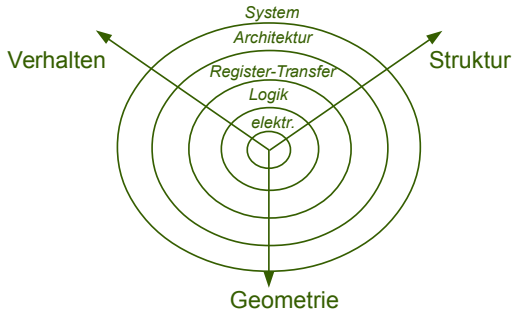
2. Hardware-Modellierung mit VHDL

2.1.1. Einführung

- **Gesucht für die Hardware-Modellierung:**
  - ***Einheitliche*** Beschreibungssprache für Hardware
    - ... aus unterschiedlichen Sichten
    - ... auf verschiedenen Ebenen



Anwendungen

- **Simulation**
  - Validierung von *logischem* und/oder *zeitlichem* Verhalten
- **Verifikation**
  - Beweis, ob Spezifikation und Implementierung übereinstimmen
- **Synthese**
  - automatische Übersetzung einer Verhaltens- in eine Strukturbeschreibung
- **Platzierung & Verdrahtung**

Spezielle Anforderungen
(von Programmiersprachen i.d.R. nicht erfüllt)

a) Modellierung von Verzögerungszeiten

b) Parallelität

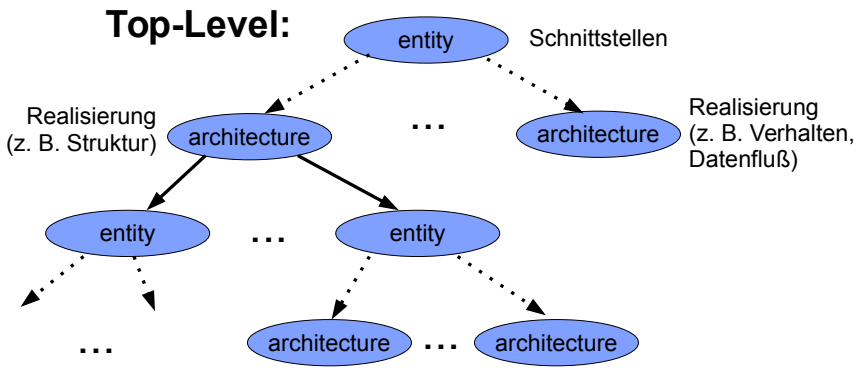
c) Beschreibung von Strukturen, Netzlisten

VHDL

- **1980: VHSIC-Projekt**
  - Very High Speed Integrated Circuits
- **1983: VHDL**
  - VHSIC Hardware Description Language
- **1987 (1993, 2000, 2002): IEEE-Standard**

- **Eine Sprache für verschiedene Sichten & Ebenen**
  - erlaubt Mischen von Beschreibungsstilen
    => kein Aufwand für Anpassungen
  - enthält alle Elemente einer (imperativen) Programmiersprache

Struktur eines VHDL-Projektes



Mögliche Inhalte einer Architektur

parallele Anweisungen

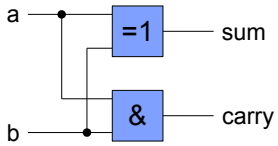
```
architecture TEST of EXAMPLE is
  -- Signal-Deklarationen etc.
begin

  -- a) Prozesse -> Verhalten
  process
  begin
    -- sequentielle Anweisungen
  end process;

  -- b) Parallele Signalzuweisungen -> Datenfluss
  x <= y;    -- Signalzuweisung (parallel)
  ...

  -- c) Komponenten-Instanziierungen -> Struktur
  I_AND2: AND2 port map (...)
  ...

end TEST;
```


Beispiel: Halbaddierer

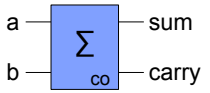
Funktionstabelle

a	b	sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Schaltbild



Schaltsymbol



- **Schnittstelle:**

```
entity HALF_ADDER is
  port (a, b: in STD_LOGIC; sum, carry: out STD_LOGIC);
end HALF_ADDER;
```

a) Verhaltensbeschreibung mit Prozess

```
architecture BEHAVIOR of HALF_ADDER is
begin

  process (a, b)
    variable tmp: STD_LOGIC_VECTOR (1 downto 0);
  begin
    tmp := ('0' & a) + ('0' & b);
    sum <= tmp(0);
    carry <= tmp(1);
  end process;

end BEHAVIOR;
```

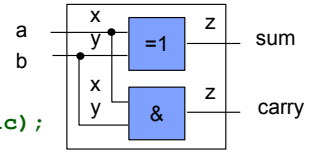

b) Datenflussbeschreibung
mit parallelen Zuweisungen

```
architecture DATAFLOW of HALF_ADDER is
begin
  sum <= a xor b;
  carry <= a and b;
end DATAFLOW;
```

- **Verhaltens- und Datenflussbeschreibungen können auch Zeitverhalten modellieren:**

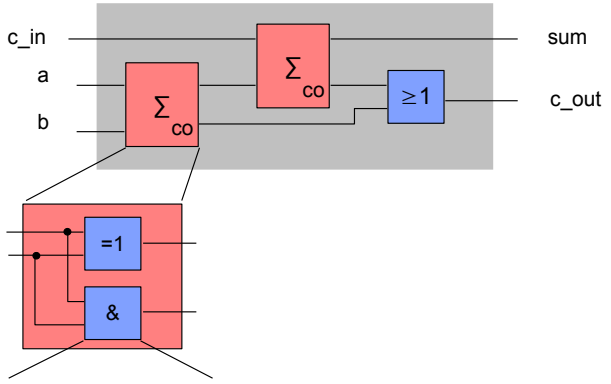
```
architecture TIMED_DATAFLOW of HALF_ADDER is
begin
  sum <= a xor b after 2 ns;
  carry <= a and b after 1 ns;
end TIMED_DATAFLOW;
```


c) Strukturbeschreibung




```
architecture STRUCTURE of HALF_ADDER is

  component XOR2
    port (x, y: in std_logic; z: out std_logic);
  end component;

  component AND2
    port (x, y: in std_logic; z: out std_logic);
  end component;

  for I0: XOR2 use entity WORK.XOR2(DATAFLOW);
  for I1: AND2 use entity WORK.AND2(DATAFLOW);

begin
  I0: XOR2 port map(x => a, y => b, z => sum);
  I1: AND2 port map(x => a, y => b, z => carry);
end STRUCTURE;
```


Hierarchischer Entwurf
Verwendung eigener Entwürfe als Bauelemente




```
entity AND2 is
  port (x, y: in std_logic; z: out std_logic);
end AND2;
```

```
entity FULL_ADDER is
  port (a, b, c_in: in std_logic; sum, c_out: out std_logic);
end FULL_ADDER;


architecture STRUCTURE of FULL_ADDER is

  component HALF_ADDER
    port (a, b: in std_logic; sum, carry: out std_logic);
  end component;

  component OR2
    port (x, y: in std_logic; z: out std_logic);
  end component;

  signal ha0_sum, ha0_carry, ha1_carry: std_logic;

begin
  I_HA0: HALF_ADDER port map(a => a, b => b,
                             sum => ha0_sum, carry => ha0_carry);
  I_HA1: HALF_ADDER port map(a => ha0_sum, b => c_in,
                             sum => sum, carry => ha1_carry);
  I_OR: OR2 port map(x => ha0_carry, y => ha1_carry, z => c_out);
end;
```

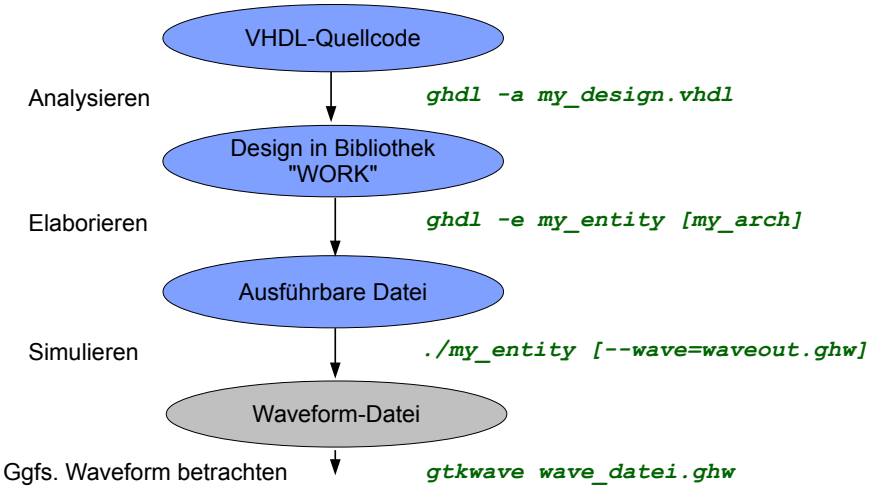

Geläufige Beschreibungs-Stile

- **Verhalten *(BEHAVIOR)***
  - Spezifikation auf Architektur-/Systemebene
  - Modellierung von Bibliothekszellen inkl. Zeitverhalten
- **Synthetisierbare Verhaltensbeschreibung *(RTL)***
  - verwendet nur Konstrukte, die automatisch in eine Struktur übersetzt werden können
  - keine Modellierung von Zeit ("after"-Klausel)
- **Strukturbeschreibung *(STRUCTURE)***
  - Ausgabe aus der Synthese
  - Verknüpfung von System-Modulen
- **Testbench *(TESTBENCH)***
  - Umgebung für einen zu validierenden Entwurf
  - erzeugt Stimuli und wertet Antworten aus

2.1.2. Simulation mit GHDL

- **Was ist GHDL?**
  - Simulator (Compiler) für VHDL
  - basiert auf GCC, Open Source
  - Arbeitsweise
    - compilierte Simulation mit Ereignissteuerung
    - erzeugt ausführbare Datei, die den Entwurf nachbildet
- **Info & Download:** *http://ghdl.free.fr*

Arbeitsschritte



Ein-/Ausgabe von Signalwerten

- **Problem**
  - Kommunikation mit den "Pins" des Entwurfs
    - Eingabe-Ports bleiben undefiniert
    - standardmäßig keine Ausgabe
- **Möglichkeiten**
  1. Ausgabe der Signale in eine Waveform-Datei
     - Option "--wave"
     - Betrachten z. B. mit GtkWave
     - Nützlich: Waveform-Datei enthält *alle* Signale (auch interne)
  2. Testbench

Testbenches

- Manuelles Erstellen von **Simulations-Stimuli** sehr aufwendig (sofern vom Simulator unterstützt)
- **Kontrolle der Ergebnisse** im Waveform-Editor sehr aufwendig und fehleranfällig

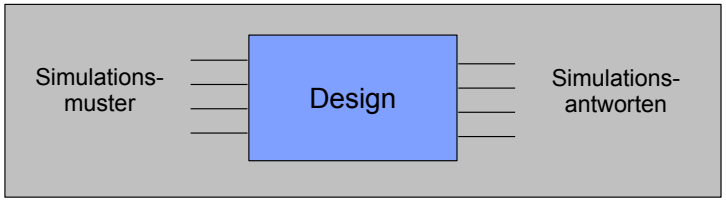
=> Idee:

- Nutze Mächtigkeit von VHDL, um Simulations- und Testprogramme zu erstellen

Eine *Testbench* ...

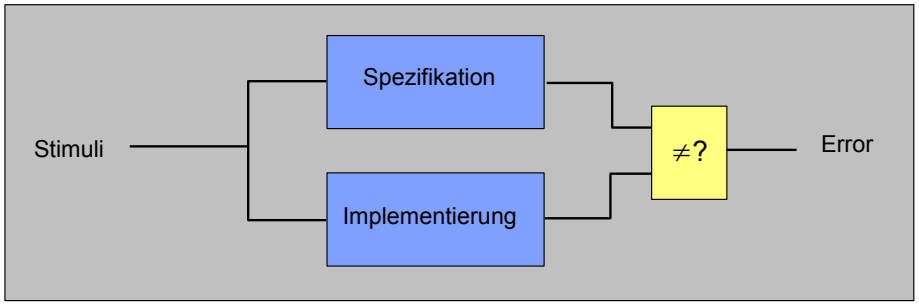
- ist eine Entity *ohne eigene Ein-/Ausgänge.*
- enthält einen Prozess (Verhaltens-Stil), der Simulationsmuster erzeugt und Antworten automatisch auswertet.
- instantiert den Schaltungsentwurf im Struktur-Stil.

Variante 1: Einfache Instanziierung



- **Testbench übernimmt Auswertung der Antworten, z. B.**
  - Vergleich mit Soll aus Tabelle
  - Plausibilitäts-Checks
  - Schreiben in Datei & Auswertung später

Variante 2: Vergleich alternativer Architekturen



- **Testbench muss nur Stimuli erzeugen**
  - ***Frage: Welche?***
    - Vollständige Simulation i. d. R. nicht möglich

Beispiel: Testbench für den Halbaddierer

a) Einfache Testbench

```
entity HALF_ADDER_TB is
end HALF_ADDER_TB;

architecture TESTBENCH1 of HALF_ADDER_TB is

  -- Component declaration
  component HALF_ADDER is
    port (a, b: in std_logic; sum, carry: out std_logic);
  end component;

  -- Configuration...
  for IMPL: HALF_ADDER use entity WORK.HALF_ADDER(BEHAVIOR);

  -- Internal signals...
  signal a, b, sum, carry: std_logic;

...
```



```
...

begin

  -- Instantiate half adder
  IMPL: HALF_ADDER port map (a => a, b => b,
                             sum => sum, carry => carry);

  -- Main process...
  process
  begin
    a <= '0'; b <= '0';
    wait for 1 ns;      -- wait a little bit
    assert sum = '0' and carry = '0' report "0 + 0 is not 0/0!";

    a <= '0'; b <= '1';
    wait for 1 ns;      -- wait a little bit
    assert sum = '1' and carry = '0' report "0 + 1 is not 1/0!";

...
```

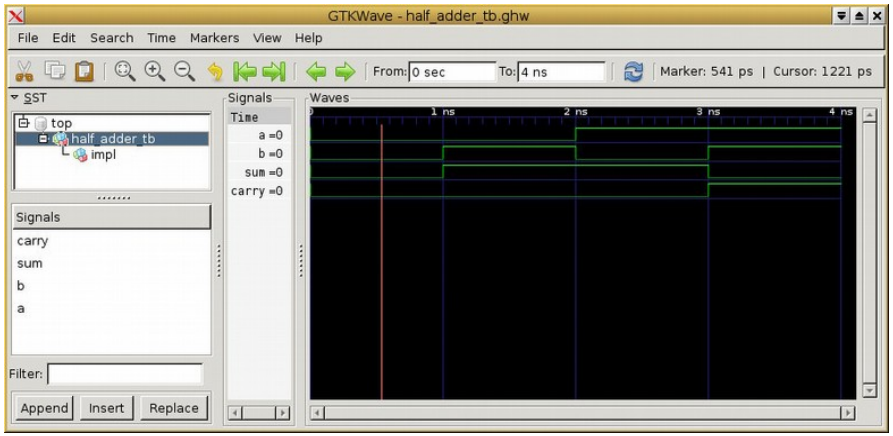


```
...

    a <= '1'; b <= '0';
    wait for 1 ns;      -- wait a little bit
    assert sum = '1' and carry = '0' report "1 + 0 is not 1/0!";

    a <= '1'; b <= '1';
    wait for 1 ns;      -- wait a little bit
    assert sum = '0' and carry = '1' report "1 + 1 is not 0/1!";

    -- Print a note & finish simulation now...
    assert false report "Simulation finished" severity note;
    wait;               -- end simulation

  end process;

end architecture;
```


Beispiel-Lauf mit GHDL

```
> ghdl -a and2.vhdl xor2.vhdl half_adder.vhdl half_adder_tb.vhdl
> ghdl -d
entity half_adder
architecture structure of half_adder
architecture behavior of half_adder
architecture dataflow of half_adder
architecture timed_dataflow of half_adder
entity and2
architecture dataflow of and2
entity xor2
architecture dataflow of xor2
entity half_adder_tb
architecture testbench1 of half_adder_tb
> ghdl -e half_adder_tb
> ./half_adder_tb --wave=half_adder_tb.ghw
half_adder_tb.vhdl:99:5:@4000000:(assertion note): Simulation finished
> gtkwave half_adder_tb.ghw half_adder_tb.save
```

GtkWave



b) Testbench zum Vergleich von Verhalten & Struktur

```
architecture TESTBENCH2 of HALF_ADDER_TB is

  -- Component declaration...
  component HALF_ADDER is
    port (a, b: in std_logic; sum, carry: out std_logic);
  end component;


  -- Configuration...
  for SPEC: HALF_ADDER use entity WORK.HALF_ADDER(BEHAVIOR);
  for IMPL: HALF_ADDER use entity WORK.HALF_ADDER(STRUCTURE);


  -- Internal signals...
  signal a, b, sum_spec, carry_spec, sum_impl, carry_impl: std_logic;

...
```



```
...

begin

  -- Instantiate half adder...
  SPEC: HALF_ADDER port map (a => a, b => b, sum => sum_spec,
                             carry => carry_spec);
  IMPL: HALF_ADDER port map (a => a, b => b, sum => sum_impl,
                             carry => carry_impl);

  -- Main process...
  process
  begin
    a <= '0'; b <= '0';
    wait for 1 ns;      -- wait a little bit
    assert sum_spec = sum_impl and carry_spec = carry_impl
        report "Specification and implementation differ!";

    a <= '0'; b <= '1';
    wait for 1 ns;      -- wait a little bit
    assert sum_spec = sum_impl and carry_spec = carry_impl
        report "Specification and implementation differ!";

...
```



```
...

    a <= '1'; b <= '0';
    wait for 1 ns;      -- wait a little bit
    assert sum_spec = sum_impl and carry_spec = carry_impl
        report "Specification and implementation differ!";

    a <= '1'; b <= '1';
    wait for 1 ns;      -- wait a little bit
    assert sum_spec = sum_impl and carry_spec = carry_impl
        report "Specification and implementation differ!";

    -- Print a note & finish simulation now
    assert false report "Simulation finished" severity note;
    wait;               -- end simulation

  end process;

end architecture;
```

2.1.3. Hinweise zum Praktikum

- **Umgebungsvariablen**
  - In '.bashrc' einfügen ('bash' als Shell erforderlich):

```
source /opt/ees/env.sh
```

- **Tools**
  - für einige Tools existieren Wrapper-Skripte, damit sie mit korrekten Parametern/Einstellungen aufgerufen werden:

```
ees-ghdl, ees-xst, ...
```


GHDL – Kurzreferenz

- **Analysieren**
  ghdl -a <Datei>
- **Elaborieren**
  *ghdl -e <Entity> *
  [<Architecture>]
- **Simulieren**
  ./mein_design [--wave=<Wave-Datei>.ghw]
- **Sonstige Kommandos**
  ghdl -d
    - listet Bibliothek auf

  ghdl --remove
    - löscht alle erzeugten Zwischendateien

- **Allgemeine Optionen (alle Kommandos)**
  --work=<Bibliothek>
    - setzt Arbeitsbibliothek (alternative zu "WORK")

  --workdir=<Pfad>
    - Verzeichnis mit Bibliothek "WORK"

  -P<Pfad>
    - Erweitert Suchpfad für Bibliotheken

- **Hilfe**
  info ghdl
  ghdl –help
  ./mein_design --help

Praktische Hinweise

- **Übersetzungsreihenfolge**
  - Entity stets vor zugehörigen Architekturen
  - Strukturbeschreibung: tiefere Hierarchiestufen zuerst übersetzen

```
eis-ghdl -a xor2.vhdl and2.vhdl half_adder.vhdl
```

  - Bei Änderungen müssen auch abhängige Module neu analysiert werden

- **Bei mehreren Architekturen in verschiedenen Dateien:**
  - Entity-Deklaration sollte nur einmal vorkommen!

- **Bei mehreren Architekturen pro Datei:**
  - library/use-Anweisungen muss vor jeder Architektur wiederholt werden

GtkWave

- **Aufruf**

```
gtkwave meine_datei.ghw [mein_save_file.sav]
```

- **Signale auswählen**
  - im linken Teilfenster
- **Sitzung speichern**
  - Menü "File -> Write Save File" (Ctrl-W)
- **Anzeigen von Bit-Vektoren**
  - Expandieren / Kombinieren: Markieren, dann
    - Menü "Edit -> Expand" (F3)
    - Menü "Edit -> Combine Up / Down" (F4, F5)
  - Zahlenformat ändern (Vektoren): Markieren, dann
    - Menü "Edit -> Data Format -> ..."

Pakete und Bibliotheken

- **Package:** enthält allgemein nützliche Elemente
  - Funktionen
  - Typdefinitionen
  - Komponenten (beschreiben Entities)
  - ...
- **Library:** Sammlung von Packages
- **Beispiel**
  - Einbinden aller Elemente des Packages "std_logic_1164" aus der Library "IEEE":

```
library IEEE;
use IEEE.std_logic_1164.all;
```


Wichtige Libraries/Packages

- **`WORK`**
  - Default-Bibliothek; alle eigenen Entwürfe landen standardmäßig hier
- **`IEEE.std_logic_1164`**
  - z. B. Typen & Funktionen für mehrwertige Logik (z. B. *std_logic*)
- **`IEEE.numeric_std`**
  - arithmetische Operationen mit mehrwertiger Logik
- **`STD.textio`**
  - Funktionen zur Textein-/-ausgabe
- **`EIS`**
  - Spezielle Elemente (z. B. Hilfsfunktionen) für *dieses* Praktikum
- **`SXLIB`**
  - Bilbiothek mit Standard-Zellen (für IC-Entwurf mit Alliance)

Bibliothek *"EIS"*

- **Hilfsprozeduren / -Funktionen**

```
function to_character (val : in std_logic) return character;
function to_string (val : in std_logic_vector) return string;
function to_string (val: in unsigned) return string;
function to_string (val: in integer) return string;
function spec_vs_impl (spec : in std_logic_vector;
                       impl : in std_logic_vector) return boolean;
```

- **Verwendung**
  - z. B. für die Ausgabe von Signalen während der Simulation

```
library EIS;
use EIS.helpers.all;
...
assert false report "a = " & to_string(a) & ", b = " & to_string(b)
       severity note;
```


2.2. VHDL als Programmiersprache

2.2.1. Lexikalische Elemente

- **Kommentare: "--"**

```
a(0 to 3) := a(3) & a(0 to 2) -- hier wird rotiert
```

- **Bezeichner**

```
<identifier> ::= <letter> { [ "_" ] <letter_or_digit> }
```

  - VHDL ist *nicht* case-sensitiv

- **Zahlen**
  - **Dezimalzahlen (Integer-, Real-):**

```
<decimal_literal> ::= <integer> [ . <integer> ]
                      [ E [ + ] <integer> | E - <integer> ]
<integer> ::= <digit> { [ "_" ] <digit> }
```

Beispiele:

```
0   1   123_456_789 987E6   -- integer literals
0.0 0.5 2.718_28    12.4E-9 -- real literals
```

  - **Zahlen zur Basis *n***

```
<based_literal> ::= <base> # <based_int> [ . <based_int> ]
                    # [ <exponent> ]
<base> ::= <integer>
<based_int> ::= <ext_digit> { [ "_" ] <ext_digit> }
```

Beispiele:

```
2#1100_0100#   16#C4#  4#301#E1   -- the integer 196
2#1.1111_1111_111#E+11 16#F.FF#E2 -- the real number 4095.0
```


- **Zeichen (und Bit-Werte)**
  - ... stehen in einfachen Anführungszeichen, z.B.

```
'A' '*' ''' ' '
```

- **Strings (und Bit-Strings)**
  - ... stehen in doppelten Anführungszeichen, z.B.

```
"A string"
""                                  -- empty string
"String in a string: ""Hello"". " -- contains quote marks
```

  - Bit-Strings können in vereinfachter Schreibweise oktal ("O") oder hexadezimal ("X") angegeben werden:

```
B"1101110" -- length is 7
O"156"     -- length is 9, equivalent to B"001_101_110"
X"6E"      -- length is 8, equivalent to B"0110_1110"
```


2.2.2. Datenobjekte und Datentypen

Datenobjekte

- **Konstanten**

```
constant zero: bit_vector(3 downto 0) := "0000";
```

- **Variablen**

```
variable counter: integer;
```

  - speichern temporäre Daten
  - nur in Prozessen ("process"-Umgebung) oder Unterprogrammen erlaubt
- **Signale**

```
signal connect: bit;
```

  - speichern Werte *mit Informationen über den zeitlichen Verlauf*
  - z. B. zum Modellieren von Verbindungsleitungen oder Registern

-> werden später ausführlich behandelt

Datentypen – Übersicht

- **Elementare Typen**
  - Aufzählungstyp
  - Vordefinierte Typen (`integer, float, character, ...`)
  - Arrays
- **Wichtige Typen aus Bibliotheken**
  - `std_logic, std_logic_vector`
- **Weitere (hier nicht behandelt):**
  - Records
  - Untertypen
  - Zeiger
  - Dateien
  - abgeleitete Typen (Arrays, Records, Untertypen)

Aufzählungstyp

```
architecture RTL of example_fsm is
  type state_type is (start, running1, running2, destination);
  signal state : state_type;
begin

  -- Prozess für den Zustandsübergang
  process (CLK)
  begin
    ...
    case state is
      when start => if (I = '1') then state <= running1;
                    else state <= destination; end if;
      when running1 => ...
      ...
    end case;
    ...
  end process;
  ...
  -- Prozess für die Ausgabe
  ...
end RTL;
```

Aufzählungstyp für Zustände eines endl. Automaten (Steuerwerk)

Vordefinierte Datentypen

- **boolean**

  ```
  type boolean is (false, true);
  ```

- **bit**

  ```
  type bit is ('0', '1');
  ```

- **character**

  ```
  type character is ('a', …);
  ```

- **integer**

  ```
  +1, -257, 123_456, 16#00FF#, ...
  ```

  - min/max_integer = +/- 2_147_483_647

- **real**

  ```
  +1.0, -1.0E+38, 38.0, ...
  ```

  - min_real = -1.0E+38, max_real = 1.0E+38

- **natural**
  - Untertyp von *integer:*

    ```
    subtype natural is integer range 0 to <highest_integer>
    ```

- **positive**
  - Untertyp von *integer:*

    ```
    subtype positive is integer range 1 to <highest_integer>
    ```

- **bit_vector**

  ```
  "100", X"A5"
  ```

- **string**

  ```
  "hold time error", "x"
  ```


Physikalische Typen

- **time**

```
type time is range <implementation specific>
  units
       fs;
       ps = 1000 fs;
       ns = 1000 ps;
       us = 1000 ns;
       ms = 1000 us;
       sec = 1000 ms;
       min = 60 sec;
       hr = 60 min;
  end units;
```

Anwendungsbeispiel:

```
constant clock_cycle: time := 3 ns;
...
a <= b + c after clock_cycle;
```

Arrays

- **Beispiel:**

```
type mem_type_1 is array (0 to 1023) of bit;
type mem_type_2 is array (1023 downto 0) of bit;
variable RAM: mem_type_1;
variable reversed_RAM: mem_type2;
...
reversed_RAM := RAM;
```

RAM(0) RAM(1023)

reversed_RAM(1023) reversed_RAM(0)

- **Vordefinierte Array-Typen**

```
type string is array (positive range <>) of character;
type bit_vector is array (natural range <>) of bit;
```

Beispiel:

```
variable RAM: bit_vector (0 to 1023);
```


Zuweisungen an Arrays

```
variable s: string (1 to 4);
variable one_hot: bit_vector (31 downto 0);
```

- **Auflisten der Elemente:**

```
s := ('f', 'o', 'o', 'd')
```

- **Mit expliziter Positionszuweisung:**

```
s := (1 => 'f', 3 => 'o', 4 => 'd', 2 => 'o')
```

- **Mit Default-Wert:**

```
one_hot := (17 => '1', others => '0')
```

- **Gemischt:**

```
s := ('f', 4 => 'd', others => 'o')
```

- **Zuweisung an Teil-Arrays:**

```
s(3 downto 2) := "er"  -- liefert "fred"
one_hot(17) := '1';
```


STD_LOGIC & STD_LOGIC_VECTOR

- **Der Typ *bit* mit den Werte '0' und '1' ist für sinnvolle Simulationen nicht ausreichend.**
- **Der Typ *std_logic* unterstützt mehrwertige Logik:**
  - 'U' - uninitialized
  - 'X' - forcing unkown
  - '0' - forcing '0'
  - '1' - forcing '1'
  - 'Z' - high impedance
  - 'W' - weak unknown
  - 'L' - weak '0'
  - 'H' - weak '1'
  - '-' - don't care

- **Keine eingebauten Typen: Das Package *STD_LOGIC_1164* muss importiert werden:**

```
library IEEE;
use IEEE.std_logic_1164.all;
```


Exkurs: Dreiwertige Simulation

- **Problem:**
  - z. B. unbekannter Startzustand bei Schaltwerken
- **Ansatz:**
  - Dreiwertige Logik: 0, 1, X = "unbekannt"

NOT

0	1
1	0
x	x

AND

	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

OR

	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

- **Achtung: 3-wertige Simulation ist pessimistisch!**
  - Wenn Simulation 'x' liefert, kann der reale Wert dennoch fest '0' oder '1' sein.

SIGNED & UNSIGNED (1)

- **Wie *STD_LOGIC*, aber Manipulationen wie bei Integer möglich**

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
…
variable counter: unsigned(3 downto 0) := "0000"
variable data:    std_logic;
…
data := counter(0);
counter := counter + "0001";
```


SIGNED & UNSIGNED (2)

- **Achtung: Zwischen *SIGNED/UNSIGNED* und *STD_LOGIC* ist explizite Konvertierung nötig:**

```
variable counter: unsigned (3 downto 0);
variable data: std_logic_vector (3 downto 0);

data := std_logic_vector (counter);
counter := unsigned (data);
```

Type-Casting (kein Funktionsaufruf):

- Möglich bei Arrays gleicher Größe mit Elementen vom gleichem Typ

Nützliche Konvertierungsfunktionen

- **Funktionen aus `IEEE.numeric_std.all`:**

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;

function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
```

- **Hinweis:**
  - Wenn *TO_INTEGER* ein Vektor mit unbekannten Werten ('X', 'U', ...) übergeben wird, wird ein gültiger (und damit falscher!) Integer-Wert zurück gegeben.
  - Der Vektor sollte deshalb vorher mit der folgenden Funktion aus `IEEE.std_logic_1164.all` überprüft werden:

```
function IS_X (ARG: STD_LOGIC_VECTOR) return boolean;
```


2.2.3. Ausdrücke und Operatoren

- **Operatoren, absteigend nach Präzedenz geordnet:**

```
**        abs       not
mod       rem
*         /
+(sign)   -(sign)
+         -         &
=         /=        <       <=      >       >=
and       or        nand    nor     xor
```

- **Anmerkungen**
  - Die logischen Operationen (and, or, ...) sind für die Typen *boolean* und *bit* sowie entsprechende Vektoren definiert.
  - Die relationalen Operatoren (=, /=, ...) liefern ein Ergebnis vom Typ *boolean*.
  - '&' verknüpft zwei Vektoren oder Einzelelemente zu einem neuen Vektor, z.B.:

```
x_vec := "10" & '0' & "10" -- liefert "10010", x_vec muss 5 Elemente haben
```

2.2.4. Sequentielle Anweisungen

- **Sequentielle Anweisungen stehen immer in einer *Prozess*-Umgebungen**
  - Das Gegenstück, die *parallelen Anweisungen*, werden später behandelt.
- **Eine VHDL-Architektur kann einen oder mehrere Prozesse enthalten.**
- **Alle Prozesse laufen untereinander parallel ab.**
- **Prozesse sind "Pseudo-Endlosschleifen". Wann sie neu starten, wird durch die *Sensitivitätsliste* definiert.**

Prozesse

- **Syntax**

```
[<label>:] process [(<sensitivity list>)]
     {<declarations>}
begin
     {<sequential statements>}
end process [<label>];
```

- **Beispiel**

Sensitivitätsliste: Prozess wird immer ausgeführt, wenn an a oder b ein Ereignis eintritt.

```
process (a, b)
begin
  if (a < b) then c <= '1';
  else c <= '0';
  end if;
end process;
```


Übersicht über sequentielle Anweisungen

a) Signal- und Variablenzuweisungen

b) Bedingungen: `if, case`

c) Schleifen: `loop`

d) Leere Anweisung: `null`

e) `assert`-Anweisung

f) `wait`-Anweisung

g) Prozeduren und Funktionen

a) Variablen- und Signalzuweisungen

- **Signalzuweisung**

```
y <= a and b;    -- y ist ein Signal (z.B. Ausgang der Entität)
```

- **Variablenzuweisung**

```
y := a and b;    -- y ist eine Variable (im Prozess deklariert)
```

- **Signale & Variablen sind *unterschiedliche* Objekte**
  - Details später

b) "if"- und "case"-Anweisung

- **if**

```
if <condition> then
   { <sequential statements> }
{ elsif <condition> then
   { <sequential statements> } }
[ else
   { <sequential statements> } ]
end if;
```

- **case**

```
case <expression> is
   when <choices> => <sequential statements>
   ...
   [ when others => <sequential statements> ]
end case;
```


Beispiele für *"case"*

```
process
  variable BCD: std_logic_vector (1 to 4);
begin
  ...
  case BCD is
    when "0000" => LED <= "0000000000";
    when "0001" => LED <= "0000000001";
    ...
    when "1001" => LED <= "1000000000";
    when others => LED <= "1111111111";
  end case;

  case BCD is
    when "0000" | "0001" | "0010" => LESS_THAN_THREE <= '1';
    when others => LESS_THAN_THREE <= '0';
  end case;
  ...
end process;
```

- **"others" muss die letzte Wahlmöglichkeit sein**

c) Schleifen

- **Syntax**

```
[<label>:] [<iteration_scheme>] loop
  { <sequential statements> }
  { next [<label>] [when <condition>]; }
  { exit [<label>] [when <condition>]; }
end loop [<label>];
```

- **Iterations-Schemata**
  - **for**
  - **while**
  - keins (dann wiederholte Ausführung bis Abbruch durch "**exit**")

Beispiele

- ***"for"*-Schleife:**

```
L: for i in 1 to 10 loop -- i von 1 bis 10
     ...                 -- sequentielle Anweisungen
   end loop;
```

- ***"while"*-Schleife mit "*next*" und "*exit*"**

```
L1: while i<10 loop
  L2: while j<20 loop
        ...
        next L2 when i=j;  -- Sprung zu L2
        ...
        exit L1 when i>4;  -- Sprung zu L3
        ...
      end loop L2;
    end loop L1;
L3: ...
```

d) Die "*null*"-Anweisung

- **... tut nichts.**

- **Anwendung in "case"-Anweisungen:**
  - Die "null"-Anweisung macht explizit, dass in bestimmten Fällen nichts passieren soll (und nicht etwa der Fall vergessen wurde).
  - Manche Tools erwarten vollständige "*case*"-Anweisungen.

```
case controller_command is
   when forward => engage_motor_forward;
   when reverse => engage_motor_reverse;
   when idle => null;
end case;
```


e) "*assert*"-Anweisung

- **Syntax**

```
assert <condition>
    [report <string>]
    [severity {note|warning|error|failure}];
```

- **Beispiele**

```
assert spec_out = impl_out
  report "Specification and implementation differ!";
  -- default severity is "error"

assert false report "Just some information" severity note;
  -- gibt eine Meldung während der Simulation aus
```


f) "*wait*"-Anweisung

- **"*wait...*" simuliert das Warten um eine bestimmte Zeit oder auf ein Ereignis.**

- **Varianten:**

```
wait on A, B;          -- warten bis sich Signal A oder B ändert
wait until (X < 10);   -- warten bis boole'scher Ausdruck erfüllt
wait for 7 ns;         -- warten bis Zeit verstrichen ist
wait;                  -- warten für immer (zum Beenden der Simulation)
```

- **Eine Sensitivitätsliste lässt sich immer durch äquivalente "wait"-Anweisung ersetzen:**

```
process (a, b, c)
begin
  -- Haupt-Code
end process;
```

```
process
begin
  -- Haupt-Code
  wait on a, b, c;
end process;
```


Behandlung im Simulator

- **Eine "*wait*"-Anweisung bewirkt i.A., dass der Simulator zu einem anderen Prozess wechselt.**
  - Die Verwaltung der Prozesse (zur Auswahl des nächsten) geschieht intern über Ereignis-Listen.
  - Zur Simulation der Zeit besitzt der Simulator einen Zeitzähler, und die Prozesse, die in einer "*wait*"-Anweisung warten, werden in einer Prioritätswarteschlange verwaltet.

- **Beispiel:**

```
entity CLK_GEN is
  port (clk_out1, clk_out2: out std_logic);
end CLK_GEN
```

Beispiel (Forts.)

```
architecture BEHAVIOR of CLK_GEN is
begin

  P1: process
    variable clk1: std_logic := '0';
  begin
    clk_out1 <= clk1;
    clk1 := not clk1;
    wait for 5 ns;
  end process;

  P2: process
    variable clk2: std_logic := '0';
  begin
    clk_out2 <= clk2;
    clk2 := not clk2;
    wait for 3 ns;
  end process;

end BEHAVIOR;
```


g) Prozeduren und Funktionen

- **Prozeduren und Funktionen können innerhalb von Prozessen deklariert werden**
  - im Deklarationsteil (vor "begin")
- **Sinnvolle Anwendung: Testbenches!**
- **Prozeduren ohne Parameter**

```
procedure run_cycle is
  variable period: time := 10ns;
begin
  clk <= '0';  -- 'clk' ist ein global deklariertes Signal
  wait for period / 2;
  clk <= '1';
  wait for period / 2;
end procedure;
```


- **Prozedur mit Parametern**

```
procedure run_multiplication (a, b: in integer) is
...

procedure get_char1 (val: in std_logic; ret: out character) is
...

procedure inrement (val: inout integer) is
...
```

  - Die Modi 'out' und 'inout' erlauben auch die Rückgabe von Werten. ('inout'-Parameter sind vergleichbar mit Referenzen in C++)
  - Parameter vom Modus 'out' oder 'inout' werden als Variablen-Objekt behandelt, Parameter vom Modus 'in' als Konstante.

- **Funktionen**

```
function get_char2 (val: in std_logic) return character is
begin
  ...
  return 'q';    -- Rückgabe
end;
```

- **Aufruf von Prozeduren & Funktionen**

```
...
run_cycle;      -- parameterlose Prozedur
run_multiplication (13, x);
c := get_char2 ('X');
...
```

2.3. Besondere Konzepte von VHDL

2.3.1. Strukturbeschreibungen

- **Beispiel für eine Struktur-Beschreibung:**

```
architecture STRUCTURE of HALF_ADDER is

  component XOR2
    port (x, y: in std_logic; z: out std_logic);
  end component;

  component AND2
    port (x, y: in std_logic; z: out std_logic);
  end component;

  for U0: XOR2 use entity WORK.XOR2(DATAFLOW);
  for U1: AND2 use entity WORK.AND2(DATAFLOW);

begin
  U0: XOR2 port map(x => a, y => b, z => sum);
  U1: AND2 port map(x => a, y => b, z => carry);
end STRUCTURE;
```

Deklaration verwendeter Entitäten (vergleichbar mit Funktions-Headern in C)

Konfiguration: Gibt an, welche Entität/Architektur jeweils verwendet wird

Instantiierung

Port-Bindung (Form: <Port> => <lokal definiertes Signal>)

2.3.2. Parallele Anweisungen

```
architecture TEST of EXAMPLE is
  -- Signal-Deklarationen etc.
begin

  process
  begin
    -- sequentielle Anweisungen
  end process;
  ...

  x <= y;   -- Signalzuweisung (parallel)
  ...

  U0: SOME_MODULE port map ( ... )

end TEST;
```

parallele Anweisungen

Sequentielle und Parallele Anweisungen

- **Sequentielle Anweisungen** **("Sequential statements")**
  - werden logisch gemäß der Reihenfolge im Programm abgearbeitet
  - stehen in "process"-Umgebung
  - Während der Abarbeitung vergeht (logisch) keine Zeit

- **Parallele Anweisungen** **("Concurrent statements")**
  - werden logisch parallel abgearbeitet (unabhängig von der Reihenfolge im Programm)
  - werden durch Signalereignisse getriggert
  - Beispiele
    - Signalzuweisungen außerhalb von "process"-Umgebungen
    - "process" als Ganzes
    - Modul-Instanzierungen gelten auch als parallele Anweisung

Parallele Signalzuweisungen

- **Parallelen Signalzuweisungen...**

```
and_out <= i1 and i2;
or_out  <= i2 or i3;
```

- **... lassen stets sich durch äquivalente Prozesse darstellen:**

```
process (i1, i2)
begin
  and_out <= i1 and i2;
end process;

process (i2, i3)
begin
  or_out <= i2 and i3;
end process;
```

2.3.3. Zeitmodellierung

Bereits bekannt:

- ***wait*-Anweisung**

```
wait on A, B;          -- warten bis sich Signal A oder B ändert
wait until (X < 10);   -- warten bis boole'scher Ausdruck erfüllt
wait for 7 ns;         -- warten bis Zeit verstrichen ist
wait;                  -- warten für immer (zum Beenden der Simulation)
```

- **Sensitivitätsliste**

```
process (a, b, c)
begin
  -- Haupt-Code
end process;
```

```
process
begin
  -- Haupt-Code
  wait on a, b, c;
end process;
```

Hinweis: *Ein Prozess muss immer mindestens eine "wait"-Anweisung oder eine Sensitivitätsliste enthalten!*

Weitere Möglichkeit:

- ***after*-Klausel bei *Signal*zuweisungen**

```
architecture TIMED_DATAFLOW of HALF_ADDER is
begin

  process (a, b)
  begin
    sum <= a xor b after 2 ns;
    carry <= a and b after 1 ns;
  end process;

end TIMED_DATAFLOW;
```

-> Unterschied zwischen *Variablen* und *Signalen* beachten!
(*after*-Klausel ist nur mit Signalen möglich)

Signalzuweisung

- **Signalzuweisung erzeugt Simulations-*Ereignis***
  - Zuweisung verzögert bis Simulationszyklus durch *"wait"*-Anweisung angestoßen wird
- **Scheduling**
  - gemäß angegebenen Verzögerungszeiten
  - Aktionen zur gleichen Zeit gemäß Reihenfolge im Programm

Achtung: Eine erneute Zuweisung an ein Signal kann vorherige aufheben.

Beipiel

```
process
begin
  S1 <= '1' after 30 ns;       -- wirkungslos
  S1 <= '0' after 20 ns;       -- Ereignis in 20 ns
  S2 <= '1' after 10 ns, '0' after 30 ns;  -- zwei Ereignisse
  wait on CLK;                 -- triggert Simulationszyklus
end process;
```


Verzögerungsmodelle (*after*-Klausel)

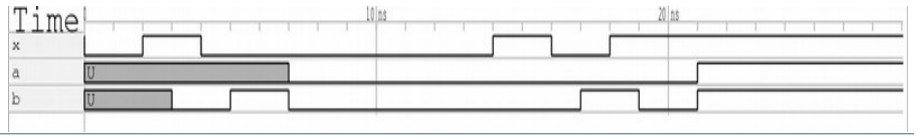
a) Trägheits-Modell ("*inertial*", Default)

- Bei schneller Änderung des Ausgangssignales innerhalb der Verzögerungszeit wird ältere Zuweisung ignoriert

b) Transport-Modell ("*transport*")

- Ausgangssignal wird exakt weitergegeben

```
signal x, a, b: std_logic;

...
a <= x after 3 ns;      -- or 'a <= inertial x after 3 ns;'
b <= transport x after 3 ns;
...
```



Vergleich: Variablen und Signale

- **Variablen ...**
  - ... speichern Werte,
    verhalten sich also genau wie Variablen in Programmiersprachen.
  - ... müssen ***innerhalb einer Prozess-Umgebung*** deklariert werden,
  - ... sind also nur innerhalb des Prozesses sichtbar.
- **Signale ...**
  - ... speichern Werte ***und Informationen zum zeitlichen Verlauf.***
  - ... können auch auf Prozess- oder Architektur-Ebene deklariert werden.
  - Ein-/Ausgänge einer Entität sind Signale.
- **Übliche Verwendung:**
  - **Signale:** reale Registerinhalte und Signalleitungen
  - **Variablen:** Zwischenergebnisse einer Berechnung u.ä.

Beispiel

- **Signalzuweisung**

```
process
  signal a, b: std_logic;
begin
  a <= '1';     -- erzeugt Ereignis "a='1' in 0ns"
  b <= a;       -- erzeugt Ereignis "b='U' in 0ns"
  wait;         -- Simulator bearbeitet Ereignisliste
end;               => a = '1', b = 'U'
```

- **Variablenzuweisung**

```
process
  variable a, b: std_logic;
begin
  a := '1';     -- setzt a = '1' (Zuweisung sofort)
  b := a;       -- setzt b = '1' (Zuweisung sofort)
  wait;
end;               => a = '1', b = '1'
```

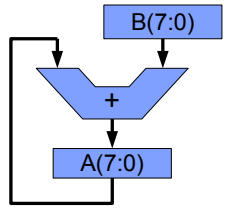

2.4. Synthese

Was ist Synthese?

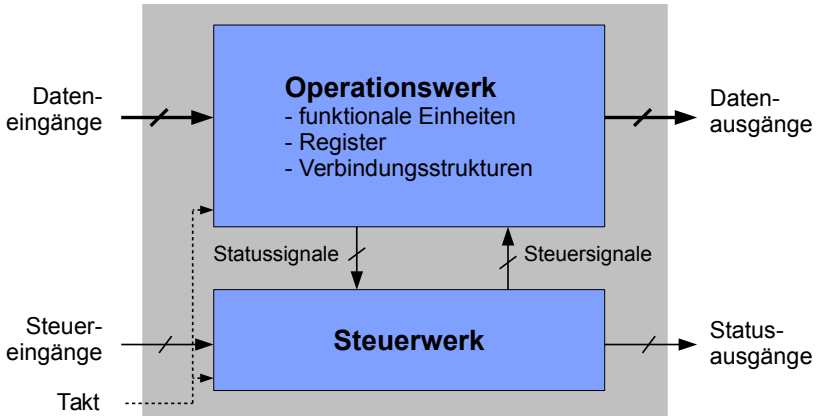
- **Eingabe**
  - Beschreibung auf Register-Transfer-Ebene,
    z.B. VHDL im Verhaltens-/Datenfluss-Stil
- **Ausgabe**
  - Gatternetzliste, z.B. VHDL im Struktur-Stil
- **"Synthetisierbares VHDL":**
  - Teilmenge an Sprachkonstrukten, die automatisch in eine Struktur umgesetzt werden können
  - Unterstützte Teilmenge hängt z. T. von Synthese-Tool ab
  - Üblicher Architektur-Name: "RTL"

2.4.1. Entwurf auf Register-Transfer-Ebene

- **Eine Schaltung auf Gatter-Ebene besteht aus:**
  - **Komponenten:** primitive Gatter (AND, OR, ...), Flipflops
  - **Verbindungen:** einzelne Leitungen
- **Ein Entwurf auf Register-(Transfer-)Ebene enthält:**
  - **Algorithmus**
    - Folge von *Register-Transfer*-Anweisungen, z.B.: `A <- A + B`
  - **Komponenten**
    - funktionale Einheiten (z. B. Addierer) mit definierter **Wortbreite**
    - Register
    - Steuerwerk
  - **Verbindungen:** Busse
    (mit definierter *Wortbreite*)

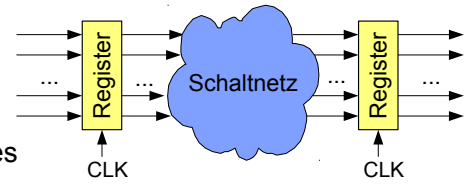


Zielstruktur



Regeln für synchrone Schaltwerke

- **Alle sequentiellen Elemente sind flankengetriggerte D-Flipflops.**
  - Keine pegel-getriggerten Latches



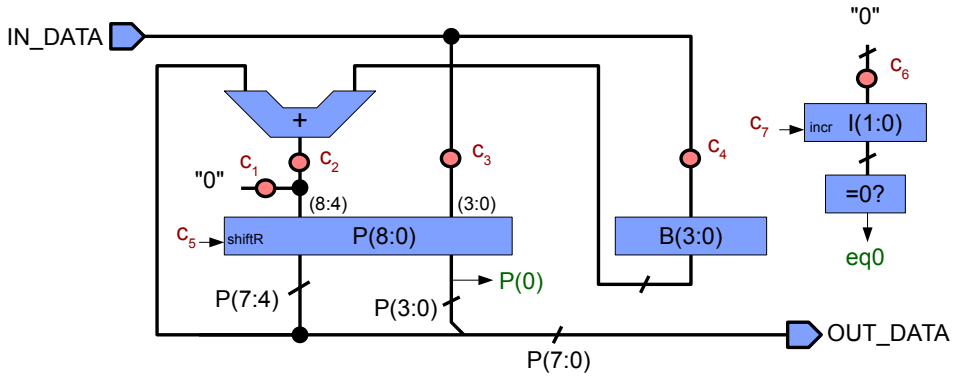
- **Alle Flipflops werden mit dem gleichen Takt angesteuert und wechseln gleichzeitig Ihren Zustand.**
  - Kein asynchroner Reset
- **Es gibt keine Verbindungen zwischen der Taktleitung und den logischen Signalen.**
  - Keine "gated clocks"

Beispiel: Einfacher Multiplizierer

a) Algorithmus

```
        declare bus    IN_DATA (3:0), GO;
        declare bus   OUT_DATA (7:0), OUT_VALID;
        declare register P(8:0), B(3:0), I(1:0);

RESET:  OUT_VALID <- 0, P(8:4) <- 0, I <- 0, P(3:0) <- IN_DATA;
WAITB:  B <- IN_DATA, if GO = 0 then goto WAITB fi;

LOOP:   if P(0) = 0 then goto CONT fi;
        P(8:4) <- P(8:4) + B;
CONT:   P(7:0) <- P(8:1), P(8) <- 0,
        I <- I + 1;
        if I <> 0 then goto LOOP fi;

DONE:   OUT_DATA <- P(7:0), OUT_VALID <- 1;
```

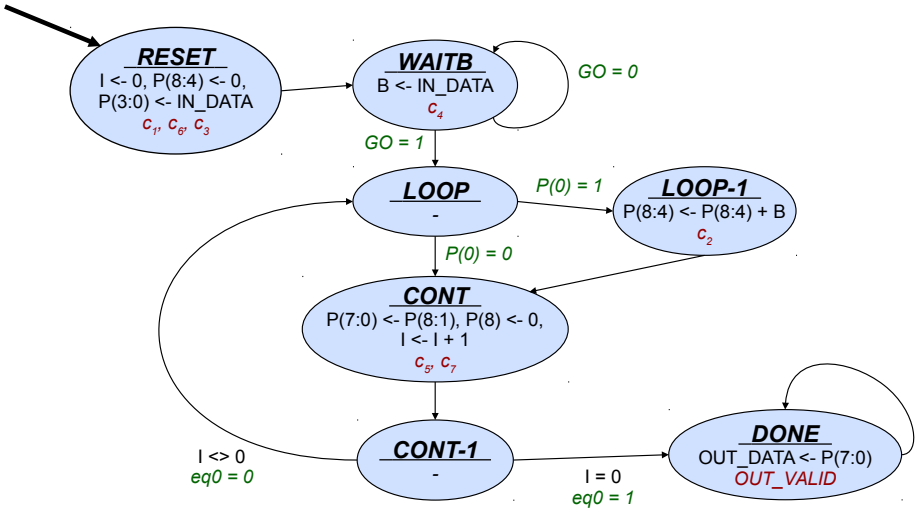

b) Strukturdiagramm für Operationswerk



c) Zustände und Steuersignale

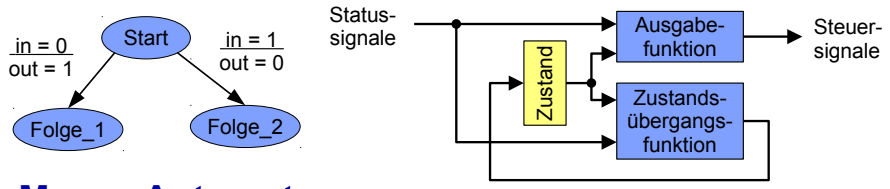
Zustand	Steuersignale	
		`declare in bus  IN_DATA (3:0), GO;`
		`declare out bus  OUT_DATA (7:0), OUT_VALID;`
		`declare register P(8:0), B(3:0), I(1:0);`
RESET	c_1, c_6, c_3	`RESET: OUT_VALID <- 0, I <- 0,`
		`P(8:4) <- 0, P(3:0) <- IN_DATA;`
WAITB	c_4	`WAITB: B <- IN_DATA, if GO = 0 then goto WAITB fi;`
LOOP		`LOOP: if P(0) = 0 then goto CONT fi;`
LOOP-1	c_2	`P(8:4) <- P(8:4) + B;`
CONT	c_5	`CONT: P(7:0) <- P(8:1), P(8) <- 0,`
	c_7	`I <- I + 1;`
CONT-1		`if I <> 0 then goto LOOP fi;`
DONE		`DONE: OUT_DATA <- P(7:0), OUT_VALID <- 1;`

d) Zustandsübergangsdiagramm

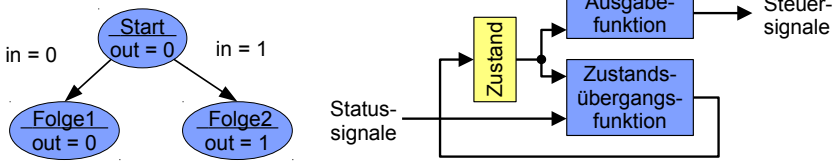


Mealy- und Moore-Automaten

- **Mealy-Automat**
  - Ausgabe hängt von Zustand *und* Eingabe ab



- **Moore-Automat**
  - Ausgabe hängt *nur* von Zustand ab



2.4.2. Synthese von Schaltnetzen

- **Möglicher Beschreibungsstil**
  - Datenfluss (nur parallele Anweisungen)
  - Verhalten (sequentielle Anweisungen), **wenn damit rein kombinatorisches Verhalten beschrieben wird** (alle Eingänge in Prozess-Sensitivitätsliste)
- **Übersetzung erfolgt nach einfachen Regeln:**
  - **Operatoren "*and*", "*or*", ...**
    -> einfache Gatter, je nach Datentyp mit passender Wortbreite
  - **Signal-, Variablenzuweisung (parallele oder sequentielle Anweisung)**
    -> Verbindungsleitungen
  - **Komplexere Operatoren (+, –, *, ...)**
    -> Module aus Bibliothek oder Modulgeneratoren (z. B. für CLAs)
  - **Steueroperationen (*if, case*)**
    -> Multiplexer
  - **Schleifen mit festen Grenzen**
    -> Aufrollen der Schleife

Beispiel: Halbaddierer

```
entity HALF_ADDER is
  port (a, b: in STD_LOGIC; sum, carry: out STD_LOGIC);
end HALF_ADDER;

architecture RTL of HALF_ADDER is
begin
  sum <= a xor b;
  carry <= a and b;
end RTL;
```


Beispiel: 8-Bit-Addierer

```
entity ADDER2 is
  port (  a: in std_logic_vector (7 downto 0);
          b: in std_logic_vector (7 downto 0);
        sum: out std_logic_vector (7 downto 0)
      );
end ADDER2;


architecture RTL_1 of ADDER2 is
begin
  sum <= a + b;
end RTL_1;


architecture RTL_2 of ADDER2 is
begin
  process (a, b)   -- alle Eingänge in Sensitivitätsliste
  begin
    sum <= a + b;
  end process;
end RTL_1;
```


Beispiel: Inkrementierer

```
entity INCR4 is
  port (a: in std_logic_vector(3 downto 0);
        en: in std_logic;
        y: out std_logic_vector(3 downto 0));
end INCR4;


architecture RTL of INCR4 is
begin
  process (a, en) -- wichtig: alle Eingänge in Sensitivitätsliste
  begin
    if en = '1' then
      y <= a + "0001";
    else
      y <= a;
    end if;
  end process;
end RTL;
```


Beispiel: einfache ALU

```
entity ALU32 is
  port (a, b: in std_logic_vector(31 downto 0);
        sel: in std_logic(1 downto 0);
        y: out std_logic_vector(31 downto 0));
end ALU32;

architecture RTL of ALU32 is
begin
  process (a, b, sel)  -- wichtig: alle Eingänge in Sensitivitätsliste
  begin
    case sel is
      when "00" =>  y <= a + b;
      when "01" =>  y <= a AND b;
      when "10" =>  y <= a OR b;
      when "11" =>  y <= NOT b;
    end case;
  end process;
end RTL;
```

Beispiel: Paritätsgenerator

```
entity parity_gen is
  port (a: in std_logic_vector (0 to 7);
        even, odd: out std_logic);
end parity_gen;

architecture rtl of parity_gen is
begin

  process (a)
    variable tmp: std_logic;
    variable n: integer;
  begin
    tmp := '0';
    for n in a'range loop
      tmp := tmp xor a(n);
    end loop;
    even <= tmp;
    odd <= not tmp;
  end process;

end rtl;
```


Hinweise

- Für reale Register oder Leitungen Signale verwenden (keine Variablen).
- Für reale Register oder Signale stets den Typ *std_logic* oder *std_logic_vector* verwenden (nicht *integer*).
- Vergessene Signale in der Sensitivitätsliste führen zu schwer zu findenden Fehlern. Parallele Signalzuweisungen sind in der Hinsicht sicherer.

Optimierungsmöglichkeiten in VHDL

- **Beispiel: Transponier-Schaltnetz für MIDI-Noten**
  - Bedingte Addition von 8-Bit-Konstanten -24, -12, +12, +24 oder 0
    - (entspricht +/- 1 oder 2 Oktaven)
  - Auswahl durch Drehschalter
    (setzt genau eines der Signale 'dn2', 'dn1', 'up1', 'up2' auf 1)

- **Modul-Schnittstelle**

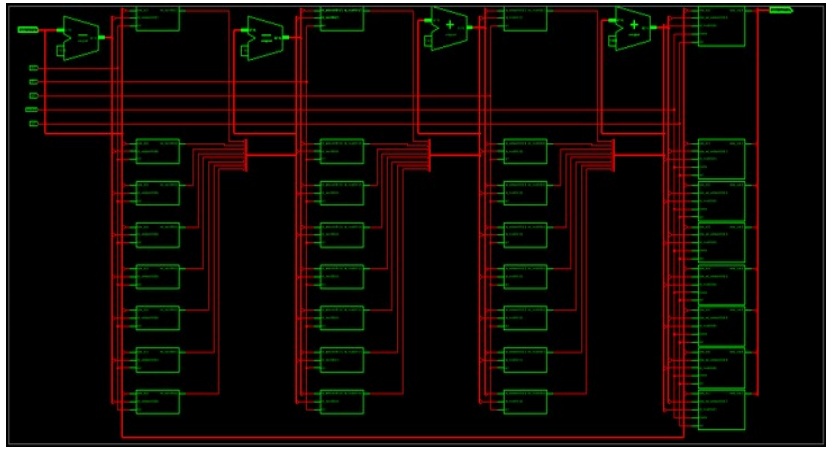
```
entity TransAdd is
  port (
    enable, dn2, dn1, up1, up2: in std_logic;
    data_in: in std_logic_vector (7 downto 0);
    data_out: out std_logic_vector (7 downto 0)
  );
end TransAdd;
```


Erster Ansatz

```
architecture RTL of TransAdd is
begin

  process (enable, dn2, dn1, up1, up2, data_in)
    variable ret: std_logic_vector (7 downto 0);
  begin
    ret := data_in;
    if enable = '1' then
      if dn2 = '1' then ret := ret - 24; end if;
      if dn1 = '1' then ret := ret - 12; end if;
      if up1 = '1' then ret := ret + 12; end if;
      if up2 = '1' then ret := ret + 24; end if;
    end if;
    data_out <= ret;
  end process;

end RTL;
```

Synthese-Ergebnis

- 2 Addierer + 2 Subtrahierer ; 19 Slices

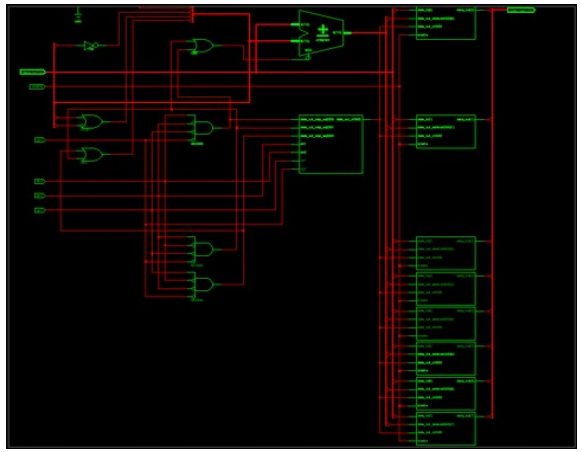


Verbesserter Entwurf

```
architecture RTL of TransAdd is
begin

  process (enable, dn2, dn1, up1, up2, data_in)
    variable ret: std_logic_vector (7 downto 0);
    variable sel: std_logic_vector (0 to 3);
  begin
    ret := data_in;
    if enable = '1' then
      sel := (dn2, dn1, up1, up2);
      case sel is
        when "1000" => ret := data_in - 24;
        when "0100" => ret := data_in - 12;
        when "0010" => ret := data_in + 12;
        when "0001" => ret := data_in + 24;
        when "0000" => ret := data_in;
        when others => ret := (others => '-');  -- Don't Care
      end case;
    end if;
    data_out <= ret;
  end process;

end RTL;
```


Synthese-Ergebnis

- ***ein*** Addierer/Subtrahierer ; ***10*** Slices



2.4.3. Synthese von Schaltwerken

- **Latches (pegelgesteuert)**
  - werden automatisch erzeugt, wenn ein Signal nicht mit jeder Änderung der Eingangssignale seinen Wert ändert
  - ***sind oft nicht beabsichtigt (vgl. synchroner Entwurfstil)***

- **Flipflops (flankengetriggert)**
  - werden automatisch erzeugt, wenn eine sequentielle Signalzuweisung von einer Bedingung der Form

    ```
    rising_edge(clk)
    ```

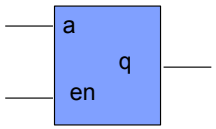
    oder

    ```
    clk'event and clk = '1'
    ```

    abhängt.

Sequentieller VHDL-Code, der sich nicht in Latches oder Flipflops übersetzen lässt, ist i.d.R. nicht synthetisierbar.

Beispiel: Synthetisierbares Latch




```
entity LATCH is
  port (en, a : in std_logic; q: out std_logic );
end LATCH;

architecture RTL of LATCH is
begin
  process (en, a)
  begin
    if (en='1') then q <= a; end if;
  end process;
end RTL;
```


Unerwünschte Latches

- **Vergessene Zuweisungen**

```
process (a)
begin
  if a = '1' then
    q <= '0';
    p <= '1';
  else
    q <= '1';
    -- keine Zuweisung an p => für p wird Latch erzeugt
  end if;
end process;
```

Tipp: Am Anfang des Prozesses Defaults für alle Ausgänge zuweisen

- **Vergessene Eingänge in Sensitivitätsliste**

```
process (a)
begin
  q <= a and b;
  -- Änderung an b alleine verändert q nicht
end process;
```


Beispiel: Synthetisierbares D-Flipflop

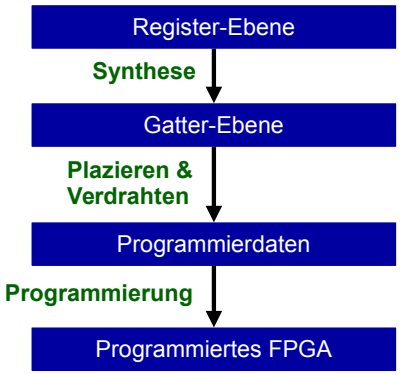
```
entity DFF is
  port (clk, a : in std_logic; q: out std_logic );
end DFF;

architecture RTL_1 of DFF is  -- Variante 1
begin
  process (clk)
  begin
    if rising_edge(clk) then q <= a; end if;
  end process;
end RTL_1;

architecture RTL_2 of DFF is  -- Variante 2
begin
  process
  begin
    wait until rising_edge(clk);
    q <= a;
  end process;
end RTL_2;
```


Beispiel: Synthetisierbarer Zähler

```
entity COUNTER is
  port ( clk, down, up: in std_logic;
         cnt_out: out std_logic_vector (7 downto 0) );
end COUNTER;


architecture RTL of COUNTER is
  signal state: unsigned (7 downto 0);
begin

  -- Ausgabeschaltnetz (hier einfach als parallele Anweisung)...
  cnt_out <= std_logic_vector (state);

  -- Übergangsprozess, modelliert Register 'state' und Übergangsfunktion ...
  process (clk)
  begin
    if rising_edge(clk) then
      if (down = '1' and up = '1') then state <= "00000000"; end if;
      if (down = '1' and up = '0') then state <= state - 1; end if;
      if (down = '0' and up = '1') then state <= state + 1; end if;
    end if;
  end process;

end RTL;
```

2.4.4. Werkzeuge im Praktikum

Entwurfsablauf bei Xilinx (ISE 14.6)



	Xilinx	Praktikum
Synthese	`xst`	`ees-xst`

Plazieren & Verdrahten	`ngdbuild`
	`map`
	`par`
	`bitgen`
		`ees-implement /`
		`ees-implement_viscy`

Programmierung	`impact`	`ees-impact`

Xilinx-Dokumentation: $XILINX/doc/usenglish/books/manuals.pdf

Tool: *ees-xst*

- **Funktion**
  - Synthese eines VHDL-Entwurfes (komplette Hierarchie)
- **Syntax**

  `ees-xst [-p] <top-level>.vhd [ <modul2>.vhd ... ]`

  -p : erzeuge Pad-Zellen für die Ports des Top-Level-Moduls
- **Ausgabe**

  <top-level>.ngc : Synthese-Ergebnis (technologienahe Strukturbeschreibung)
  <top-level>.ngr : RTL-Struktur
  <top-level>.prj : Projektdatei für XST
  <top-level>.ifn : Skript für XST
  <top-level>.log : Log-Datei (Timing-Report, Flächenverbrauch, Fehlermeldungen, ...)
- **Anmerkungen**
  - Die erzeugten Strukturbeschreibungen (*.ngr und *.ngc) können in der Entwicklungsumgebung *ise* angezeigt werden (einfach öffnen, kein Projekt anlegen!).
  - Zum Feinabstimmung der Parameter kann jetzt XST direkt aufgerufen werden; XST benötigt die Projekt- und Skript-Datei.

Tool: *ees-implement*

- **Funktion**
  - Plazierung und Verdrahtung inkl. Erzeugen eines Bit-Files
- **Syntax**

  `ees-implement <design>.ngc <constraints>.ucf`
- **Ausgabe**

  <design>.bit :
  Programmierdaten für das im Praktikum verwendete FPGA
- **Anmerkungen**
  - Die .ucf-Datei definiert u.a., welcher Port welchem FPGA-Pin zugewiesen wird (siehe Beispiel)
  - Die .ngc-Datei muss zuvor mit '*eis-xst **-p***' erzeugt worden sein.

Beispiel: Auf-/Abwärts-Zähler

1. **Zähler in VHDL beschreiben**
2. **Testbench schreiben & simulieren**
3. **Zähler synthetisieren**
   - Ergebnis (Log-Datei und Strukturen) anschauen
4. **UCF-Datei erzeugen**
5. **Plazieren & Verdrahten ("Implementieren")**
6. **FPGA programmieren**
7. **Fertig!**

2.5. Zusammenfassung

- **Hardwarebeschreibungssprachen**
  - *Wozu dienen sie? Welche Eigenschaften sind gewünscht?*
- **Teilmenge der Sprache VHDL**
  (nicht vollständig, aber für einfache bis mittlere Projekte ausreichend)
  - Entities & Architekturen
  - Variablen & Signale
  - Parallele & sequentielle Anweisungen
  - Beispiele zur Modellierung von typischen Schaltungen
- **Simulation mit Testbench**
- **Synthese mit Xilinx XST**

Lernziel: **Modellierung, Simulation und Synthese von Entwürfen auf Gatter- und Register-Ebene**