Prozessorarchitektur

Kapitel 2: Einführung in VHDL

M. Schölzel

Inhalt

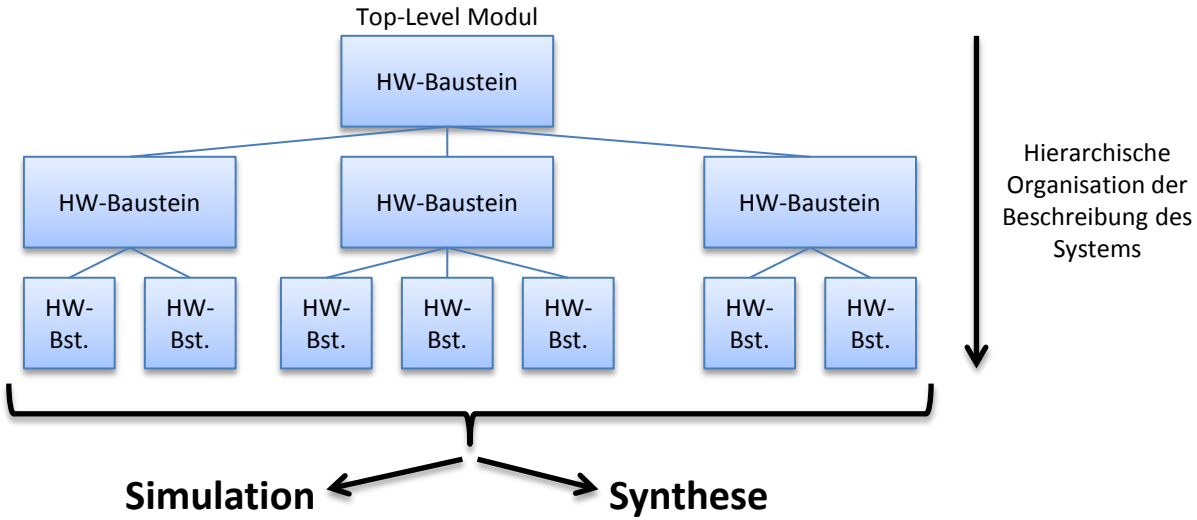
- Beschreibung von Bausteinen in VHDL
- Simulationssemantik
- Synthesefähige Beschreibungen
  - Kombinatorische Logik
  - Sequentielle Logik
- Zusammenfassung

Inhalt

- Beschreibung von Bausteinen in VHDL
- Simulationssemantik
- Synthesefähige Beschreibungen
  - Kombinatorische Logik
  - Sequentielle Logik
- Zusammenfassung

Überblick

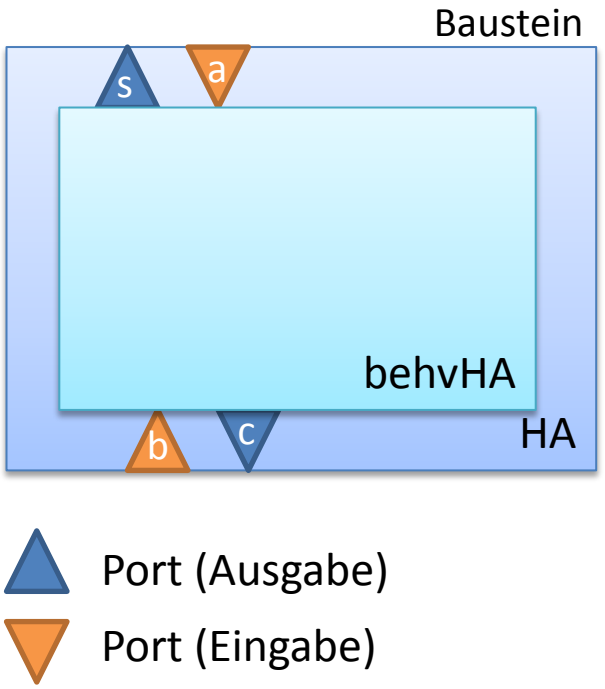
- VHDL = VHSIC Hardware Description Language
- VHSIC = Very-High Speed Integrated Circuits
- Entwickelt Anfang der 1980er Jahre im Auftrag des Verteidigungsministeriums der USA
- Ziel:
  - Technologieunabhängige Beschreibung von Hardwarebausteinen
  - Beschreibung auf unterschiedlichen Abstraktionsebenen (Logikeben, RT-Ebene,…)
  - Beschreibung aus unterschiedlichen Sichten (Verhaltenssicht, Struktursicht)
  - Validierung in einem Simulator
- Aufbau und Verwendung:



Hardware-Baustein

- Ein Baustein stellt eine Komponente eines Systems dar
- Kapselung der Funktionen der Komponente
- Zugriff über Schnittstellen

```
entity HA is
port (
    a : in std_logic;
    b : in std_logic;
    s : out std_logic;
    c : out std_logic
);
end HA;
```



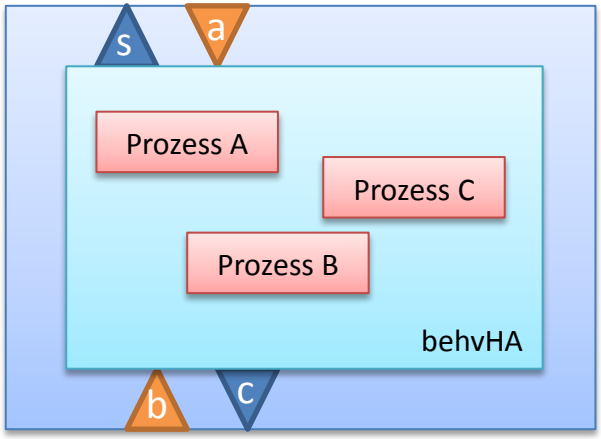
- Beschreibung des Verhaltens des Bausteins:

```
architecture behvHA of HA is
begin
    .
    .
    .
end behvHA;
```

Verhalten eines Bausteins beschreiben

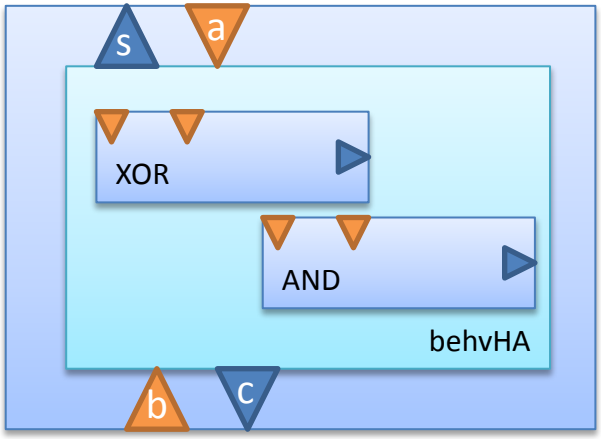
Verhaltensorientiert

- Funktionale Beschreibung des Verhaltens mittels nebenläufiger Prozesse



Strukturorientiert

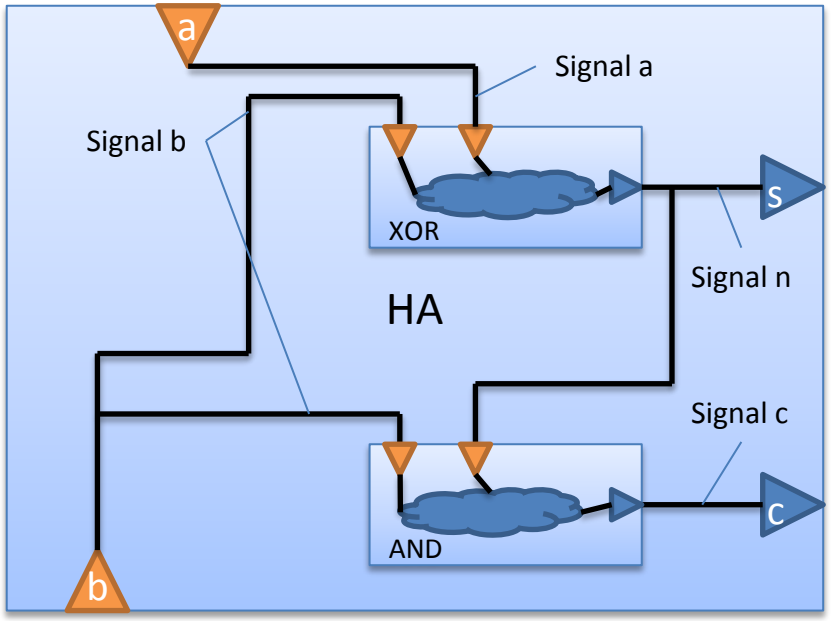
- Aufbau eines Bausteins aus Komponenten (Sub-Bausteinen)
- Verdrahtung der Komponenten



Verdrahtung (1)

- Verdrahtung von Bausteinen und Komponenten durch Signale:
- Ein-/Ausgabeport eines Bausteins repräsentiert ein Signal in diesem Baustein
- Deklaration weiterer Signale möglich zur Verdrahtung von Komponenten:

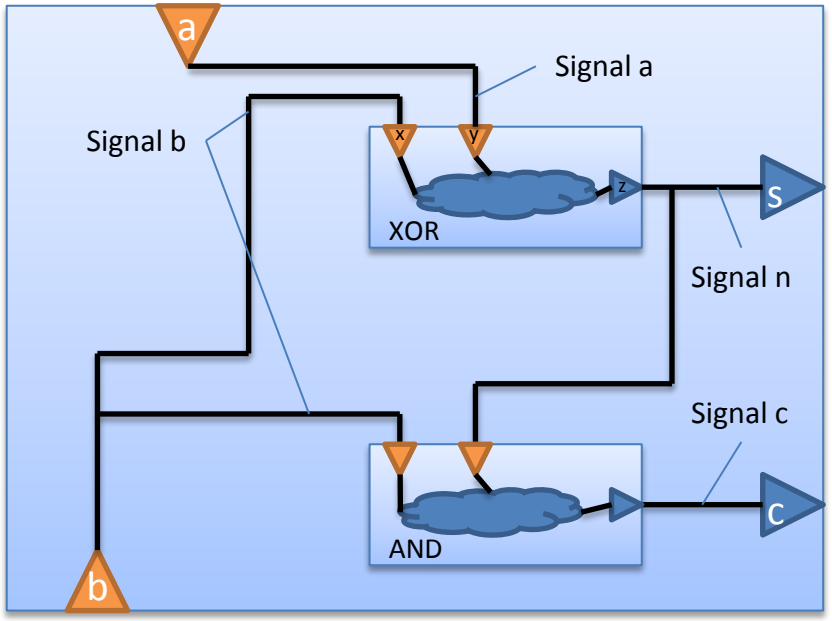
```
architecture behvHA of HA is
  signal n: std_logic;
begin
    .
    .
    .
end behvHA;
```



Verdrahtung (2)

- Verwendete Komponenten müssen aufgeführt werden:

```
architecture behvHA of HA is
  component XOR is
    port (
      x : in std_logic;
      y : in std_logic;
      z : out std_logic);
  end component;

  ...

  signal n: std_logic;
begin
   .
   .
   .
end behvHA;
```



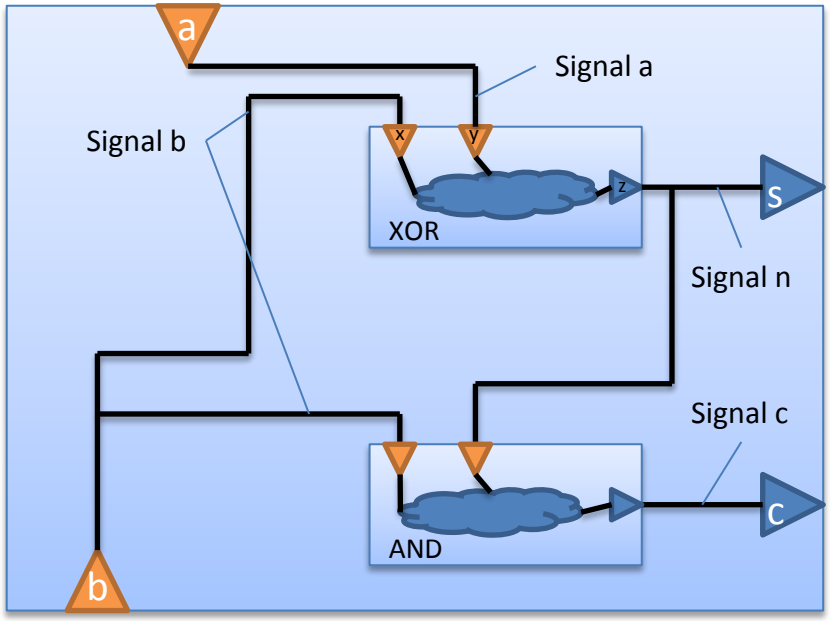
Verdrahtung (3)

- Verwendete Komponenten müssen verdrahtet werden:

```
architecture behvHA of HA is
  component XOR is
    port (
      x : in std_logic;
      y : in std_logic;
      z : out std_logic);
  end component;

  ...

  signal n: std_logic;
begin
   XOR port map (
     x => b,
     y => a,
     z => n
   );

   .
   .
   .
end behvHA;
```

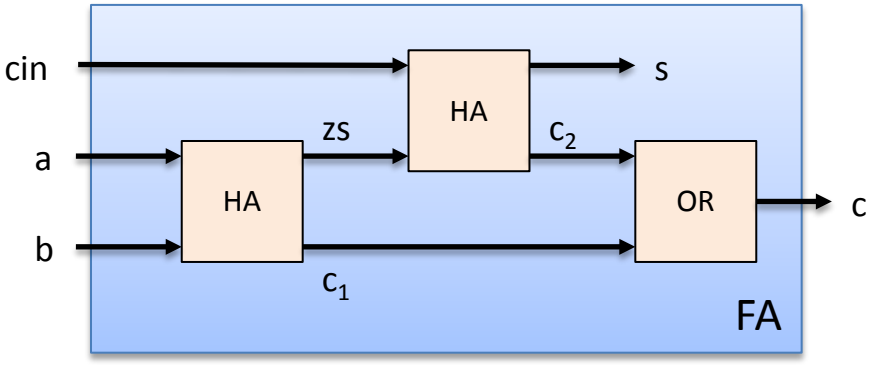
Portnamen der verwendeten Komponente

Signal- oder Portname aus dem umgebenden Baustein



Beispiel Volladdierer

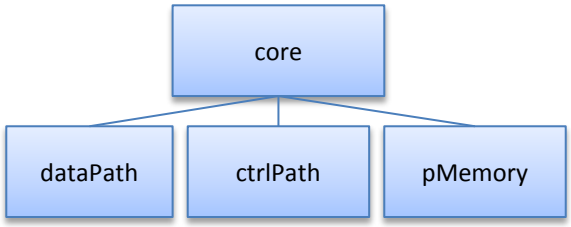
```
library ieee;
use ieee.std_logic_1164.all;

entity FA is
port (
  a  : in std_logic;
  b  : in std_logic;
  ci : in std_logic;
  s  : out std_logic;
  co : out std_logic
);
end FA;
```

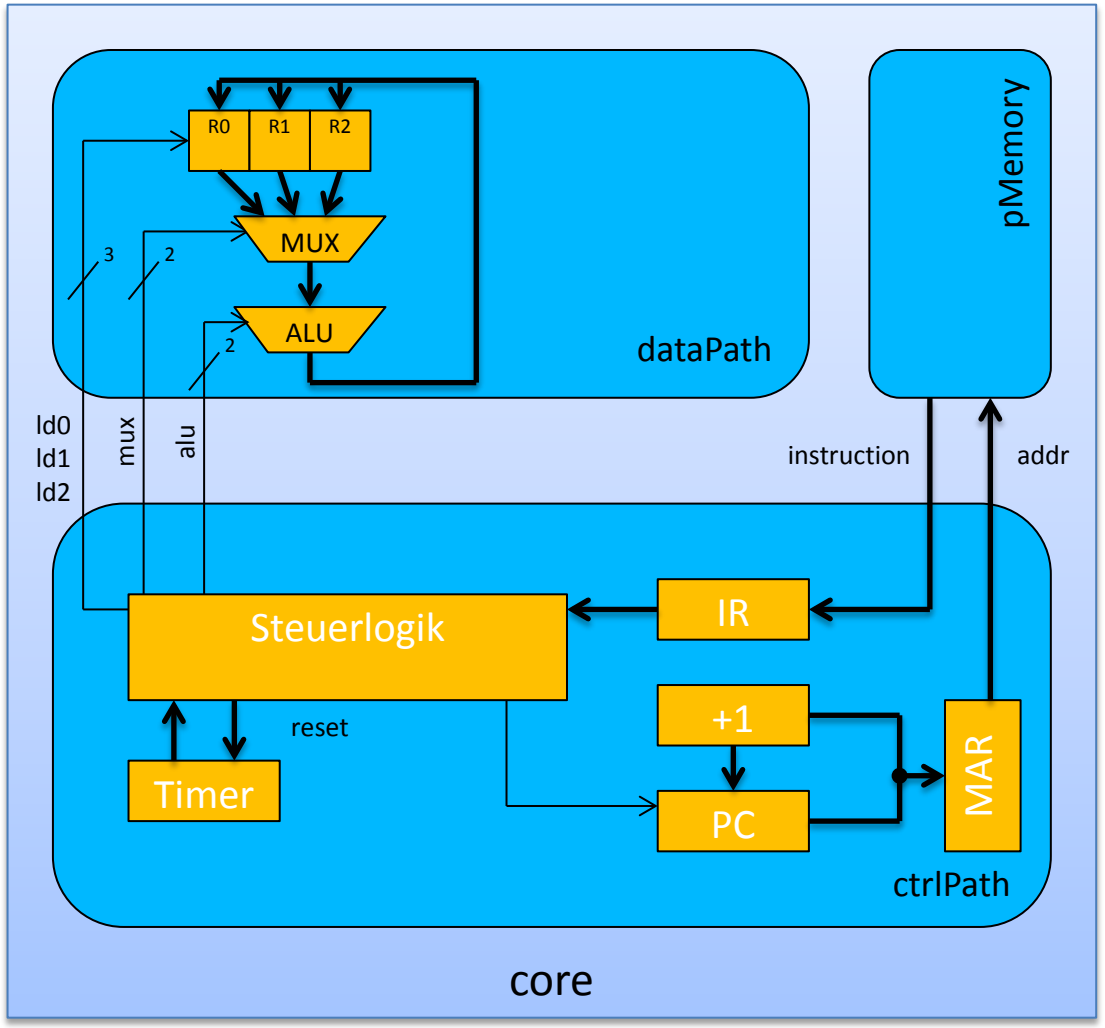



```
architecture structFA of FA is

  component HA is
    port (
      a : in std_logic;
      b : in std_logic;
      s : out std_logic;
      c : out std_logic
    );
  end component;

  signal c1: std_logic;
  signal c2: std_logic;
  signal zs: std_logic;

begin
  ha1: HA port map (
    a => a,
    b => b,
    s => zs,
    c => c1
  );
    ha2: HA port map (
    a => ci,
    b => zs,
    s => s,
    c => c2
  );

  co <= c1 or c2;

end structFA;
```

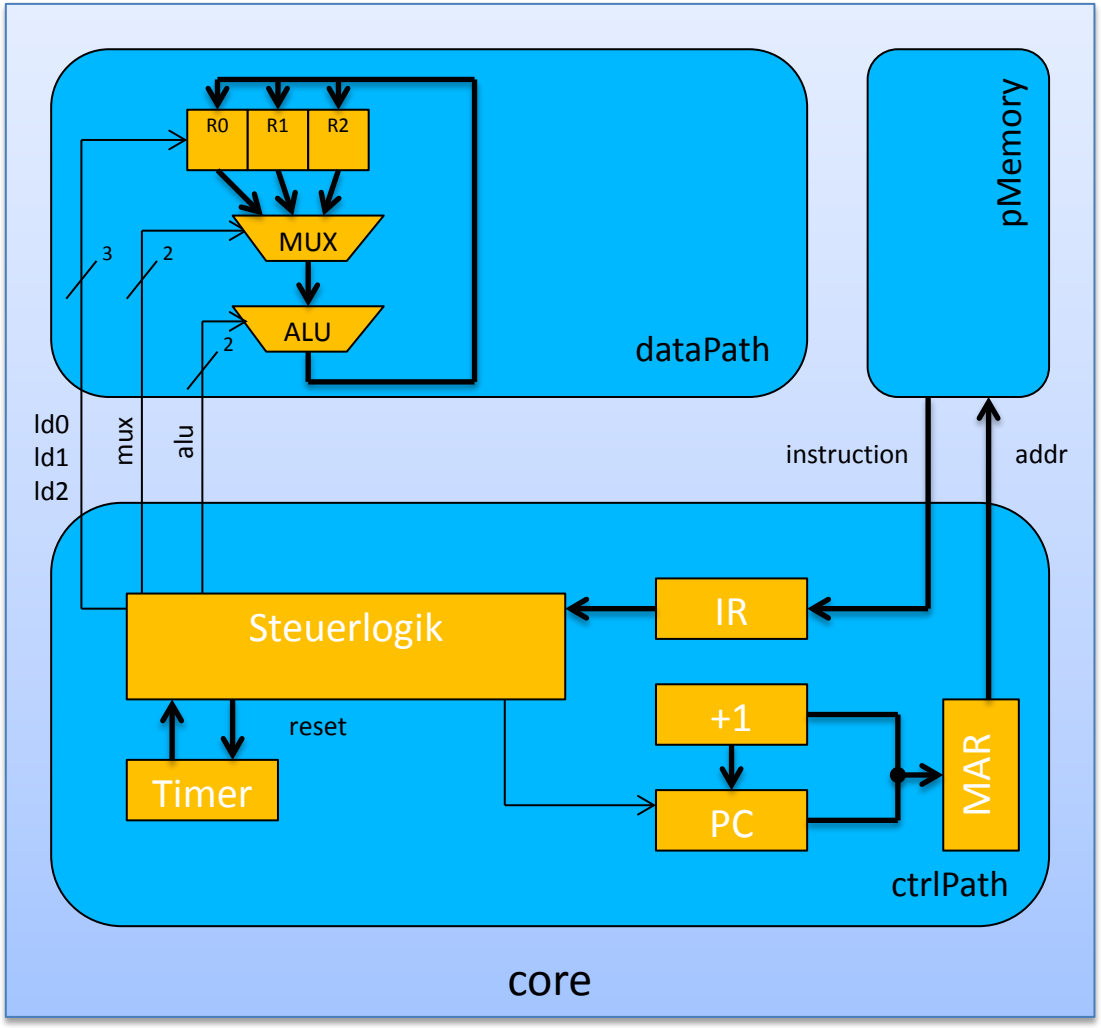
Beispiel: MiniProzessor

- 4 Bausteine
- Hierarchie der Bausteine:





MiniProzessor: Schnittstellen




```
entity dataPath is
  PORT(
    clk     : IN std_logic;
    reset   : IN std_logic;
    ld0     : IN std_logic;
    ld1     : IN std_logic;
    ld2     : IN std_logic;
    mux     : IN std_logic_vector(1 downto 0);
    alu     : IN std_logic_vector(1 downto 0)
  );
end dataPath;
```

```
entity ctrlPath is
  PORT(
    clk     : IN std_logic;
    reset   : IN std_logic;
    addr    : OUT TProgAddr;
    instruction : IN TInstruction;
    ld0     : OUT std_logic;
    ld1     : OUT std_logic;
    ld2     : OUT std_logic;
    mux     : OUT std_logic_vector(1 downto 0);
    alu     : OUT std_logic_vector(1 downto 0)
  );
end ctrlPath;
```

```
entity pMemory is
  PORT(
    reset       : IN std_logic;
    inAddr      : IN TProgAddr;
    instruction : OUT TInstruction
  );
end pMemory;
```

Verdrahtung der Komponenten

```
Library IEEE;
use IEEE.std_logic_1164.ALL;
use work.lib.all;

entity core is
  PORT(
        clk    : IN std_logic;
        reset  : IN std_logic
      );
end core;

architecture simpleCore of core is

  .
  .
  .

  signal addrBus : TProgAddr;
  signal dataBus: TInstruction;
  signal ld0_sig : std_logic;
  signal ld1_sig : std_logic;
  signal ld2_sig : std_logic;
  signal mux_sig : std_logic_vector(1 downto 0);
  signal alu_sig : std_logic_vector(1 downto 0);
begin
  ctrl: ctrlPath port map (...);

  mem: pMemory port map (...);

  dp: dataPath port map (...);
end simpleCore;
```

```
component pMemory is
    port (...);
end component;

component ctrlPath is
    port (...);
end component;

component dataPath is
    port (...);
end component;
```

```
reset       : IN std_logic;
inAddr      : IN TProgAddr;
instruction : OUT TInstruction
```

```
clk         : IN std_logic;
reset       : IN std_logic;
addr        : OUT TProgAddr;
instruction : IN TInstruction;
ld0         : OUT std_logic;
ld1         : OUT std_logic;
ld2         : OUT std_logic;
mux         : OUT std_logic_vector(1 downto 0);
alu         : OUT std_logic_vector(1 downto 0)
```

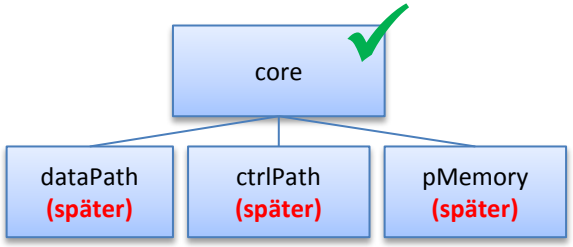
```
clk    : IN std_logic;
reset  : IN std_logic;
ld0    : IN std_logic;
ld1    : IN std_logic;
ld2    : IN std_logic;
mux    : IN std_logic_vector(1 downto 0);
alu    : IN std_logic_vector(1 downto 0)
```

```
clk => clk,
reset => reset,
addr => addrBus,
instruction => dataBus,
ld0 => ld0_sig,
ld1 => ld1_sig,
ld2 => ld2_sig,
alu => alu_sig,
mux => mux_sig
```

```
reset => reset,
inAddr => addrBus,
instruction => dataBus
```

```
clk => clk,
reset => reset,
ld0 => ld0_sig,
ld1 => ld1_sig,
ld2 => ld2_sig,
alu => alu_sig,
mux => mux_sig
```

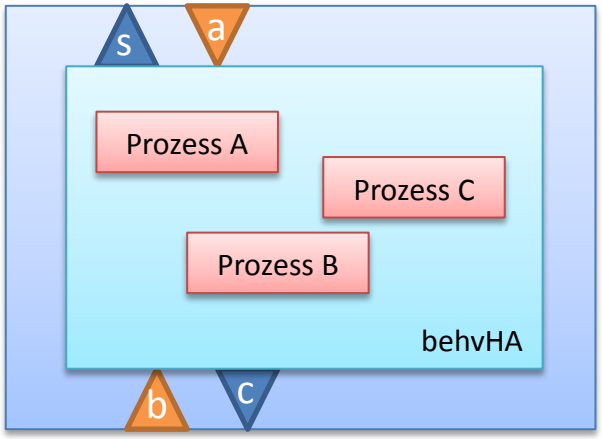
Verhaltensbeschreibung



Verhalten eines Bausteins beschreiben

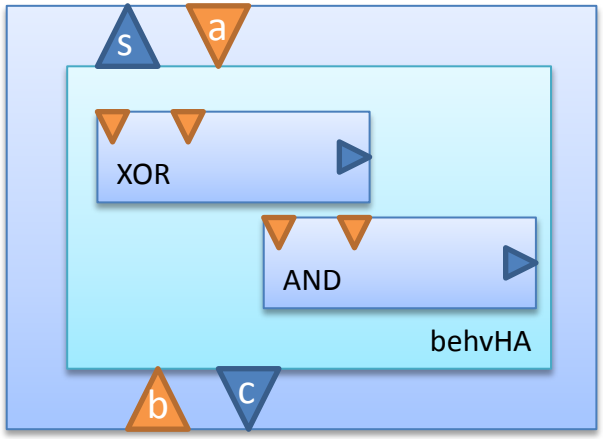
Verhaltensorientiert

- Funktionale Beschreibung des Verhaltens mittels nebenläufiger Prozesse



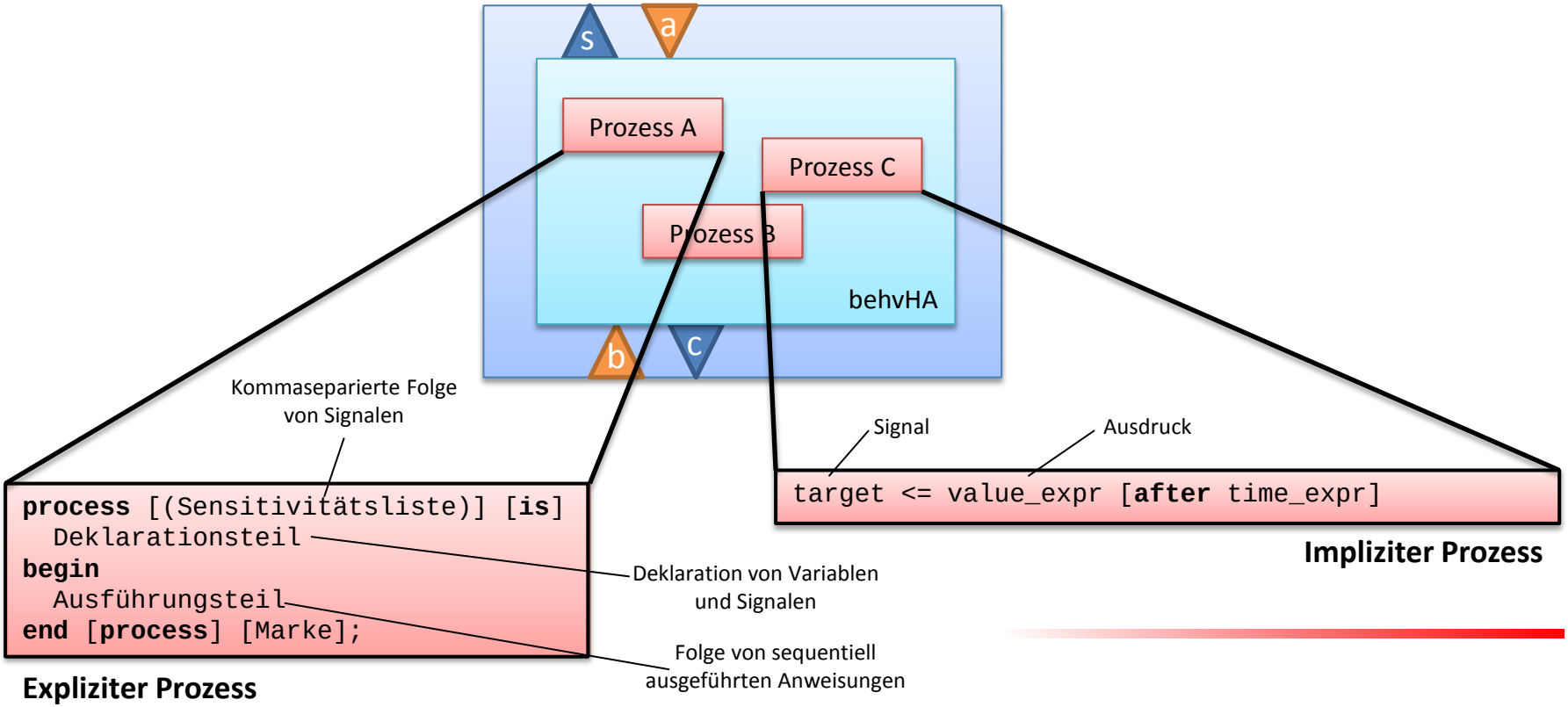
Strukturorientiert

- Aufbau einer Komponente aus Bausteinen mit Schnittstellen
- Verdrahtung der Bausteine



Prozesse

- Prozesse dienen der verhaltensorientierten Beschreibung einer Architektur
- Zwei Arten von Prozessen:
  - Explizite Prozesse
  - Implizite Prozesse (nebenläufige Signalzuweisungen)

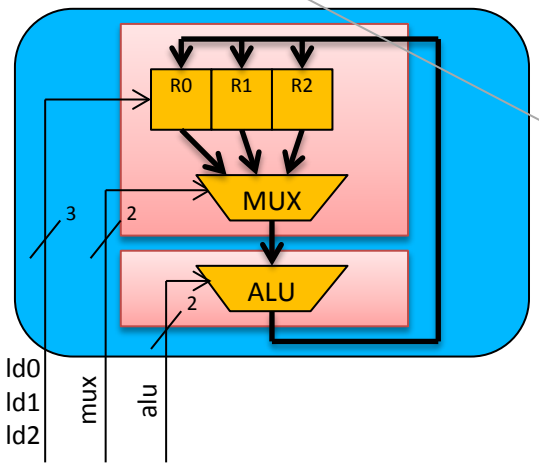


Beispiel Datenpfad

```
architecture behv of dataPath is
  signal operand : TData;
  signal result  : TData;
begin
  RFProcess : process (clk, reset, mux)
    type rfType is array (0 to 2) of TData;
    variable reg : rfType;
  begin
  ...
  end process;

  ALUProcess : process (operand, alu)
  begin
  ...
  end process;
end behv;
```

```
if reset='1' then
    reg(0) := (others => '0');
    reg(1) := (others => '0');
    reg(2) := (others => '0');
elsif clk = '1' and clk'event then
    if ld0 = '1' then
      reg(0) := result;
    elsif ld1 = '1' then
      reg(1) := result;
    elsif ld2 = '1' then
      reg(2) :=  result;
    end if;
end if;

operand <= reg(conv_integer(unsigned(mux)));
```

```
  case alu is
    when "00" =>
      result <= operand;
    when "01" =>
      result <= operand + 1;
    when "10" =>
      result <= operand - 1;
    when others =>
      result <= operand;
  end case;
```



Explizite vs. Implizite Prozesse

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

entity FA is
port (
  a  : in std_logic;
  b  : in std_logic;
  ci : in std_logic;
  s  : out std_logic;
  co : out std_logic
);
end FA;
```

Anweisungsfolge: Abarbeitung in der angegebenen Reihenfolge

```
architecture behavFA of FA is
begin
  fullAdder: process (a, b, ci)
  begin
    s  <= a XOR b XOR ci;
    co <= (a AND b) OR (ci AND (a XOR b));
  end process;
end behavFA;
```

Expliziter Prozess

Nebenläufige Signalzuweisungen: Abarbeitungsreihenfolge nicht festgelegt

```
architecture behavFA of FA is
begin
  s  <= a XOR b XOR ci;
  co <= (a AND b) OR (ci AND (a XOR b));
end behavFA;
```

Implizite Prozesse

Implizite Prozesse

- Impliziter Prozess kann leicht in expliziten Prozess transformiert werden
- Impliziter Prozess:

```
target <= value_expr
```

- Zugehöriger expliziter Prozess:

```
process (signalliste) is
begin
    target <= value_expr
end process;
```

- Dabei enthält *signalliste* alle Signale aus *value_expr*
- Beispiel:

```
s  <= a XOR b XOR ci;
```

```
process (a,b,ci) is
begin
  s <= a XOR b XOR ci
end process;
```

Signale vs. Variablen (1)

```
architecture behavFA of FA is
  [Deklarationsteil_Architektur]
begin
  process (a, b, ci)
    [Deklarationsteil_Prozess]
  begin
    s  <= a XOR b XOR ci;
    co <= (a AND b) OR (ci AND (a XOR b));
  end process;
end behavFA;
```

Deklaration von Signalen

Deklaration von Signalen und Variablen

- Ein Signal dient der Kommunikation zwischen Prozessen/Bausteinen
- Deklaration eines Signals:

```
signal sName {,sName}: Datentyp;
```

- Deklaration in Architekturen, Prozessen, Funktionen/Prozeduren

- Variablen speichern Werte innerhalb eines Prozesses
- Deklaration einer Variablen

```
variable vName {,vName}: Datentyp;
```

- Deklaration nur in Prozessen, Funktionen/Prozeduren

Signale vs. Variablen (2)

- Signalzuweisung:

```
Signal_Name <= Ausdruck
```

- Ausdruck muss vom gleichen Typ wie das Signal sein
- Verhalten:
  - Zuweisung wird gepuffert und erst "später" ausgeführt; wichtig zur Simulation parallelen Verhaltens
- Beispiel :

```
SIGNAL a, b, c: bit;

a <= b;
c <= a;
```

- Variablenzuweisung:

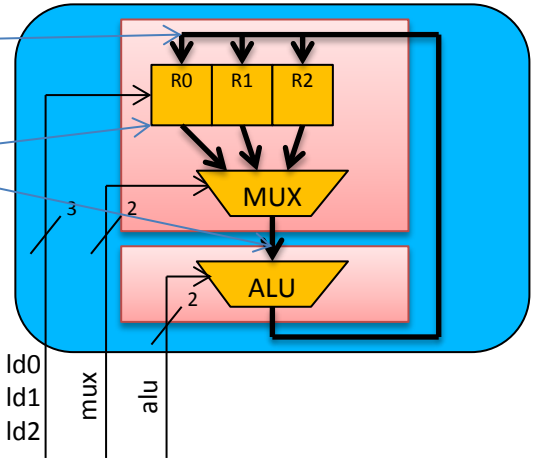
```
Var_Name := Ausdruck
```

- Ausdruck muss gleichen Typ wie die Variable sein
- Verhalten:
  - Zuweisung wird sofort ausgeführt
- Beispiel:

```
VARIABLE a, b, c: bit;

a := b;
c := a;
```

Beispiel Datenpfad

```
architecture behv of dataPath is
  signal result  : TData;
  signal operand : TData;
begin
  RFProcess : process (clk, reset, mux)
    type rfType is array (0 to 2) of TData;
    variable reg : rfType;
  begin
  ...
  end process;

  ALUProcess : process (operand, alu)
  begin
  ...
  end process;
end behv;
```




```
constant WIDTH : natural := 8;
subtype TData is std_logic_vector(WIDTH - 1 downto 0);
```

TData ist 8-Bit Vektor mit MSB an Position 7

Datentypen

- Datentypen definieren eine Menge von Werten und zulässige Operationen auf diesen Werten
- Unterscheidung in:
  - Skalare Datentypen
    - Integer-Datentypen
    - Real-Datentypen
    - Physikalische Datentypen (enthalten Dimensionsangaben)
    - Aufzählungstypen
  - Zusammengesetzte Datentypen
    - Felder
    - Verbunde
- VHDL ist eine streng getypte Sprache
- Zuweisung von Variable/Signal *a* an *b* nur möglich, wenn
  - *a* hat gleichen Typ wie *b*
  - Typ von *a* ist Subtyp von *b*
- Deklaration von Datentypen im
  - Deklarationsteil eines Packages
  - Deklarationsteil einer Architektur
  - Deklarationsteil eines Prozesses

Integer-Datentyp

- Integer-Datentypen haben einen zusammenhängenden Wertebereich ganzer Zahlen
- Formal existiert ein anonymer Datentyp universal_integer, der alle ganzen Zahlen enthält
- Deklaration nicht anonymer Integer-Datentypen

```
type Bezeichner is range Integer_Literal to Integer_Literal;
type Bezeichner is range Integer_Literal downto Integer_Literal;
```

- Beispiel:

```
type integer is range -2147483647 to 2147483647;
```

- Deklarierte Operationen:
  - Relationale Operationen: =, /=, <, <=, >, >=
  - Arithmetische Operationen: +, -, *, / (ganzzahlige Division)
  - Potenzierung: **
  - Absolutwert: abs
  - Modulo: mod
  - Rest nach ganzzahliger Division: rem

Aufzählungstypen

- Deklaration

```
type Bezeichner is (Element {, Element});
```

- Reihenfolge der Aufzählung definiert totale Ordnung auf den Elementen

- Beispiel:

```
type state is (idle, start, stop);

type boolean is (false, true);
type bit is ('0', '1');
type character is (nul, soh, …, ' ', '!', '"', …, 'a', 'b', 'c',…)
```

Attribute auf skalaren Datentypen

Attribute auf skalaren Datentypen T:

- T'base: Basisdatentyp von T
- T'high: maximaler Wert
- T'low: minimaler Wert
- T'left: Wert der linken Bereichsgrenze
- T'right: Wert der rechten Bereichsgrenze
- T'ascending: ist true, gdw. Wertebereich aufsteigend sortiert ist

Beschränkte Felder

Bei der Deklaration werden Bereichsgrenzen festgelegt

Deklaration:

```
type Bezeichner is array (Wertebereich {,Wertebereich}) of Datentyp_name;
```

Beispiel:

```
type byte is array (7 downto 0) of bit;
type wort is array (1 to 4) of byte;
type brett is array (1 to 9, 1 to 9) of integer;

signal a: byte;
```

Zugriff auf Feldelemente:

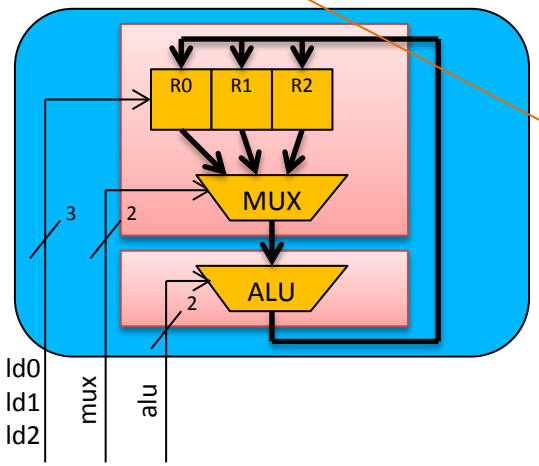
```
a <= ('0', '1', '0', '1', '1', '1', '0', '0');
a <= (7 downto 5 => '1', others => '0');
a(7) <= '0';
a <= (3 => '1', 1 downto 0 => '1', 2 => '0');
```

Beispiel Beschränkte Felder

```
architecture behv of dataPath is
  signal operand : TData;
  signal result  : TData;
begin
  RFProcess : process (clk, reset, mux)
    type rfType is array (0 to 2) of TData;
    variable reg : rfType;
  begin
  ...
  end process;

  ALUProcess : process (operand, alu)
  begin
  ...
  end process;
end behv;
```

```
if reset='1' then
    reg(0) := (others => '0');
    reg(1) := (others => '0');
    reg(2) := (others => '0');
elsif clk = '1' and clk'event then
    if ld0 = '1' then
      reg(0) := result;
    elsif ld1 = '1' then
      reg(1) := result;
    elsif ld2 = '1' then
      reg(2) :=  result;
    end if;
end if;

operand <= reg(conv_integer(unsigned(mux)));
```

```
  case alu is
    when "00" =>
      result <= operand;
    when "01" =>
      result <= operand + 1;
    when "10" =>
      result <= operand - 1;
    when others =>
      result <= operand;
  end case;
```



Unbeschränkte Felder

Bereichsgrenzen werden erst bei der Instanziierung einer Variablen/eines Signals festgelegt

Deklaration:

```
type Bezeichner is array (Disk_Typ range <> {,Disk_Typ <>})
                  of Datentyp_name;
```

Beispiele (aus den packages standard und std_logic_1164):

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

Instanziierung

```
signal Z_Bus: bit_vector(3 downto 0);
signal Z_Bus: std_logic_vector(0 to 3);
```

Zugriff auf Feldelemente wie bisher

Beispiel unbeschränkte Felder

dataPath.vhd

```
.
.
.
architecture behv of dataPath is
  signal result  : TData;
  signal operand : TData;
begin
  RFProcess : process (clk, reset, mux)
    type rfType is array (0 to 2) of TData;
    variable reg : rfType;
  begin
  ...
  end process;

  ALUProcess : process (operand, alu)
  begin
  ...
  end process;
end behv;
.
.
.
```

lib.vhd

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

package lib is
  constant WIDTH : natural := 8;
  constant JMP   : std_logic_vector(7 downto 0) := "10000000";
  constant JZ    : std_logic_vector(3 downto 0) := "1001";
  constant JNZ   : std_logic_vector(3 downto 0) := "1010";
  constant INC   : std_logic_vector(3 downto 0) := "0001";
  constant DEC   : std_logic_vector(3 downto 0) := "0010";
  constant R0    : std_logic_vector(3 downto 0) := "0000";
  constant R1    : std_logic_vector(3 downto 0) := "0001";
  constant R2    : std_logic_vector(3 downto 0) := "0010";
  constant R3    : std_logic_vector(3 downto 0) := "0011";

  subtype TInstruction is std_logic_vector(WIDTH - 1 downto 0);
  subtype TProgAddr is std_logic_vector(WIDTH - 1 downto 0);
  subtype TData is std_logic_vector(WIDTH - 1 downto 0);
end package lib;

package body lib is

end package body;
```

Vordefinierte Datentypen

Im package standard deklarierte skalare Datentypen

- type boolean is (false, true);
- type bit is ('0', '1');
- type character is (nul, soh, …, ' ', '!', '"', …, 'a', 'b','c',…)
- type integer is range -2147483648 to 2147483647;
- type real is range -1.0E308 to 1.0E308;
- type std_ulogic is ('U','X','0','1','Z','W','L','H','-')
- subtype std_logic is resolved std_ulogic;

Im package standard deklarierte Subtypen und Felder

- subtype natural is integer range 0 to integer'high;
- subtype positive is integer range 1 to integer'high;
- type string is array (positive range <>) of character;
- type bit_vector is array (natural range <>) of bit;

Ausführungsteil in Prozessen

Der Ausführungsteil in einem Prozess ist eine Folge von Anweisungen

Anweisungsfolge wird sequentiell abgearbeitet

Anweisungen können sein:

- Signal-/Variablenzuweisungen
- if-Anweisung
- case-Anweisung
- Schleifen
  - for
  - while
  - loop
- wait-Anweisung

if-Anweisung

Syntax:

```
if Bedingung then
  {sequentielle_Anweisung}
{elsif Bedingung then
 {sequentielle_Anweisung}}
[else
 {sequentielle_Anweisung}]
end if;
```

Beispiele:

```
If Bedingung 1 then
  sequentielle Anweisungen
End if;
If Bedingung 2 then
  sequentielle Anweisungen
End if;

...

If Bedingung n then
  sequentielle Anweisungen
End if;
```

```
If Bedingung 1 then
  sequentielle Anweisungen
elsif Bedingung 2 then
  sequentielle Anweisungen
elsif Bedingung 3 then


...

elsif Bedingung n then
  sequentielle Anweisungen
End if;
```

case-Anweisung

Syntax:

```
case Ausdruck is
  when Auswahl => {sequentielle_Anweisung}
  {when Auswahl => {sequentielle_Anweisung} }
end case;
```

Beispiel:

```
case Ausdruck is
  when Wert_1 => Anweisungen
  when Wert_2 | Wert_3 | Wert_4 => Anweisungen
  when Wert_5 to Wert_6 => Anweisungen
  when Wert_7 | Wert_8 to Wert_10 => Anweisungen
  ...

  when Wert_n => Anweisungen
  when others  => Anweisungen
End case;
```

Fehlt die others-Anweisung, dann müssen alle möglichen Fälle explizit aufgezählt werden

Schleifen

Syntax loop:

```
loop
  Anweisung
  ...
  exit when Bedingung
  ...
  Anweisung
end loop
```

Syntax for:

```
for Bezeichner in disk_Werteberich loop
  Anweisungen
end loop
```

Beispiel:

```
for k in 0 to 15 loop
  ...
end loop
```

Syntax while

```
while Bedingung loop
  Anweisungen
end loop
```

Beispiel

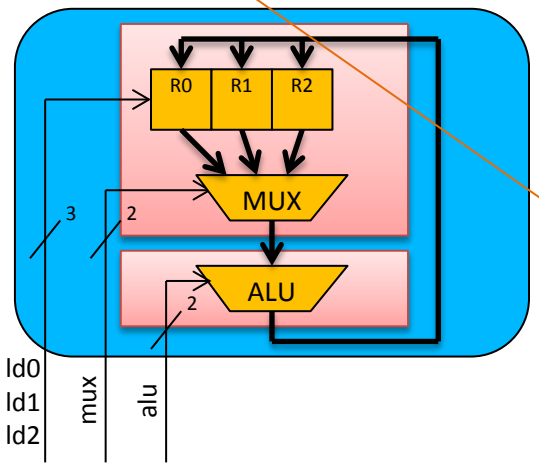
```
while k < 15 loop
  Anweisungen;
  k := k + 1;
end loop
```

Beispiel Datenpfad

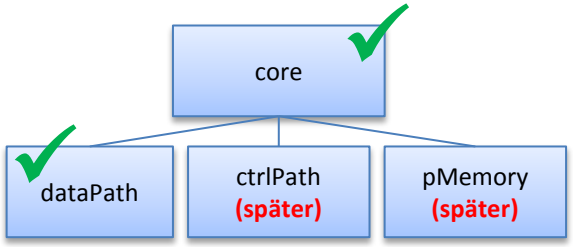
```
architecture behv of dataPath is
  signal operand : TData;
  signal result  : TData;
begin
  RFProcess : process (clk, reset, mux)
    type rfType is array (0 to 2) of TData;
    variable reg : rfType;
  begin
  ...
  end process;

  ALUProcess : process (operand, alu)
  begin
  ...
  end process;
end behv;
```

```
if reset='1' then
    reg(0) := (others => '0');
    reg(1) := (others => '0');
    reg(2) := (others => '0');
elsif clk = '1' and clk'event then
    if ld0 = '1' then
      reg(0) := result;
    elsif ld1 = '1' then
      reg(1) := result;
    elsif ld2 = '1' then
      reg(2) :=  result;
    end if;
end if;

operand <= reg(conv_integer(unsigned(mux)));
```

```
case alu is
  when "00" =>
    result <= operand;
  when "01" =>
    result <= operand + 1;
  when "10" =>
    result <= operand - 1;
  when others =>
    result <= operand;
end case;
```



Verhaltensbeschreibung



Die wait-Anweisung

- Darf nur in Prozessen verwendet werden
- Syntax der *wait*-Anweisung:

```
wait
  [on Signal_Name {, Signal_Name}]
  [until Bedingung]
  [for Zeit];
```

- wait on hält einen Prozess an, bis eine Änderung in einem der angegebene Signale auftritt
- wait until hält einen Prozess an, bis die Bedingung wahr wird
- Es ergibt sich folgende Äquivalenz:

```
wait on s                    wait until s'event
```

- wait for weckt einen Prozess nach der angegebenen Simulationszeit auf

Sensitivitätsliste

- Sensitivitätsliste wird für die Simulation benötigt
- Ein Prozess wird aktiviert, sobald sich ein Signal aus seiner Sensitivitätsliste ändert
- Prozess ohne wait mit nicht leerer Sensitivitätsliste:

```
P: process(clk, res)
begin
  if res = '1' then
    out <= '0';
  elsif clk = '1' and clk'event then
    out <= not in;
  end if;
end process;
```

- Semantisch äquivalenter Prozess mit wait und leerer Sensitivitätsliste:

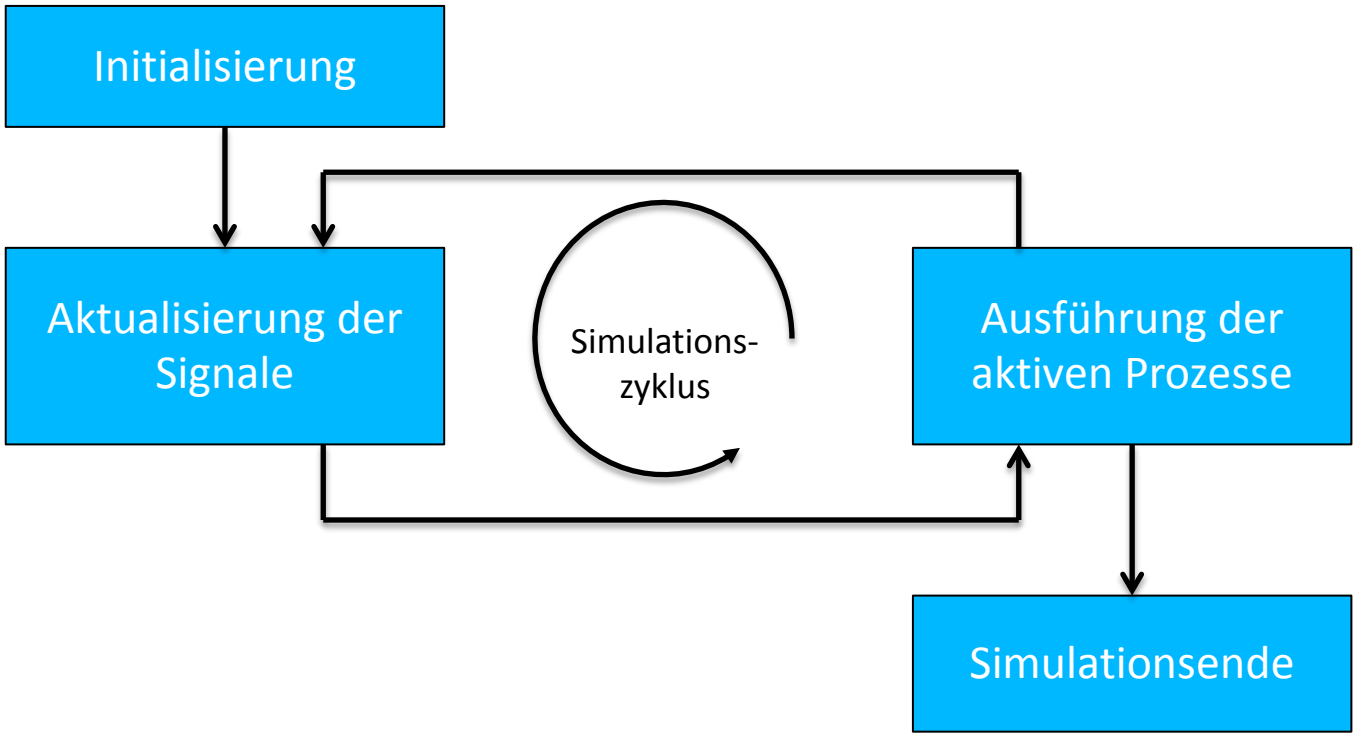
```
P: process
begin
  if res = '1' then
    out <= '0';
  elsif clk = '1' and clk'event then
    out <= not in;
  end if;
  wait on clk, rst;
end process;
```

Inhalt

- Beschreibung von Bausteinen in VHDL
- Simulationssemantik
- Synthesefähige Beschreibungen
  - Kombinatorische Logik
  - Sequentielle Logik

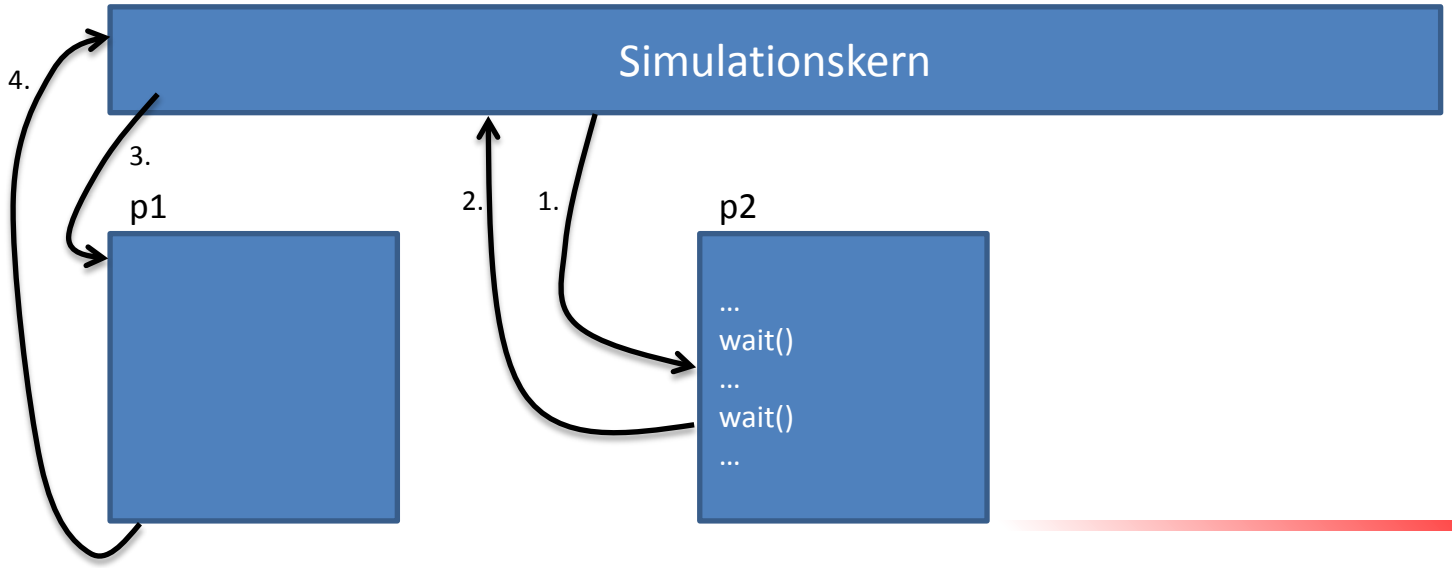
Simulationsprinzip

- Für die Simulation muss auf der untersten Ebene jeder Baustein verhaltensorientiert beschrieben sein
- Wird in einem Prozess keine wait-Anweisung ausgeführt, gerät die Simulation in eine Endlosschleife



Simulationsprinzip

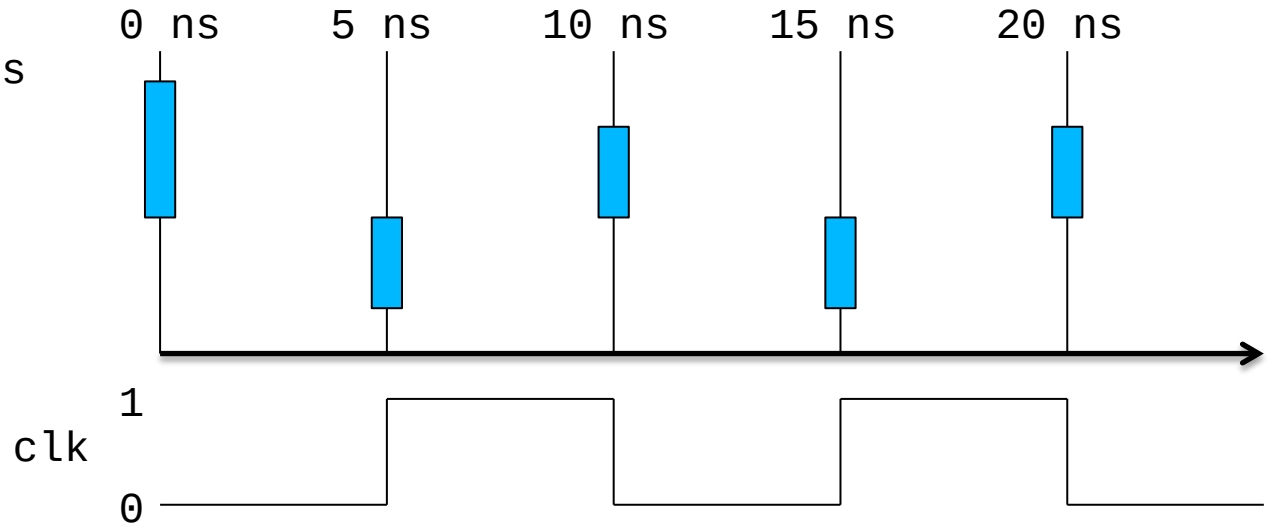
- VHDL verwendet das Prinzip der Co-Routinen.
- D.h.: Jeder Prozess gibt an statisch definierten Punkten (wait-Anweisung) die Kontrolle an den Simulationskern zurück.
- Dieser aktiviert dann einen weiteren Prozess oder beendet die Simulation.
- Prozesse mit nicht-leerer Sensitivitätsliste enthalten am Ende eine implizite wait-Anweisung
- Beispiel:
  - Ereignisse aus den Sensitivitätslisten von p1 und p2 sind aufgetreten
  - Reihenfolge der Aktivierung von p1 und p2 nicht festgelegt



Beispiel:

- Prozess ohne Sensitivitätsliste:

```
1 CLOCK_GEN: process
2 begin
3   clk <= '0';
4   wait for 5 ns;
5   clk <= '1'
6   wait for 5 ns;
7 end process;
```



Initialisierungsphase

- Jedem Signal wird ein Wert zugewiesen, abhängig von seinem Datentyp
- Simulationsuhr wird auf 0 gesetzt
- Alle Prozesse des zu simulierenden Bausteins werden gestartet; d.h. sie werden bis zum ersten wait ausgeführt

Exkurs: Wertzuweisung an Signale

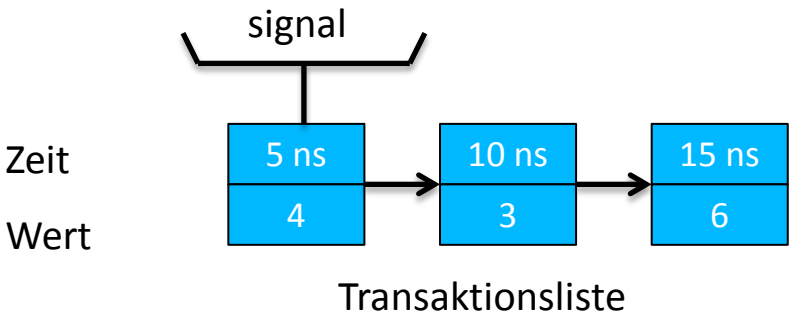
Jedes Signal besitzt eine Transaktionsliste
Wertzuweisung

```
sig <= sig_val after time_val;
```

zum Simulationszeitpunkt curr_time erzeugt Eintrag

(cur_time + time_val, sig_val) in Transaktionsliste

Erst nach Abschluss eines kompletten Simulationszyklus werden Signalwerte in der Aktualisierungsphase aktualisiert:

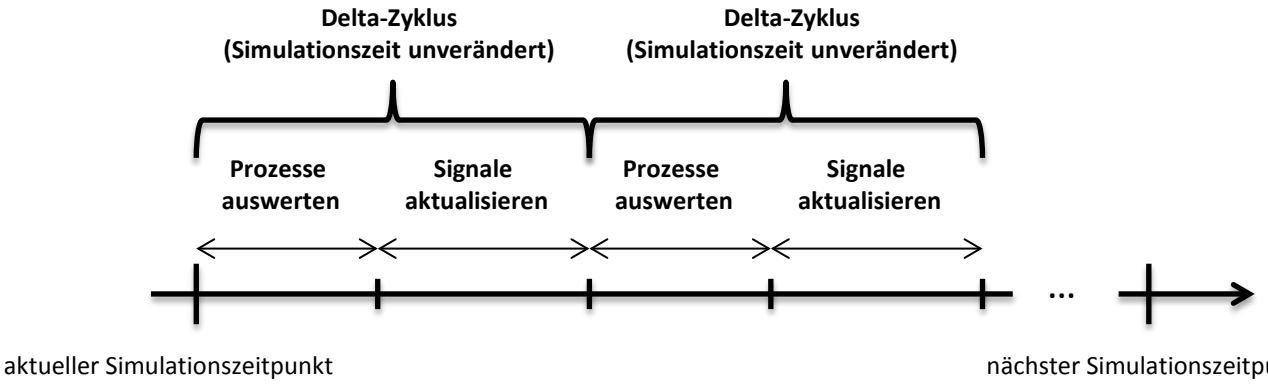


Aktualisierungsphase

- Weitersetzen der aktuellen Simulationszeit t_current auf nächsten Zeitpunkt t_next, an dem ein Ereignis stattfindet.
- Dazu Betrachtung von
  - Transaktionsliste der Signale
  - Weckliste der Prozesse (wegen wait for time_val – Anweisung)
- Ausführen aller Signalzuweisungen, die bei t_next ausgeführt werden müssen
- Ändert das den Wert eines Signals aus einer Sensitivitätsliste, dann wird der zugehörige Prozess geweckt und in der folgenden Ausführungsphase ausgeführt
- Deltazyklen entstehen, wenn t_next = t_current ist

Ausführungsphase/Stoppregeln

- Alle in der letzten Aktualisierungsphase geweckten Prozesse werden ausgeführt
- Ausführung in einer nicht festgelegten Reihenfolge
- Da aber Signalwerte sich während dieser Ausführungsphase nicht ändern (passiert nur in der Aktualisierungsphase) hat die Ausführungsreihenfolge keinen Einfluss auf die Ergebnisse
- Simulation endet, falls
  - Transaktionsliste und Weckliste nach der Ausführungsphase leer oder
  - Vorgegebene Simulationszeit ist erreicht



Inhalt

- Beschreibung von Bausteinen in VHDL
- Simulationssemantik
- Synthesefähige Beschreibungen
  - Kombinatorische Logik
  - Sequentielle Logik
- Zusammenfassung

Beschreiben kombinatorischer Schaltungen

- Ziel: Modellierung einer kombinatorischen Schaltung, die den Wert eines Signals (Leitung) berechnet
  - Ausgabesignale sind funktional abhängig von Eingaben

- Wichtig:
  - Änderung irgend eines Eingangssignals muss in der Simulation Berechnungsprozess aktivieren
  - Bei jeder Aktivierung des Prozesses muss ein Wert auf jedes Ausgangssignale geschrieben werden, sonst kann ein speicherndes Verhalten des nicht beschriebenen Signals entstehen

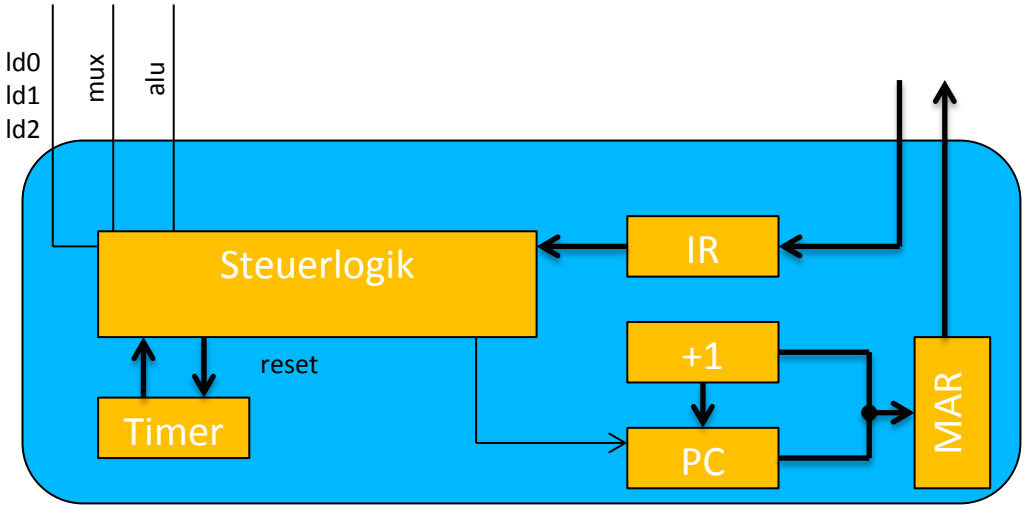
- Modellierung mit:
  - Nebenläufiger Signalzuweisung oder
  - Prozess

Beispiel Steuerlogik

```
Control: process (timer, ir)
begin
  ldMAR <= '0'; ldIR <= '0'; incPC <= '0'; resetTimer <= '0';
  ld0 <= '0'; ld1 <= '0'; ld2 <= '0'; alu <= "00"; mux <= "00";

  if timer = "000" then
    ldMar <= '1';
  elsif timer = "001" then
    ldIR <= '1';
  elsif timer = "010" then
    incPC <= '1';
  elsif timer = "011" then
    if IR(7 downto 4) = "0001" then
      alu <= "01";
      case IR(1 downto 0) is
        when "00"   => ld0 <= '1'; mux <= "00";
        when "01"   => ld1 <= '1'; mux <= "01";
        when "10"   => ld2 <= '1'; mux <= "10";
        when others => mux <= "00";
      end case;
    elsif IR(7 downto 4) = "0010" then
      alu <= "10";
      case IR(1 downto 0) is
        when "00"   => ld0 <= '1'; mux <= "00";
        when "01"   => ld1 <= '1'; mux <= "01";
        when "10"   => ld2 <= '1'; mux <= "10";
        when others => mux <= "00";
      end case;
    end if;
    resetTimer <= '1';
  end if;
end process;
```

Prozess ist sensitiv auf alle verwendeten Signale

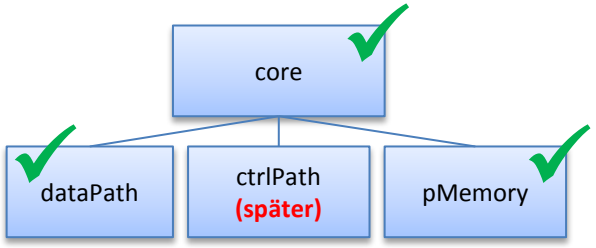
Alle definierten Signale werden bei jeder Aktivierung des Prozesses geschrieben



Speicher (ROM)

```
Library IEEE;
use IEEE.std_logic_1164.ALL;
use IEEE.std_logic_arith.ALL;
use IEEE.std_logic_textio.ALL;
use IEEE.std_logic_unsigned.ALL;
LIBRARY STD;
USE std.textio.ALL;

use work.lib.ALL;

entity pMemory is
  PORT(
        reset  : IN std_logic;
        inAddr : IN TProgAddr;

        instruction : OUT TInstruction
      );
end pMemory;

architecture behv of pMemory is
  constant max_ram_count      : integer :=127;
begin
             memory_process : process (inAddr, reset)
             type ram is array (0 to max_ram_count) of TInstruction;
             variable memory     : ram := (others => (others => '0'));

             begin
               if reset='1' then
                          memory(0) := "0001" & "0000"; -- inc r0
                          memory(1) := "0001" & "0001"; -- inc r1
                          memory(2) := "0001" & "0010"; -- inc r2
                          memory(3) := "0010" & "0001"; -- dec r1
               end if;
               instruction <= memory(conv_integer(unsigned(inAddr)));
             end process;
end behv;
```

Verhaltensbeschreibung



Inhalt

- Beschreibung von Bausteinen in VHDL
- Simulationssemantik
- Synthesefähige Beschreibungen
  - Kombinatorische Logik
  - Sequentielle Logik
- Zusammenfassung

Beschreiben sequentieller Schaltungen

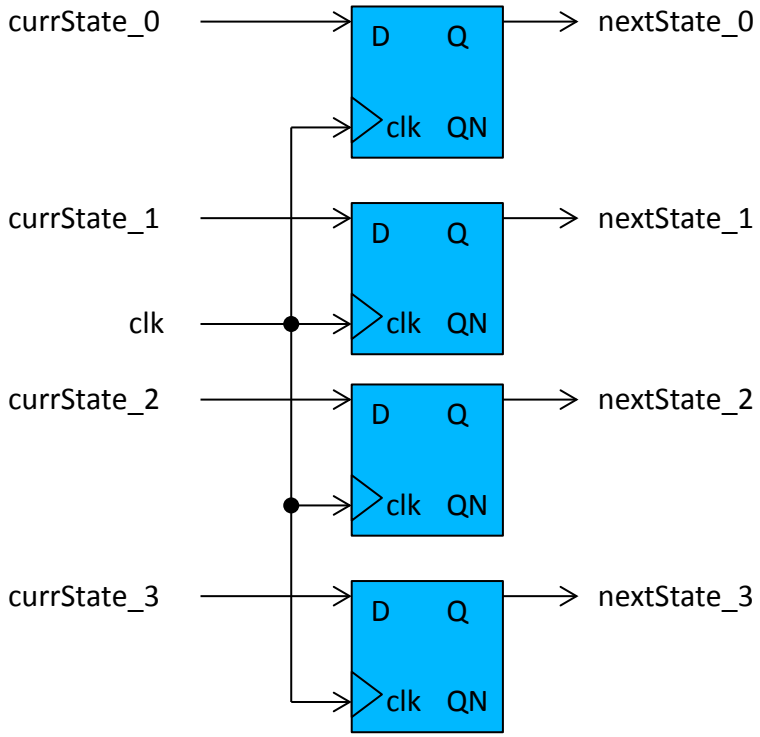
- Sequentielle Schaltungen enthalten Latches oder Flip-Flops
- Beschreibung von Flip-Flops ohne set/clear durch Prozesse mit spezieller Struktur:

```
process
  begin
    wait until clockEdge;
    -- Hier Beschreibung der synchronen Logik
  end process;
end architecture;
```

- clockEdge ist dabei:
  - clock_name = logic-0
  - clock_name = logic-1
  - rising_edge(clock_name) -- nur für std_logic/std_ulogic
  - falling_edge(clock_name) -- nur für std_logic/std_ulogic
- Für Signale, die unter der Kontrolle der wait-Anweisung geschrieben werden, werden Flip-Flops synthetisiert
- Für Variablen, die unter der Kontrolle der wait-Anweisung geschrieben werden, werden Flip-Flops synthetisiert, falls:
  - die Variable gelesen werden kann, bevor sie geschrieben wird
- Wird die Variable immer definiert, bevor sie gelesen wird, dann wird kein speicherndes Element für die geschriebene Variable benötigt

Beispiel: Flip-Flop aus einem Signal

```
entity ff is
  port(
    clk: in bit;
    currState: in integer range 0 to 15;
    nextState: out integer range 0 to 15
  );
end ff;

architecture infer of ff
begin
  process
  begin
    wait until clk = '1';
    nextState <= currState;
  end process;
end architecture;
```



nextState **wird unter der Kontrolle der *wait*-Anweisung geschrieben**

Beispiel: Flip-Flop aus einer Variablen

Flip-Flop wird auch für *temp* erzeugt:

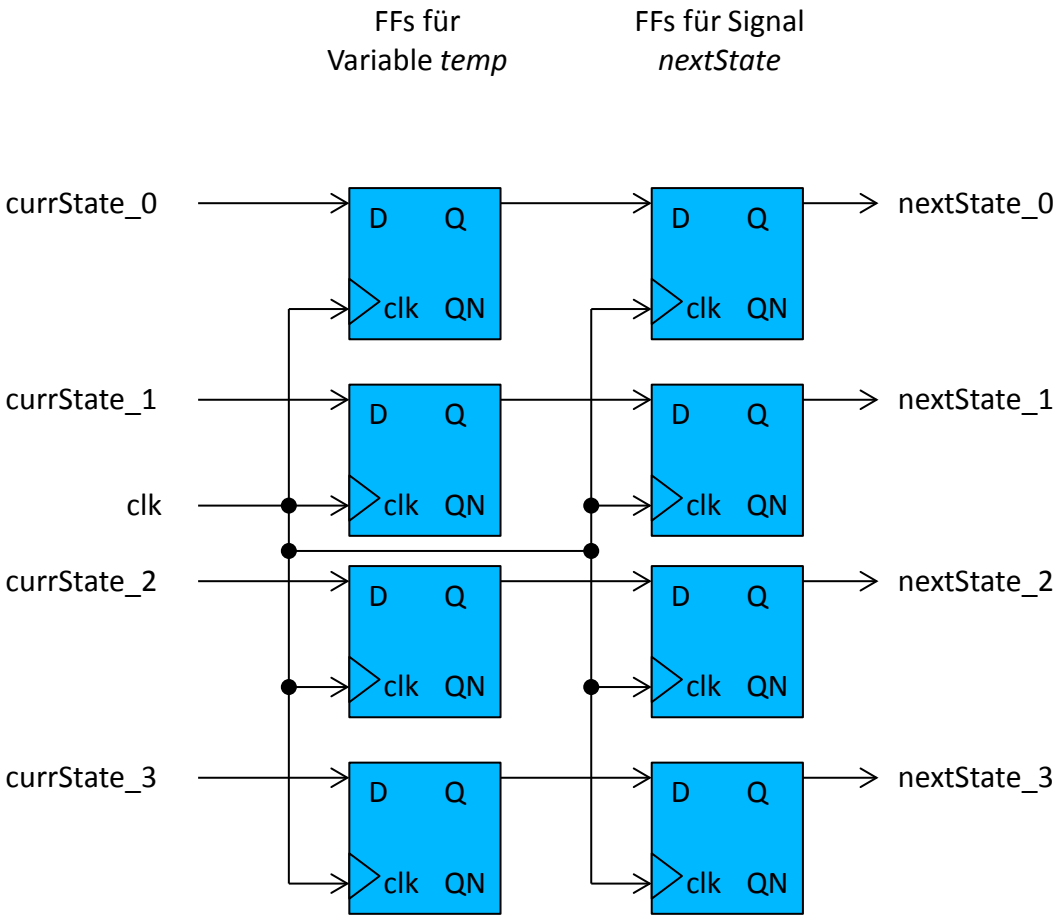
```
process
  variable temp : integer range 0 to 15;
begin
  wait until clk = '1';
  nextState <= temp;
  temp := currState;
end process;
```

Für *temp* wird kein Flip-Flop erzeugt:

```
process
  variable temp : integer range 0 to 15;
begin
  wait until clk = '1';
  temp := currState;
  nextState <= temp;
end process;
```

Schaltbild für den oberen Fall?

Antwort


FFs für
Variable *temp*
FFs für Signal
nextState
currState_0
currState_1
clk
currState_2
currState_3
D
Q
clk
QN
nextState_0
nextState_1
nextState_2
nextState_3

Flip-Flops mit synchronem set/clear

```
process
  begin
    wait until clk = '1';
    if rst = '1' then
      counter <= 0;
    else
      counter <= counter + 1;
    end if;
  end process;
end architecture;
```

Flip-Flops mit asynchronem set/clear

- Beschreibung von Flip-Flops mit asynchronem set/clear durch Prozesse mit spezieller Struktur:

```
process(Sensitivitätsliste)
  begin
    if condition_1 then
      -- Asynchrone Logik 1
    elsif condition_n then
      -- Asynchrone Logik 2
    elsif clock_edge then
      -- Synchrone Logik
    end if;
  end process;
end architecture;
```

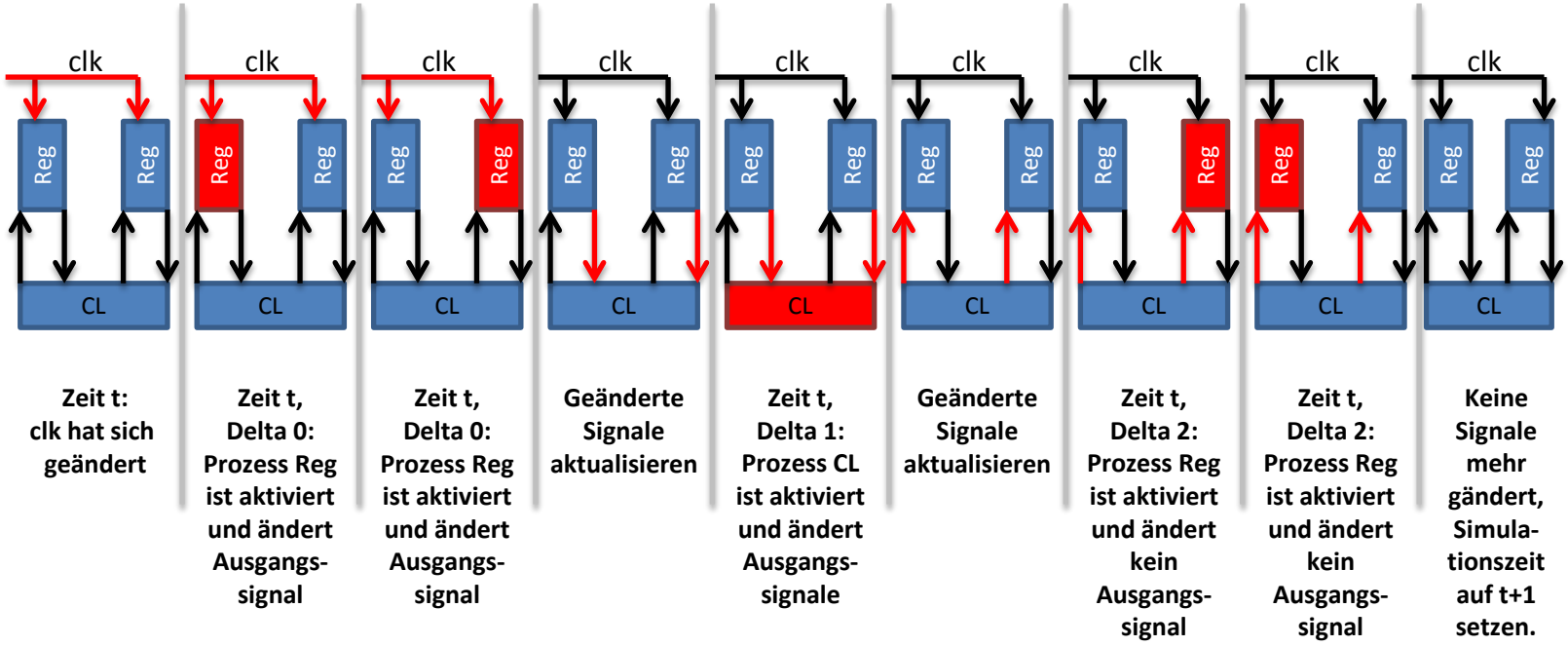
- *clockEdge* ist dabei:
  - clock_name = clk_val and clock_name'event
  - rising_edge(clock_name) -- nur für std_logic/std_ulogic
  - falling_edge(clock_name) -- nur für std_logic/std_ulogic
- Sensitivitätsliste enthält alle gelesenen Signale des asynchronen Teils

Beispiel: Zähler mit asynchronem reset

```
process(clk, rst)
  begin
    if rst='1' then
      counter <= 0;
    elsif clk = '1' and clk'event then
      counter <= counter + 1;
    end if;
  end process;
end architecture;
```

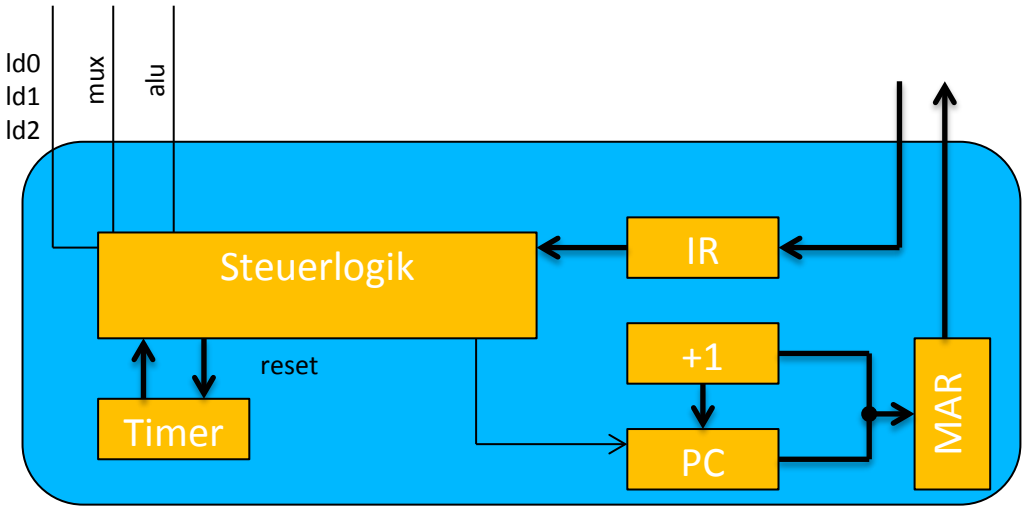
Beispiel: Sequentielle Schaltung und Delta-Zyklus

- Reg ist sequentielle Logik, die ausgehende Signale nur ändert, wenn Aktivierung wegen clk-Signal stattfand.
- CL ist kombinatorische Logik, die immer aktiviert wird, wenn sich eines ihrer Eingangssignale ändert.



Kontrollpfad

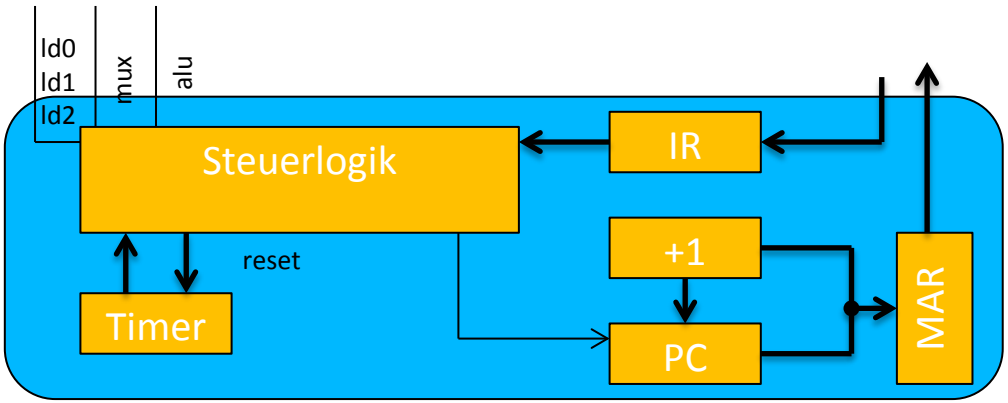
```
architecture behv of ctrlPath is
            signal timer      : std_logic_vector(2 downto 0);
            signal IR         : TInstruction;
            signal PC         : TProgAddr;
            signal MAR        : TProgAddr;
            signal resetTimer : std_logic;
            signal incPC      : std_logic;
            signal ldIR       : std_logic;
            signal ldMAR      : std_logic;
begin
            timerProcess : process (clk, reset)
            begin
              ...
            end process;

            PCProcess : process (clk, reset)
              ...
            end process;

            IRProcess : process (clk, reset)
              ...
            end process;

            MARProcess : process (clk, reset)
              ...
            end process;

            addr <= MAR;

            Control: process (timer, ir)
            begin
              ...
            end process;
end behv;
```



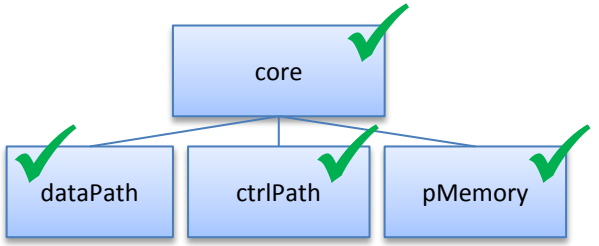
Kontrollpfad

```
timerProcess : process (clk, reset)
begin
  if reset='1' then
    timer <= (others => '0');
  elsif clk = '1' and clk'event then
    if resetTimer = '1' then
      timer <= (others => '0');
    else
      timer <= timer + 1;
    end if;
  end if;
end process;

PCProcess : process (clk, reset)
begin
  if reset='1' then
    PC <= (others => '0');
  elsif clk = '1' and clk'event then
    if incPC = '1' then
      PC <= PC + 1;
    end if;
  end if;
end process;
```

```
IRProcess : process (clk, reset)
begin
  if reset='1' then
    IR <= (others => '0');
  elsif clk = '1' and clk'event then
    if ldIR = '1' then
      IR <= instruction;
    end if;
  end if;
end process;

MARProcess : process (clk, reset)
begin
  if reset='1' then
    MAR <= (others => '0');
  elsif clk = '1' and clk'event then
    if ldMAR = '1' then
      MAR <= PC;
    end if;
  end if;
end process;

addr <= MAR;
```



Verhaltensbeschreibung



Latches

- Beschreibung von Latches durch Prozesse
- Zu einem Signal wird ein Latch generiert, wenn es in dem Prozess einen Pfad gibt, auf dem dieses Signal nicht definiert wird (und der Prozess kein Flip-Flop für das Signal erzeugt)
- Zu einer Variablen wird ein Latch generiert, wenn es in dem Prozess einen Pfad gibt, auf dem dieses Signal nicht definiert wird und es einen Pfad gibt, auf dem die Variable gelesen wird, bevor sie definiert wird
- Beispiel:

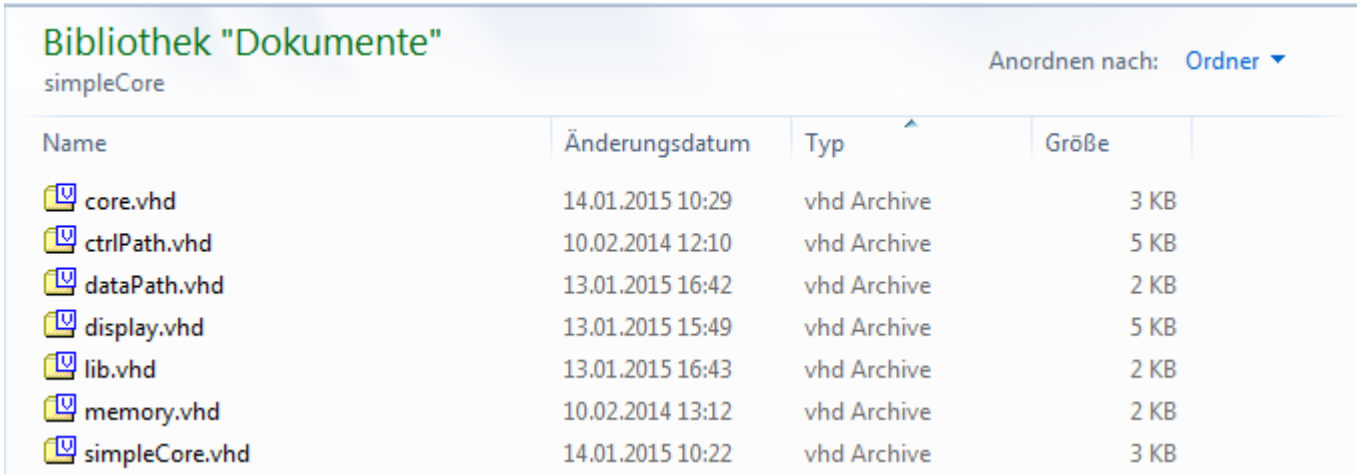
```
process(clk, curr_state)
begin
  if clk = '1' then
    next_state <= curr_state;
  end if;
end process;
```

Inhalt

- Beschreibung von Bausteinen in VHDL
- Simulationssemantik
- Synthesefähige Beschreibungen
  - Kombinatorische Logik
  - Sequentielle Logik
- Zusammenfassung

Quelltexte

- Quelltexte für Beispiel



Strukturierung des VHDL-Codes

- Design besteht in der Regel aus mehreren Komponenten (Bausteinen)
- In einer VHDL-Datei können mehrere Bausteine beschrieben werden
- Im ganzen Design verwendete Datentypen, Operatoren Funktionen/Prozeduren und Komponenten können in packages organisiert werden
- Deklaration eines Packages

```
package package_name is
  [Deklarationen];
end package package_name;

package body package_name is
  [Funktionen/Prozeduren/Bausteine]
end package body;
```

} Definition der Schnittstelle

} Implementierung => Bibliothek

- Einbinden einer Bibliothek:

```
library library_name;
use library_name.package_name.all
```

- Bibliothek work enthält übersetze Quellen des aktuellen Projekts
- Eigene Packages werden auch in Bibliothek work abgelegt werden

Beispiel für eigene Bibliothek

lib.vhd

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

package lib is
  constant WIDTH : natural := 8;
  constant JMP   : std_logic_vector(7 downto 0) := "10000000";
  constant JZ    : std_logic_vector(3 downto 0) := "1001";
  constant JNZ   : std_logic_vector(3 downto 0) := "1010";
  constant INC   : std_logic_vector(3 downto 0) := "0001";
  constant DEC   : std_logic_vector(3 downto 0) := "0010";
  constant R0    : std_logic_vector(3 downto 0) := "0000";
  constant R1    : std_logic_vector(3 downto 0) := "0001";
  constant R2    : std_logic_vector(3 downto 0) := "0010";
  constant R3    : std_logic_vector(3 downto 0) := "0011";

  subtype TInstruction is std_logic_vector(WIDTH - 1 downto 0);
  subtype TProgAddr is std_logic_vector(WIDTH - 1 downto 0);
  subtype TData is std_logic_vector(WIDTH - 1 downto 0);
end package lib;

package body lib is

end package body;
```

Zusammenfassung

- Strukturorientierte/Verhaltensorientierte hierarchische Beschreibung von Bausteinen
- Simulationssemantik in VHDL
- Beschreibung kombinatorischer Prozesse
- Beschreibung sequentieller Prozesse