller-Support in den USA, England, Italien, Rumänien (oder wo auch immer) weitergegeben werden
- VHDL ist nicht case-sensitiv
- Es empfiehlt sich, vorwiegend Kleinbuchstaben zu verwenden und die Syntax durch einen Editor mit Syntax-Highlighting hervorheben zu lassen

3.4 Nebenläufige Anweisungen

- In einer **Schaltung** arbeiten normalerweise alle Elemente **gleichzeitig**, also **parallel**
- Dies ist **anders** als in einem Computerprogramm!
- Parallele Elemente werden in VHDL durch **nebenläufige Anweisungen** programmiert

Nebenläufige Signalzuweisung:

- Mit der Signalzuweisung „`<=`“ wird die Datenübergabe an ein Signal beschrieben
- Einem Signal kann ein anderes **Signal**, eine **Konstante** oder ein **logischer Ausdruck** zugewiesen werden
 - `a <= b;`
 - `c <= '1';`
 - `d <= e and f;`
 - `g <= (h or k) xor (not 1);`
- Jedes Mal, wenn sich ein Wert rechts der Signalzuweisung ändert, erfolgt automatisch eine neue Signalzuweisung

VHDL beschreibt Parallelverarbeitung

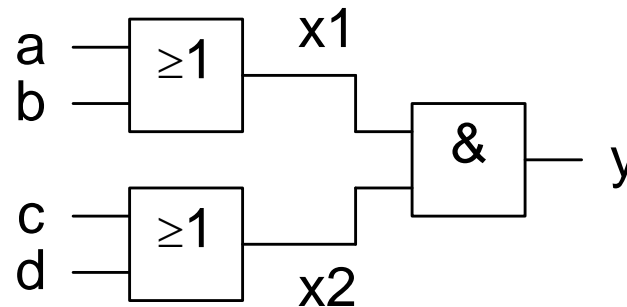
Noch einmal:

- Mehrere nebenläufige Anweisungen werden parallel ausgeführt
- Die **Reihenfolge** der Beschreibungen ist **beliebig**!
- Folgende Beschreibungen führen zu dem **gleichen Verhalten**:

```
x1 <= a or b;  
x2 <= c or d;  
y  <= x1 and x2;
```

```
y  <= x1 and x2;  
x1 <= a or b;  
x2 <= c or d;
```

- Beide Beschreibung entsprechen folgender Schaltung:



3.5 Prozesse

- **Prozesse** sind ein wesentliches Element, um in VHDL Funktionalität zu beschreiben
- Innerhalb von Prozesse erfolgen Abläufe **sequenziell**, während unterschiedliche Prozesse **parallel** zueinander ablaufen
- Prozesse bilden also praktisch eine Brücke zwischen sequenzieller und paralleler Verhaltensbeschreibung
- Man unterscheidet zwei Arten von Prozessen:
 - Prozesse für kombinatorische Funktionalität
 - Prozesse für sequenzielle Funktionalität (Automaten)

Prozesse für kombinatorische Funktionalität

- Die schon kennen gelernten nebenläufigen Anweisungen entsprechen Prozessen
- Statt einer nebenläufigen Anweisung kann komplexere Funktionalität oft einfacher in einem Prozess dargestellt werden

Prozesse für sequenzielle Funktionalität

- werden später betrachtet

Prozesse für kombinatorische Funktionalität

- Ein Prozess ist eine Folge von VHDL-Befehlen, die nacheinander ausgeführt werden
- Die Ausführung beginnt, sobald sich ein (oder mehrere) Signale ändern
- Die Definition der relevanten Signale erfolgt in einer Sensitivity List

```
process (signal_a, signal_b, signal_c)
begin
    ...
    ...
    ...
    ...
end process;
```

Die Funktionen innerhalb eines Prozesses werden sequenziell abgearbeitet, jedes Mal, wenn sich ein Signal in der Sensitivity List ändert

Sequenzielle Anweisungen

Sequenzielle Signalzuweisungen

- Innerhalb eines Prozesses können sequenzielle Signalzuweisungen ähnlich den nebenläufigen Signalzuweisungen erfolgen
- Während die **Syntax gleich** ist, gibt es im Verhalten einen **fundamentalen Unterschied**
- Sequenzielle Signalzuweisungen werden **nacheinander** ausgeführt und können daher vorherige Anweisungen aufheben
- Folgende Anweisungen sind **erlaubt** und führen zu **unterschiedlichem Verhalten**

<pre>a <= b or c; a <= d or e;</pre>
--

<pre>a <= d or e; a <= b or c;</pre>
--

- Die Signalzuweisungen werden bis zum Ende der Prozessbeschreibung **vorgemerkt** und **erst dann ausgeführt**
- Sinnvoll ist eine mehrfache Signalzuweisung, wenn zunächst ein Standardwert gewählt wird und in einer if-then-else-Anweisung geändert werden soll
- Eine doppelte **nebenläufige** Signalzuweisung zum Signal a wäre ein **Fehler**
 - Ausnahme: Busse mit hochohmigen Treibern (Tri-State)

if-then-else-Anweisung

- In einem Prozess kann der Programmablauf durch if-then-else-Anweisungen gesteuert werden
- Eine Verschachtelung mehrerer Anweisungen ist möglich

Definition:

```
if condition then  
    statements;  
{elsif condition then  
    statements;}  
{elsif condition then  
    statements;}  
else  
    statements;  
end if;
```

Beispiele:

```
if (a='0' and b='0') then  
    y1 <= '1';  
    y2 <= '1';  
elsif (c='1') then  
    y2 <= d and e;  
else  
    y2 <= f;  
end if;
```

```
if (a='0') then  
    if (b=c) then  
        y <= '1';  
    end if;  
else  
    y <= d;  
end if;
```

Operatoren

Die folgenden **Operatoren** sind definiert:

- Vergleiche: =, /=, <, <=, >, >=
- Logische Operatoren: not, and, nand, or, nor, xor, xnor
- std_logic_vector können als Dualzahlen addiert, subtrahiert und verglichen werden. Hierzu müssen Packages eingebunden werden.
- Das Ergebnis einer Addition, Subtraktion hat die gleiche Wortbreite wie der breitere Operand

Achtung:

- Wenn zwei 8 bit Zahlen addiert werden, ist das Ergebnis wieder eine 8 bit Zahl
- Zum Vermeiden von Überlauf sollten die Eingangswerte ggf. auf 9 bit erweitert werden

case-when-Anweisung

case-when-Anweisung

- Eine case-when-Anweisung dient der Auswahl verschiedener Optionen für ein Signal
- Mehrere Optionen können durch einen ODER-Operator „|“ gemeinsam behandelt werden (siehe Beispiel)

Definition:

```
case expression is  
  when choice_1 =>  
    statements;  
  when choice_2 =>  
    statements;  
  when others =>  
    statements;  
end case;
```

Beispiel:

```
case command is  
  when "00" =>  
    y <= a;  
  when "01" | "10" =>  
    y <= b;  
  when others =>  
    y <= c;  
end case;
```

„others“-Fall bei case-when-Anweisung

- In der Digitaltechnik werden „eigentlich“ **nur die Werte** ,0' und ,1' benutzt
- Für die Beschreibung realer Schaltungen werden jedoch **weitere Werte**, wie ,X' oder ,U' benutzt, um unbekannte oder noch nicht initialisierte Signale anzugeben
- Der Datentyp std_logic kann 9 verschiedene Werte annehmen
- Als Konsequenz kann z.B. ein 2 bit Signal **mehr als 4 Werte** annehmen (nämlich 81)
 - Neben „00“, „01“, „10“, „11“ sind dies etwa „X0“, „1U“, usw.
- Eine Case-Anweisung muss (theoretisch) diese 81 Kombinationen behandeln
- Der „others“-Fall deckt alle noch nicht behandelten Möglichkeiten ab

Frage: Wie definiert man in VHDL das Verhalten bei undefinierten Eingängen?

- In der Praxis gibt es zwei Möglichkeiten:
 - Die Ausgänge werden ebenfalls als undefiniert gesetzt (für Simulation)
 - Ein Fall der Case-Anweisung deckt alle undefinierten Kombinationen ab
 - o Für die Synthese, da undefinierte Werte **nicht synthesefähig** sind

Beispiel: Multiplexer

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

entity multiplex is
    port ( sel      : in  std_logic_vector(1 downto 0);
          in_a      : in  std_logic_vector(7 downto 0);
          in_b      : in  std_logic_vector(7 downto 0);
          in_c      : in  std_logic_vector(7 downto 0);
          in_d      : in  std_logic_vector(7 downto 0);
          output     : out std_logic_vector(7 downto 0) );
end multiplex;

architecture behave of multiplex is
begin
    process(sel, in_a, in_b, in_c, in_d)
    begin
        case sel is
            when "00"    => output <= in_a;
            when "01"    => output <= in_b;
            when "10"    => output <= in_c;
            when others => output <= in_d;
        end case;
    end process;
end behave;
```

Package zur Definition der Datentypen
std_logic und std_logic_vector

Definition der Ein- und
Ausgabeports

Anhand des 2 bit
Signals sel wird ein 8 bit
Bus ausgewählt



3.6 Arithmetik

- Rechenoperationen können mit signed und unsigned durchgeführt werden
 - Konvertierung, siehe Folie 10
 - Package erforderlich: use ieee.numeric_std.all;
- Wortbreite muss beachtet werden, Addition zweier 8 bit Werte gibt wieder 8 bit
 - Kann sinnvoll sein, bei Zähler mit „wrap-around“, also Zählreihenfolge
... 253, 254, 255, 0, 1, 2, ...
- Normalerweise ist vorherige Erweiterung der Operanden sinnvoll
 - sum <= '0' & op1 + '0' & op2; -- für Datentyp **unsigned**
 - sum <= op1(7) & op1 + op2(7) & op2; -- für Datentyp **signed**

Konvertierung der Datentypen (siehe Folie 10):

- std_logic_vector nach Integer:
value_int <= to_integer(signed(value_vec));
- Integer nach std_logic_vector:
value_vec <= std_logic_vector(to_unsigned(value_int,n));
mit <n> als Wortbreite des std_logic_vector

Verfügbare synthesefähige Operationen

- Addition, Subtraktion: „+“, „-“
- Multiplikation: „*“
- Division nur eingeschränkt: „/“ (normalerweise nur Division durch 2er-Potenz)
- Vergleich:
 - gleich, ungleich: „=“, „/=“
 - kleiner, kleiner-gleich: „<“, „<=“
 - größer, größer-gleich: „>“, „>=“

Operationen mit Signalen und Konstanten

- | | |
|--|---|
| <code>a <= b + c;</code> | -- a, b und c haben gleiche Wortbreite, z.B. 8 bit |
| <code>a <= b + "00010001";</code> | -- auch Konstante möglich |
| <code>a <= b + 17;</code> | -- Konstante darf <u>bei Rechenoperation</u> Integer sein |
| <code>a <= 17; ☹</code> | -- Kein Integer bei Zuweisung |
| <code>a <= to_unsigned(17,8);</code> | -- Zuweisung mit Typumwandlung: Wert 17, 8 bit |
| <code>a <= "00010001";</code> | -- das geht immer |
| <code>a <= (others => '0');</code> | -- für Wert Null |

Arithmetik mit „integer“

Rechenoperationen können mit signed/unsigned oder integer durchgeführt werden:

- Vorteil signed: Ein undefiniertes Signal wird als undefiniert dargestellt
- Vorteil signed: Zugriff auf einzelne Stellen des Wortes möglich
- Nachteil signed: Unübersichtlich bei Zugriff auf einzelne Stellen des Wortes
- Nachteil signed: Umständlich, z.B. bei Angabe von konstanten Werten

Für einfache Schaltungen ist signed und unsigned sinnvoll, für komplexere Schaltungen ist die Rechnung in Integer übersichtlicher

Konvertierung: Siehe Folie 10

- **Problem:** Die erforderliche Wortbreite ist nicht definiert
 - Standardmäßig wird meist 32 Bit verwendet
 - ➔ Unnötiger Aufwand und Kosten
- **Lösung:** Bei der Definition eines Integer kann der Wertebereich angegeben werden
 - VHDL-Schlüsselwort „range“, z.B.:
`signal value : integer range 0 to 63;`
 - ➔ Es wird nur die erforderliche Wortbreite generiert

Beispiel: Addition zweier Zahlen

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

entity arith is
    port ( a      : in  std_logic_vector(7 downto 0);
          b      : in  std_logic_vector(7 downto 0);
          y      : out std_logic_vector(8 downto 0));
end arith;

architecture behave of arith is
    signal a_i : integer range 0 to 255;
    signal b_i : integer range 0 to 255;
    signal y_i : integer range 0 to 511;

    begin
        a_i <= to_integer(unsigned(a));
        b_i <= to_integer(unsigned(b));
        y_i <= a_i + b_i;
        y <= std_logic_vector(to_unsigned(y_i, 9));
    end behave;
```

Packages zur Definition der Datentypen
und der Konvertierungsfunktionen

Definition der Integersignale mit
beschränkter Wortbreite

Konvertierung nach Integer

Arithmetische Operation

Rückkonvertierung nach
std_logic_vector mit Angabe
der Wortbreite: 9 bit

Bonn-Rhein-Sieg

Packages zur Definition der Datentypen und der arithmetischen Funktionen

Internes Signal für
Zwischenwert

Prozess wird durch Wechsel
von sum_9_i aufgerufen

Prof. Dr. Witzke, Digitaltechnik – Kap. 5, Einführung in VHDL, Folie 34

3.7 Beschreibung von Hierarchie

- Jedes VHDL-Modul kann weitere VHDL-Module als Untermodule aufrufen
 - Aufgerufene Untermodule werden als **component** bezeichnet
- Jedes VHDL-Modul kann als Untermodul benutzt werden
 - Es ist keine spezielle Beschreibung nötig
- Ein Untermodule erhält Signale als Eingangswerte und gibt Signale aus Ausgangswerte zurück
- Die Verarbeitung in den Untermodulen erfolgt parallel zu der Funktion in den aufrufenden Modulen

Beispiel:

- Die Addition dreier Zahlen soll durch zwei Addierer mit Überlauferkennung (siehe Beispiel weiter vorne) erfolgen

Beispiel: Aufruf von Untermodulen (Teil 1)

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

entity adder3 is
  port ( op_x      : in  std_logic_vector(7 downto 0);
        op_y      : in  std_logic_vector(7 downto 0);
        op_z      : in  std_logic_vector(7 downto 0);
        sum_xyz    : out std_logic_vector(7 downto 0);
        overflow   : out std_logic);
end adder3;

architecture structure of adder3 is
  signal sum_xy      : std_logic_vector(7 downto 0);
  signal overflow0    : std_logic;
  signal overflow1    : std_logic;

begin
  [...]
```

Im Namen der Architecture wird (optional) angedeutet, dass Untermodule aufgerufen werden

Definition interner Signale

Beispiel: Aufruf von Untermodulen (Teil 2)

Aufruf des ersten Untermoduls mit eindeutiger Bezeichnung „i_0“

Das Untermodul befindet sich in der Standard-Bibliothek „work“

Signale des Untermoduls (links) werden mit Signalen des aufrufenden Moduls (rechts) verbunden

Ein zweiter Addierer (Bezeichnung „i_1“) wird aufgerufen und addiert die Summe des ersten Addierers mit dem dritten Operanden

Die Überlaufsignale beider Untermodule werden zusammengefasst

```
[...]  
begin  
  
i_0: entity work.adder  
  port map (  
    op_a      => op_x,  
    op_b      => op_y,  
    sum       => sum_xy,  
    overflow  => overflow0);  
  
i_1: entity work.adder  
  port map (  
    op_a      => op_z,  
    op_b      => sum_xy,  
    sum       => sum_xyz,  
    overflow  => overflow1);  
  
overflow <= overflow0 or overflow1;  
end structure;
```

Beispiel: Aufruf von Untermodulen (komplett)

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

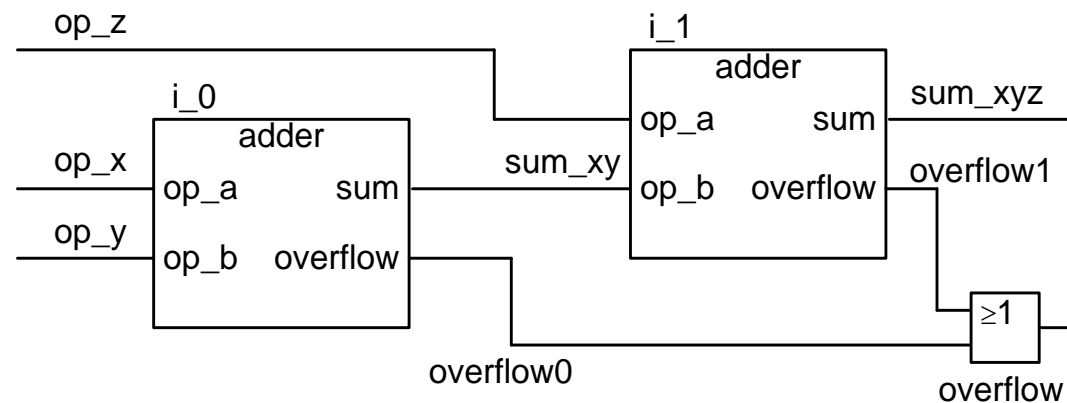
entity adder3 is
    port ( op_x      : in  std_logic_vector(7 downto 0);
          op_y      : in  std_logic_vector(7 downto 0);
          op_z      : in  std_logic_vector(7 downto 0);
          sum_xyz    : out std_logic_vector(7 downto 0);
          overflow   : out std_logic);
end adder3;

architecture structure of adder3 is
    signal sum_xy      : std_logic_vector(7 downto 0);
    signal overflow0    : std_logic;
    signal overflow1    : std_logic;

begin
    i_0: entity work.adder port map (
        op_a  => op_x,
        op_b  => op_y,
        sum   => sum_xy,
        overflow => overflow0);
    i_1: entity work.adder port map (
        op_a  => op_z,
        op_b  => sum_xy,
        sum   => sum_xyz,
        overflow => overflow1);

    overflow <= overflow0 or overflow1;
end structure;
```

Blockschaltbild für adder3



3.8 Entwurf von Automaten mit VHDL

- Bisher wurden nur VHDL-Konstrukte für **kombinatorische** Schaltungen vorgestellt
- Für den Entwurf von Automaten müssen **Flip-Flops** beschrieben werden können
- VHDL beschreibt dies über einen **Prozess für sequenzielle Funktionalität**
 - Die Daten werden hier mit der **steigenden Taktflanke** übernommen
(Genauso gut könnte die fallende Flanke benutzt werden)

```
signal a , b : std_logic;  
[...]
```

```
process  
begin
```

```
    wait until rising_edge(clk);
```

```
    b <= a;
```

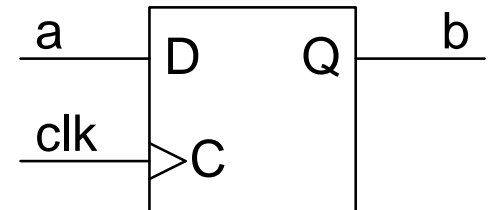
```
end process;
```

Der Prozess wird ohne Sensitivity-Liste aufgerufen

Steigende Taktflanke

Der Wert ‚b‘ wird für einen Taktperiode gespeichert.
Dies entspricht einem Flip-Flop

- Syntheseprogramme erkennen die besondere Beschreibung mit dem Schlüsselwort „rising_edge“
- Aus der Beschreibung wird ein Flip-Flop generiert



Getaktete Prozesse

- Prozesse für sequenzielle Funktionalität werden auch als **getaktete Prozesse** bezeichnet
 - In einem getakteten Prozess können **einfache Signalzuweisungen** erfolgen
 - In einem getakteten Prozess können jedoch auch logische und arithmetische Operationen durchgeführt werden

```
signal count : unsigned(3 downto 0);  
[...]  
process  
begin  
    wait until rising_edge(clk);  
  
    count <= count + 1;  
end process;
```

Unsigned kann Addition mit Integer ausführen

Getaktete Prozesse (II)

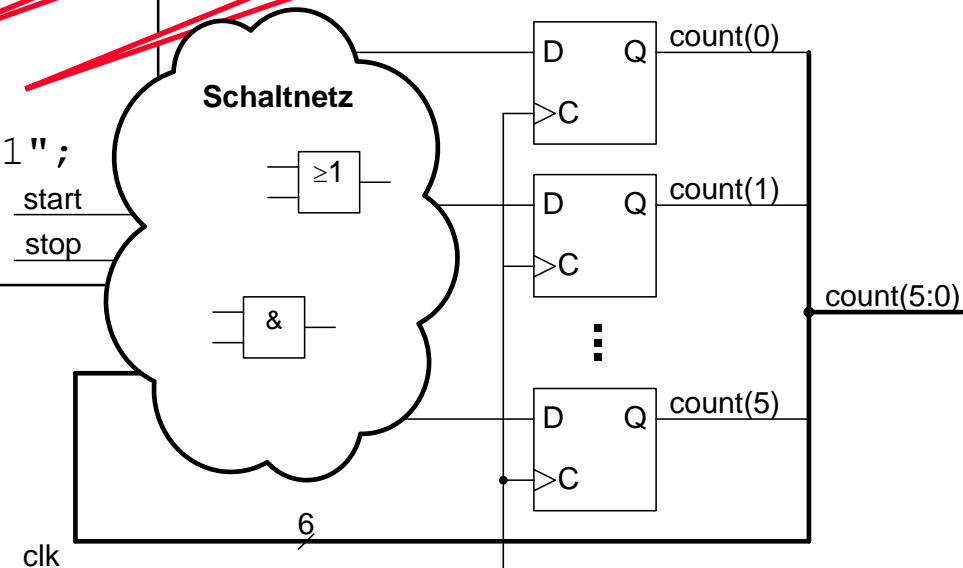
- In einem getakteten Prozess können auch komplexe Abfragen erfolgen

```
signal count : unsigned(5 downto 0);  
[...]  
process  
begin  
    wait until rising_edge(clk);  
  
    if (start = '1') then  
        count <= "000000";  
    elsif (stop = '1') then  
        count <= count;  
    else  
        count <= count + "000001";  
    end if;  
end process;
```

Zählerstand auf Null setzen

Zähler anhalten

Zähler zählt
aufwärts



- Dies entspricht einer Schaltung mit einigen Gattern und 6 Flip-Flops (für 6 bit des Signals „count“)

Beispiel: Erkennung eines Bitmusters

- Auf dem Eingang A werden serielle Daten übertragen
- Wenn vier aufeinander folgende Werte das Muster „0110“ ergeben, soll der Automat am Ausgang Q den Wert ,1', sonst ,0' ausgeben
- Umsetzung:
 - Die Eingangsdaten werden in ein Schieberegister geschrieben
 - Die Daten des Schieberegisters werden mit dem Muster verglichen

```
signal shift : std_logic_vector(3 downto 0);  
[...]  
process  
begin  
    wait until rising_edge(clk);  
  
    shift <= shift(2 downto 0) & a;  
    if (shift = "0110") then  
        q <= '1';  
    else  
        q <= '0';  
    end if;  
end process;
```



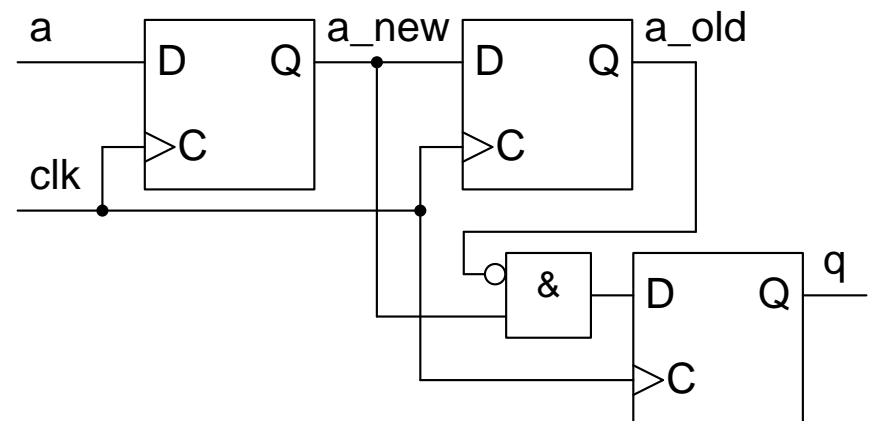
Schieberegister

Beispiel: Flankenerkennung

- Der Eingang A ist mit einem Schalter verbunden
- Beim Drücken des Schalters soll für einen Takt ein Puls $Q = '1'$ ausgegeben werden
- Umsetzung:
 - Der Eingang A wird zunächst mit dem Takt übernommen
 - Der getaktete Eingangswert wird mit dem vorherigen Wert verglichen
 - **Achtung:** Für den Eingang „a“ kein „rising_edge“ verwenden
 - ➔ „a“ ist kein Takt, und „rising_edge“ erzeugt einen Takteingang am FF

```
process
begin
    wait until rising_edge(clk);

    a_old <= a_new;
    a_new <= a;
    if ( (a_old='0') and
        (a_new='1') ) then
        q <= '1';
    else
        q <= '0';
    end if;
end process;
```



Beschreibung eines Resets

- Synchroner Reset wird durch if-Abfrage direkt nach dem „rising_edge“-wait beschrieben

Synchroner Reset

```
process
begin
    wait until rising_edge(clk);

    if (reset='1') then
        count <= "0000";
    else
        count <= count + "0001";
    end if;
end process;
```


Asynchroner Reset ist ebenfalls möglich, für „den Anfang“ sollte der synchrone Reset benutzt werden, da die Syntax etwas weniger fehleranfällig ist. ☺

„Gated Clocks“

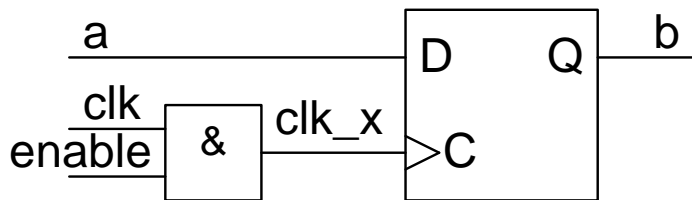
- Übergänge zwischen verschiedenen Takten sind problematisch und müssen besonders sorgfältig entworfen werden
- Von anderen Takten abgeleitete Takte („Gated Clocks“) sind fast immer unnötig und sollten vermieden werden
 - Ausnahmen: Abgeleiteter Takt für die gesamte Schaltung und „Low Power“

Falsch: „Gated Clock“

```
clk_x <= clk and enable;
```

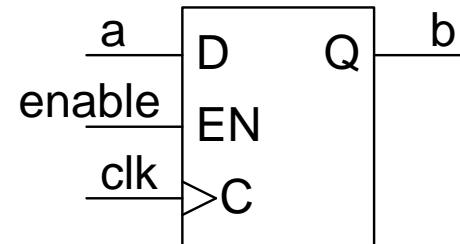


```
process  
begin  
    wait until rising_edge(clk_x);  
  
    b <= a;  
end process;
```




Richtig: FF mit Enable

```
process  
begin  
    wait until rising_edge(clk);  
  
    if (enable = '1') then  
        b <= a;  
    end if;  
end process;
```



Latches

- Datenspeicherung sollte nur mit getakteten Flip-Flops erfolgen
 - Ungetaktete Speicherelemente werden als **Latch** bezeichnet
- Latches können versehentlich auftreten, bei ungewollter Datenspeicherung in kombinatorischen Prozessen



```
process (a,b,c,d)
begin
    if      (a = "00") then
        y <= b;
    elsif (a = "01") then
        y <= c;
    elsif (a = "10") then
        y <= d;
    end if;
end process;
```

- Was passiert bei (a = "11")?
→ **Fehler:** Der Wert von y wird gespeichert

Latches (II)

- Vermeidung von Latches durch:
 - Definition aller Eingangsmöglichkeiten mit „else“
 - Angeben eines „Default“-Wertes
- Beachten Sie für das zweite Beispiel:
 - Die Signalzuweisungen werden bis zum Ende der Prozessbeschreibung **vorgemerkt** und erst dann ausgeführt (siehe oben)
 - Bei A=„00“ wird der Wert Y also **nicht** kurzzeitig ‚0‘ und dann B

```
process (a,b,c,d)
begin
    if      (a = "00") then
        y <= b;
    elsif  (a = "01") then
        y <= c;
    else
        y <= d;
    end if;
end process;
```

```
process (a,b,c,d)
begin
    y <= '0'; -- default
    if      (a = "00") then
        y <= b;
    elsif  (a = "01") then
        y <= c;
    elsif  (a = "10") then
        y <= d;
    end if;
end process;
```

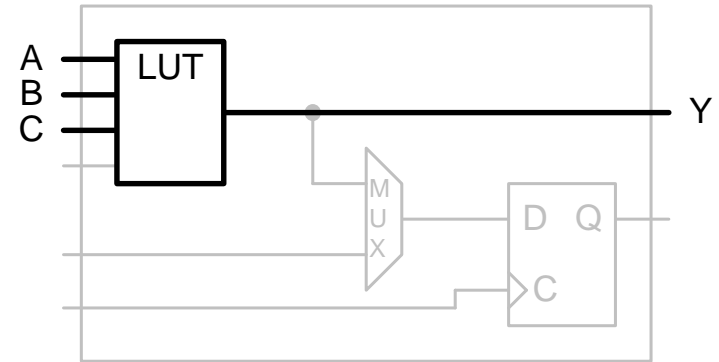
Anhang: Praktikumsversuche mit FPGAs

- Der Schaltungsentwurf erfolgt in der Hardwarebeschreibungssprache VHDL
- Die Umsetzung der VHDL-Beschreibung in eine FPGA-Schaltung erfolgt mittels EDA-Tools (Electronic Design Automation)

Beispiel:

- Die logische Funktion $Y = (A \& B) \vee C$ soll in ein FPGA umgesetzt werden
- Der VHDL-Code lautet:

```
y <= (a and b) or c;
```
- Ein EDA-Tool kann die Funktion in eine LUT umsetzen
- Das Flip-Flop bleibt ungenutzt oder kann von anderen Schaltungsteilen verwendet werden



Prinzipieller Schaltungsentwurf

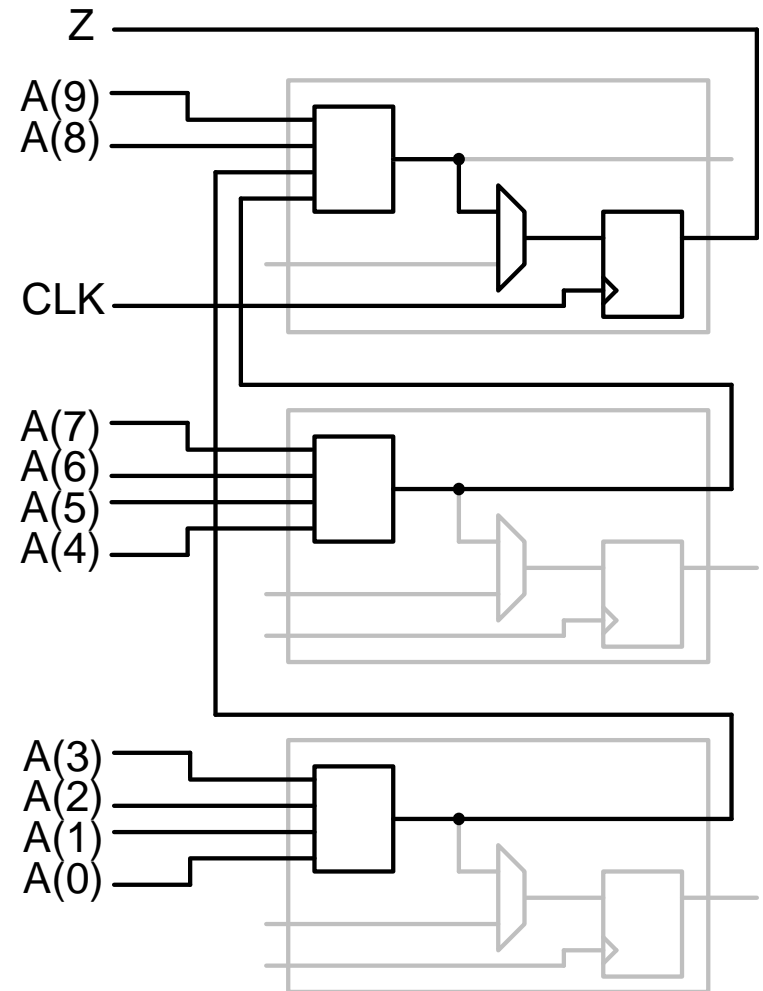
- Komplexere Logikfunktionen werden **automatisch** auf mehrere LUTs und LEs aufgeteilt

Beispiel:

- Es soll überprüft werden, ob beim 10-bit Wert A alle Stellen 1 sind; das Ergebnis soll in einem Flip-Flop gespeichert werden
- Die Funktion benötigt drei LUTs, ein FF und entspricht einem 10-Input UND

```
process
begin
    wait until rising_edge(clk);

    if (a="1111111111") then
        z <= '1';
    else
        z <= '0';
    end if;
end process;
```



Entwurfsablauf

Zwischen **Spezifikation** und **Inbetriebnahme** einer FPGA-Schaltung erfolgt der Entwurf in folgenden Entwurfsschritten:

- Eingabe des VHDL-Codes
 - Simulation (siehe Kapitel 8)
 - Synthese, Technologie-Mapping
 - Eingabe von „Constraints“ (Lage der Ein-/Ausgänge, Timing, ...)
 - Placement, Routing
 - Überprüfen des Timings
 - Erzeugen der Programmierdateien
- Dieser gesamte Ablauf wird als „**Designflow**“ bezeichnet
 - Der prinzipielle Ablauf ist für alle FPGAs, auch verschiedener Hersteller, gleich.
 - Hier wird das „Altera Quartus II“ für MAX 10 FPGAs verwendet
 - Nur mit dem Verständnis des Designflows ist eine effiziente Schaltungsentwicklung und eine Interpretation von Fehlermeldungen möglich
 - ➔ Versuchen Sie den Ablauf zu verstehen und drücken Sie nicht einfach Knöpfe

Entwicklungsumgebung

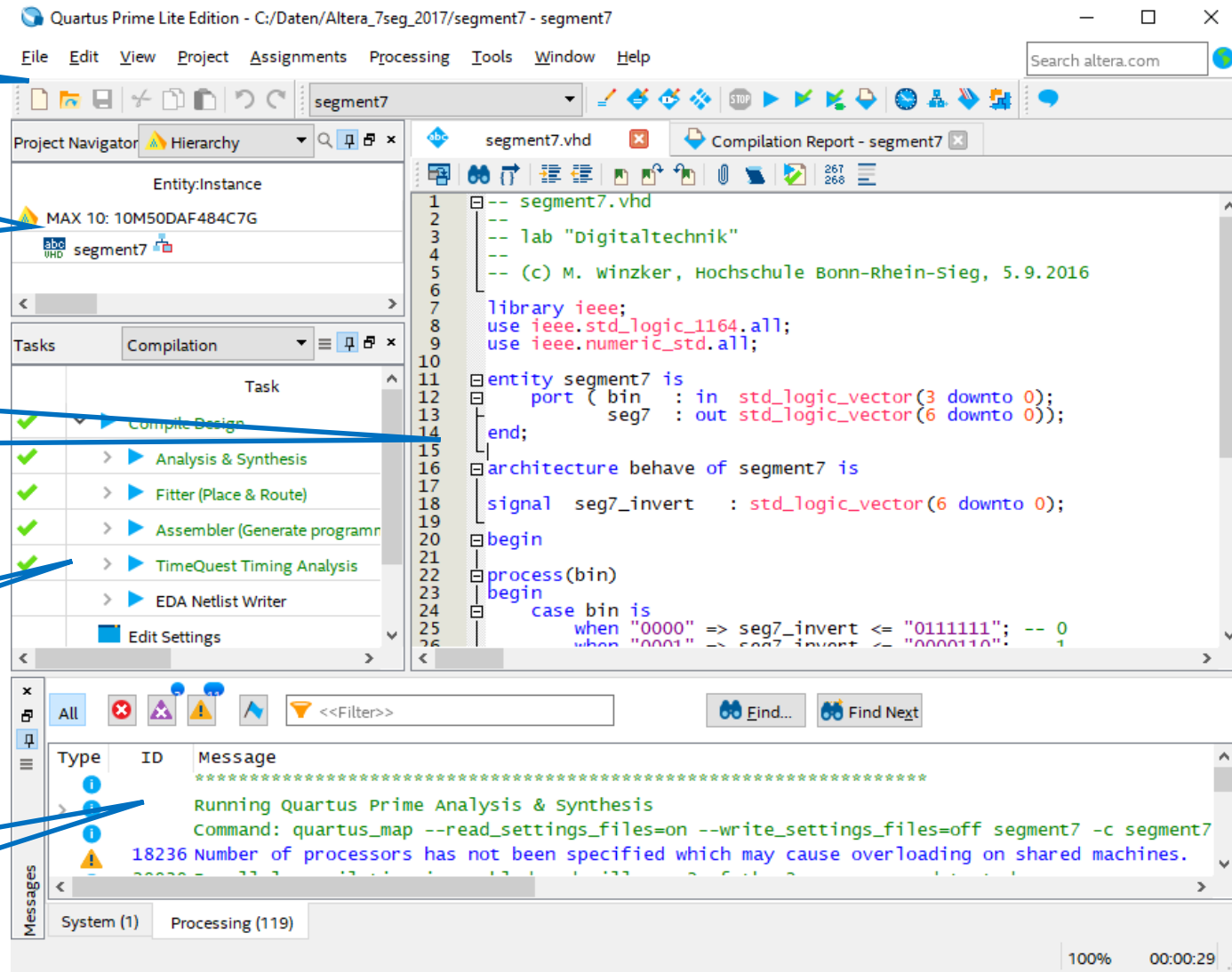
Funktionen

Hierarchie und
Liste der
Quelldateien

geöffnete
Quelldatei,
z.B. VHDL-Code

Bearbeitungs-
schritte -
„Designflow“

Ausgabe von
Meldungen



(Screenshot Quartus 17)

FPGA-Platine

USB-Port für Stromversorgung
und Programmierung

Erweiterungsports

Steckverbinder für
Arduino-Erweiterungen

Beschleunigungs-
Sensor

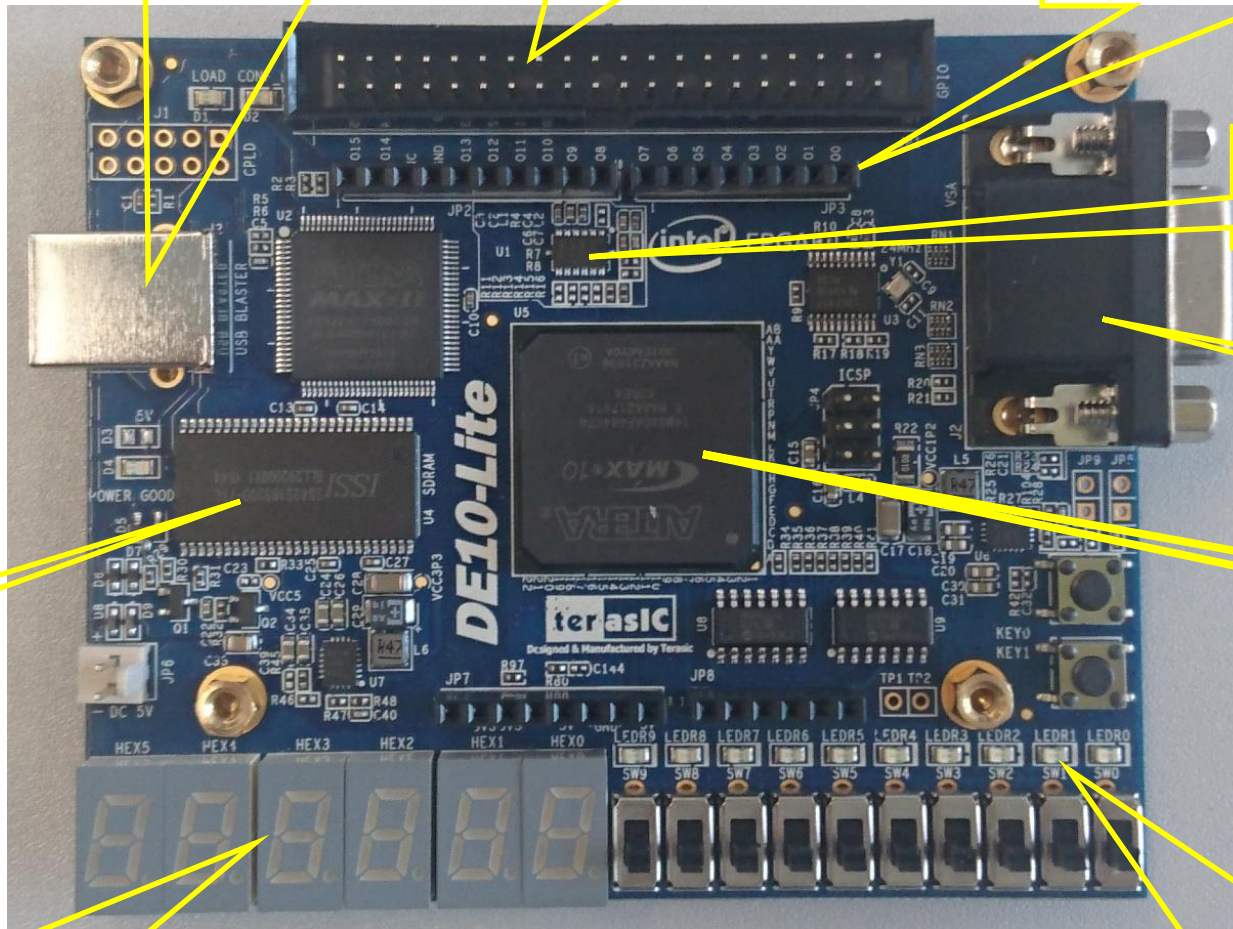
VGA-Ausgang

FPGA

DRAM

7-Segment-Anzeigen

LEDs, Taster, Schalter



Eingabe des VHDL-Codes

Als Beispiel dient eine Schaltung zur Ansteuerung der 7 Segment-Anzeige

- Der Wert `bin(3:0)` gibt eine Binärzahl an
- Die Binärzahl soll als 7 Segment-Anzeige `seg7(6:0)` ausgegeben werden
- Implementierung über Case-Anweisung
 - Wozu dient das Signal „seg7_invert“?

```
-- segment7.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity segment7 is
    port ( bin      : in  std_logic_vector(3 downto 0);
          seg7      : out std_logic_vector(6 downto 0));
end segment7;

architecture behave of segment7 is
    signal seg7_invert : std_logic_vector(6 downto 0);
begin
    [...]
```

Eingabe des VHDL-Codes (II)

```
[...]
process(bin)
begin
  case bin is
    when "0000" => seg7_invert <= "0111111"; -- 0
    when "0001" => seg7_invert <= "0000110"; -- 1
    when "0010" => seg7_invert <= "1011011"; -- 2
    when "0011" => seg7_invert <= "1001111"; -- 3
    when "0100" => seg7_invert <= "1100110"; -- 4
    when "0101" => seg7_invert <= "1101101"; -- 5
    when "0110" => seg7_invert <= "1111101"; -- 6
    when "0111" => seg7_invert <= "0000111"; -- 7
    when "1000" => seg7_invert <= "1111111"; -- 8
    when "1001" => seg7_invert <= "1101111"; -- 9
    when others => seg7_invert <= "1000000"; -- '-' for rest
  end case;
end process;

seg7 <= not seg7_invert;

end behave;
```

Eingabe von Randbedingungen („User Constraints“)

- Die Randbedingungen für die Erstellung der Schaltung werden (mit den Projekteigenschaften) in einer eigenen Datei gespeichert
 - Bei Altera hat diese Datei die Endung QSF
 - Für Praktikumsboard „DE10_Lite_Default.qsf“ (Versuch 1: „Segment7_Pin.qsf“)
 - Der VHDL-Code enthält nur die Schaltungsbeschreibung
- An Constraints können angegeben werden:
 - Lage der IO-Pins
 - Für Inbetriebnahme unbedingt erforderlich!
 - Menu-Punkt „Pin-Planner“ oder Einlesen aus Datei
 - Timing, insbesondere Taktfrequenz
 - Empfohlen (Assignments -> “TimeQuest Wizard”)
 - Kann bei einfachen oder langsamen Schaltungen (ca. <30 MHz) weggelassen werden
 - Fläche
 - Meist nicht erforderlich
 - Hilft der Software bei hohen Geschwindigkeiten (ca. >100 MHz)

Design-Flow für 7-Segment-Anzeige

- Neues Projekt erstellen
 - File -> New Projekt Wizard
 - Projektname „segment7“
 - segment7.vhd und Segment7_Pin.qsf in Projektverzeichnis kopieren
 - Add File „segment7.vhd“
 - FPGA auswählen: DE10-Lite enthält 10M50DAF484C7G
 - 10M = MAX 10 Familie
 - 50 = Größe von 50K Elementen
 - DA = Dual Power Supply, Analog Features
 - F484 = Gehäuse FBGA mit 484 Pins
 - C = Commercial
 - 7 = Geschwindigkeit
 - G = Serienfertigung, bleifreies Lot
 - Keine weiteren Einstellungen erforderlich
- Nach Projekterstellung, Constraints-File einlesen
 - Assignments -> Import Assignments -> Segment7_Pin.qsf

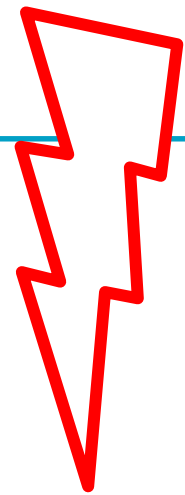
Design-Flow für 7-Segment-Anzeige (II)

- Aufruf der Entwurfsschritte über „Compile Design“
 - Analysis and Synthesis: VHDL-Datei wird in FPGA-Elemente übersetzt
 - Fitter (Place and Route): FPGA-Elemente werden auf dem FPGA positioniert und die Verbindungsleitungen werden ermittelt
 - Assembler: Übersetzung in Programmierdaten
 - Timing Analyzer: Analyse der Laufzeiten
 - EDA Netlist Writer: Nicht erforderlich (Interface für externe Programme)
- Program Device öffnet Programm zur Board-Programmierung
 - Design kann verwendet werden
- Compilation Report zeigt an:
 - 8 LUTs erforderlich für 7 Werte
 - 11 Pins verwendet: 4 Eingänge (Switches), 7 Ausgänge (7 Segmente)
 - Auslastung insgesamt 0 bis 3%
- Video: <http://youtu.be/3LJfGwNh3JA>

Hinweise zu Soft- und Hardware

- Im Praktikum stehen FPGA-Boards und Rechner zur Verfügung
 - Benutzung mit Ihrem **Hochschulaccount**
- Die EDA-Software ist kostenfrei erhältlich bei www.altera.com
 - Design Tools & Services -> Design-Software
 - Quartus II Web Edition Software (ca. 4 GB)
 - Device-Files für MAX 10
- Mehrere FPGA-Boards sind bei der **Zentralausleihe** erhältlich
 - Stromversorgung erfolgt über USB-Kabel
 - Installation der Treiber manchmal problematisch:
 - Im Geräte-Manager USB-Controller auswählen, „Treibersoftware aktualisieren“, „Aus einer Liste von Gerätetreibern auswählen“
 - Datenträger: Altera-Verzeichnis\...\drivers\usb-blaster\usbbbst.inf
 - Installation des USB-Blaster akzeptieren

Vorbereitung und Plagiate



Bitte bereiten Sie sich auf das Praktikum vor.

- Für ersten FPGA-Versuch ist die vorhandene VHDL-Datei zu erweitern
 - Die Vorlage berücksichtigt Zahlen von 0 bis 9
 - Erweitern Sie den Code mit Buchstaben A, b, C, d, E, F für die Zahlen bis 15

Plagiate

- Die Aufgaben im Praktikum sind für alle Gruppen gleich
- Bitte kopieren Sie keine Quelltexte von anderen Gruppen
 - Das Praktikum dient für Ihr Verständnis
 - Nur durch eigenes Programmieren gewinnen Sie dieses Verständnis
- Das Praktikum findet in Zweiergruppen statt
- Verlassen Sie sich nicht auf Ihren Praktikumpartner
 - Falls Ihr Partner bei einem Termin krank ist, müssen Sie auch zurechtkommen

Remote-Lab

Die TU Ilmenau hat das Remote-Lab „GOLDi“, mit dem Sie den Laborversuch ebenfalls ausprobieren können: <http://www.goldi-labs.net/>

- Registrieren
- IUT (Germany)
- Digital Demo Board (Real)

- Der Versuch verwendet ein 5M1270ZT144C5 aus der MAX V Familie
 - Device-Files für MAX V installieren
- I/O-Pins definiert in Datei: GOLDi_Digital_Demo_Board.qsf
 - Name der I/O-Pins müssen angepasst werden
 - Schiebeschalter mit Bezeichnung: Switch(7:0)
 - Vier Siebensegment-Anzeigen mit Namen: Hex0(7:0) bis Hex3(7:0)
- Bedienung des Remote-Lab:
 - Oben Binary hochladen „📁“: *.pof Dateien im Projektordner, output_files
 - Unten Versuchsablauf starten „▶“