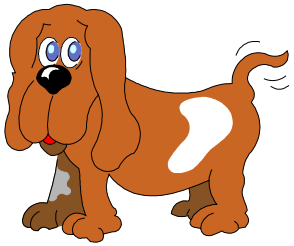
Concurrent Statements



Architecture Bodies

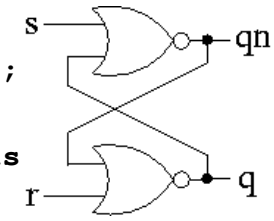
- Die Architecture beschreibt den Inhalt eines Bausteins mit Hilfe von
  - Zuweisungen (Signal Assignments)
  - Komponenten (Strukturdefinition)
  - Prozessen (sequentiell ablaufende Module)
- Anweisungen einer Architecture werden gleichzeitig (!) ausgeführt
  - Ihre Reihenfolge spielt keine Rolle

Architecture Bodies (cont.)

- **architecture** *identifier* **of** *entity_name* **is**
  { block_declarative_item }
  begin
  { concurrent_statement }
  end *identifier*;
- Zu einer Entity können mehrere Architectures gehören
- In block_declarative_item können unter anderem Signale und Konstanten deklariert werden.

Beispiel: RS-Flipflop

- ```
  library ieee;
  use ieee.std_logic_1164.all;
  ```
- ```
  entity rsff is
     port(r, s  : in  std_logic;
          q, qn : out std_logic);
  end rsff;
  ```
- ```
  architecture illegal of rsff is
  begin
     q  <= r nor qn after 10 ns;
     qn <= s nor q  after 10 ns;
  end illegal;
  ```


  ```

Beispiel: RS-Flipflop (cont.)

```
architecture legal of rsff is
   signal qn_int, q_int : std_logic;
begin
   q_int  <= r nor qn_int after 10 ns;
   qn_int <= s nor q_int  after 10 ns;
   q  <= q_int;
   qn <= qn_int;
end legal;
```

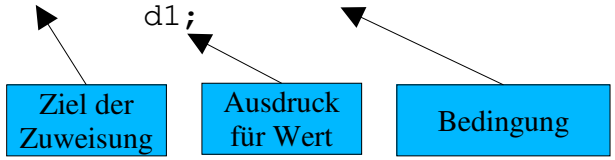
- Die in der Entity definierten Signale können in der Architecture verwendet werden, aber nur in der angegebenen Richtung

Anmerkung: Verifikation

- Diese Implementierung eines RS-Flipflops ist für die formale Verifikation sehr problematisch:
  - Zeitverhalten kann normalerweise nicht modelliert werden
  - Die kombinatorische Schleife muss deshalb durch ein "Verzögerungselement" aufgebrochen werden, wenn sich kein stabiler Zustand beweisen lässt
  - Dadurch geht in der Regel die strukturelle Äquivalenz verloren

Conditional Signal Assignment

```
dout <= d0 when sel = '0' else
        d1;
```



- Signalzuweisung wird aktiviert, wenn die Bedingung erfüllt ist.
- Allgemein:

```
name <= { waveform
          when boolean-expression else }
        waveform ;
```


Multiplexer

```
library ieee;
use ieee.std_logic.all;
```

```
entity MUX41 is
   port(sel0, sel1      : in  std_logic;
        d0, d1, d2, d3  : in  std_logic;
        dout            : out std_logic);
end MUX41;
```

```
architecture behave of MUX41 is
begin
  dout <= d0 when sel1 = '0' and sel0 = '0'
     else d1 when sel1 = '0' and sel0 = '1'
     else d2 when sel1 = '1' and sel0 = '0'
     else d3 when sel1 = '1' and sel0 = '1';
end behave;
```

Tristate-Treiber

```
library ieee;
use ieee.std_logic.all;
```

```
entity tristate is
   port(d, en : in  std_logic;
        y     : out std_logic);
end tristate;
```

```
architecture behave of tristate is
begin
  y <=  d when en = '1' else -- enable
       'Z';
end behave;
```


Signal Assignment (cont.)

- In VHDL93 darf der letzte else-Zweig fehlen
- Soll das Signal in einem Fall nicht geschrieben werden, kann in VHDL93 das Schlüsselwort unaffected statt einem Wert angegeben werden

```
d <= d0          when sel0 = '0' and
                      sel1 = '0' else
     unaffected when sel0 = '1';
```


Selected Signal Assignment

```
library ieee; use ieee.std_logic.all;
```

```
entity MUX41 is
   port(sel0, sel1      : in  std_logic;
        d0, d1, d2, d3  : in  std_logic;
        dout            : out std_logic);
end MUX41;
```

```
architecture rtl of MUX41 is
begin
   with sel1 & sel1 select
       dout <= d0 when "00",
               d1 when "01",
               d2 when "10",
               d3 when others;
end rtl;
```


Beispiel: Integer-ALU

- ALU mit Eingangssignalen a und b sowie einem Steuereingang sel
- Funktionstabelle:

sel	Funktion
0	a + b
1	a -- b
2	a × b
sonst	-a

Beispiel: Integer-ALU (2)

```
entity alu_int is
   port (a, b    : in  integer;
         sel     : in  integer;
         result  : out integer);
end alu_int;
```

```
architecture behave of alu_int is
begin
   with sel select
       result <= a + b when 0,
                 a - b when 1,
                 a * b when 2,
                 -a    when others;
end behave;
```

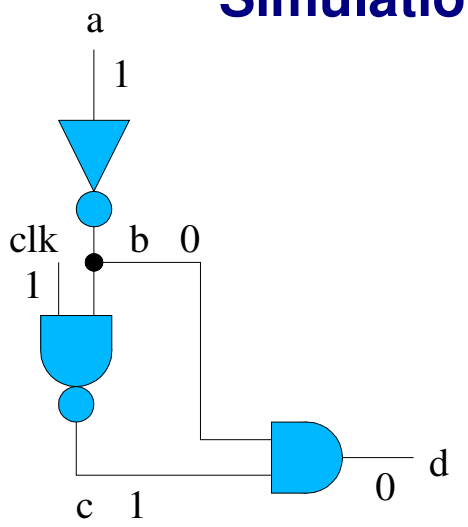

Simulation

- Die Reihenfolge der Auswertung ist wichtig!
- Beispiel:

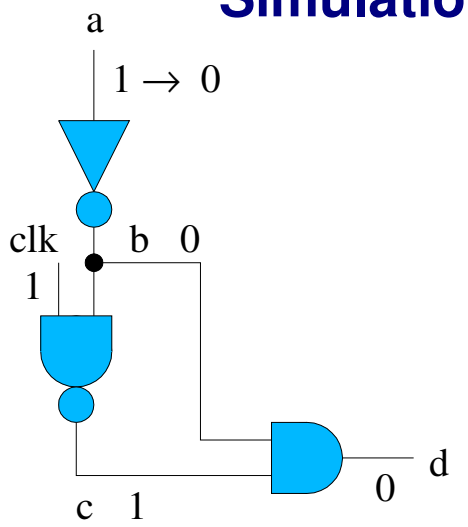
```
entity sim_ex is
   port(a, clk : in  bit;
        d      : out bit);
end sim_ex;
```

```
architecture test of sim_ex is
signal b, c : bit;
begin
   b <= not a;
   c <= clk nand b;
   d <= c and b;
end test;
```


Simulation (2)



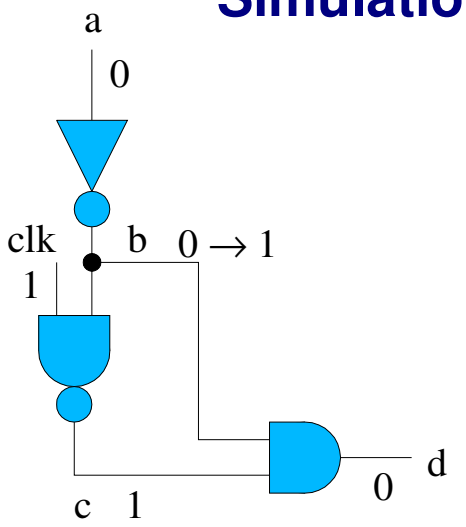
Simulation (2)



Reihenfolge A

1. a: 1 → 0

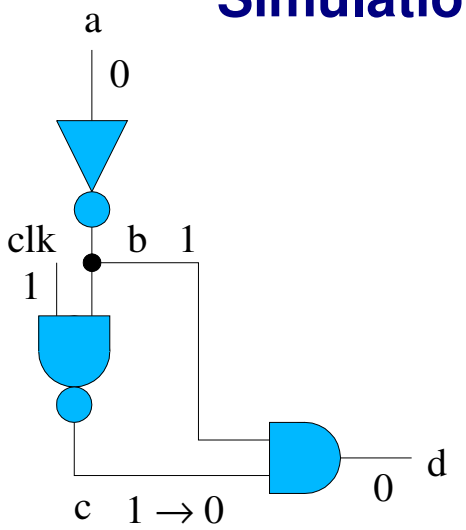
Simulation (2)



Reihenfolge A

1. a: 1 → 0
2. b: 0 → 1

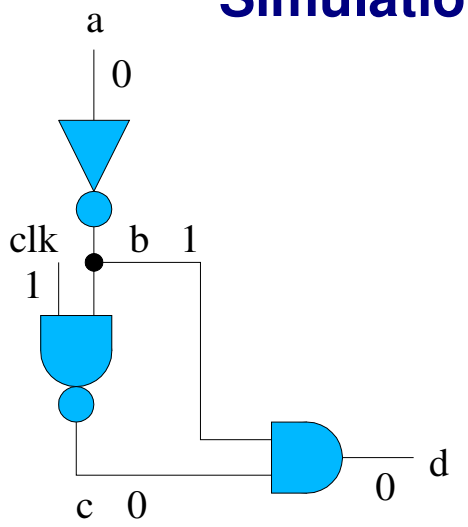
Simulation (2)



Reihenfolge A

1. a: 1 → 0
2. b: 0 → 1
3. c: 1 → 0

Simulation (2)

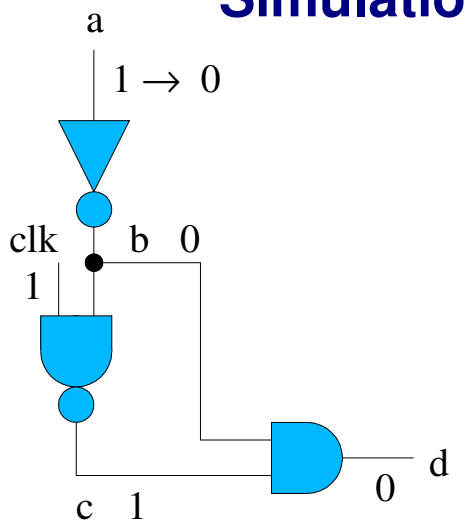


Reihenfolge A

1. a: 1 → 0
2. b: 0 → 1
3. c: 1 → 0
4. d: 0 → 0

fertig.

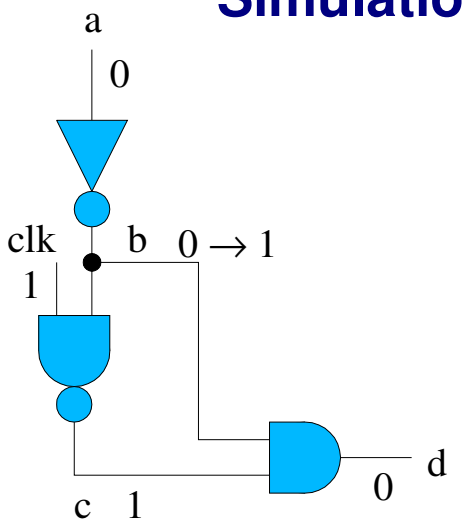
Simulation (3)



Reihenfolge B

1. a: 1 → 0

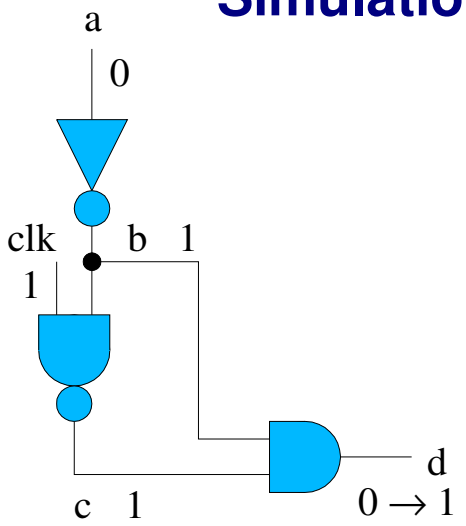
Simulation (3)



Reihenfolge B

1. a: 1 → 0
2. b: 0 → 1

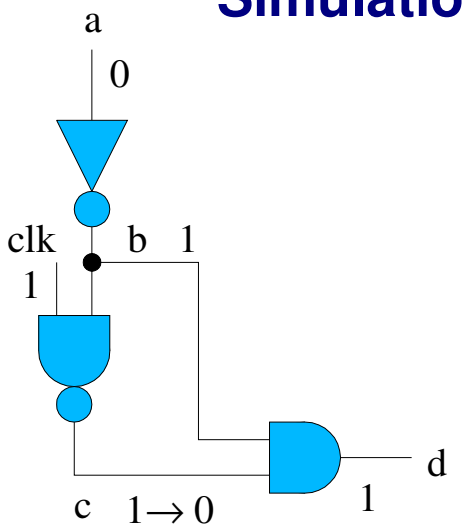
Simulation (3)



Reihenfolge B

1. a: 1 → 0
2. b: 0 → 1
3. d: 0 → 1

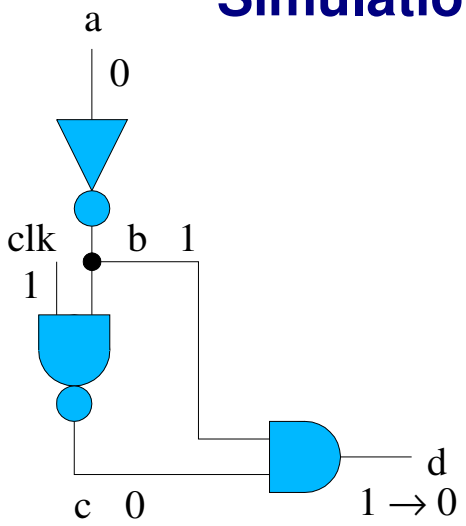
Simulation (3)



Reihenfolge B

1. a: 1 → 0
2. b: 0 → 1
3. d: 0 → 1
4. c: 1 → 0

Simulation (3)



Reihenfolge B

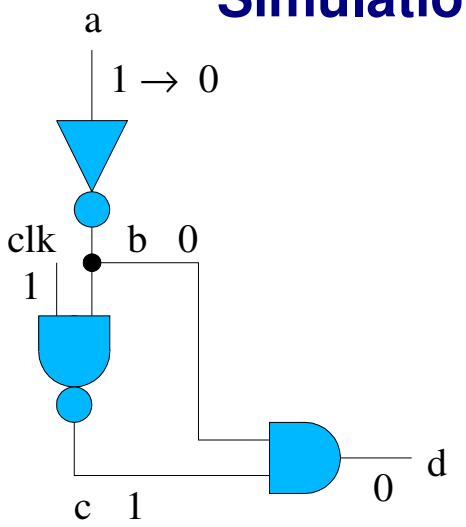
1. a: 1 → 0
2. b: 0 → 1
3. d: 0 → 1
4. c: 1 → 0
5. d: 1 → 0

fertig.

Simulation (4)

- Wert des Ausgangs hängt von der **Reihenfolge der Auswertung** ab!
- Deshalb legt die VHDL-Semantik diese Reihenfolge fest.

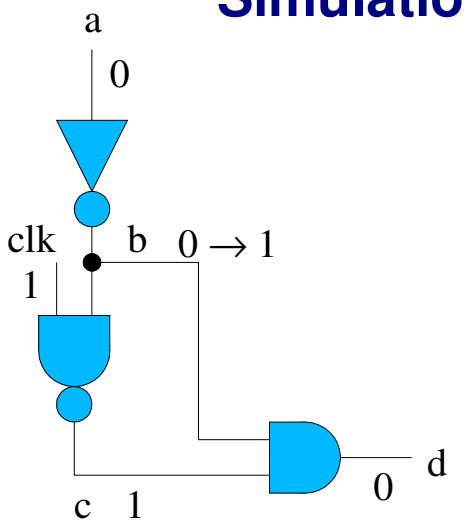
Simulation (5)



VHDL

1. a: 1 → 0

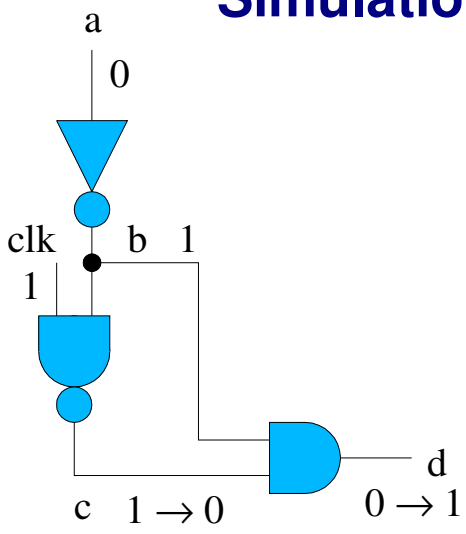
Simulation (5)



VHDL

1. a: 1 → 0
2. b: 0 → 1

Simulation (5)

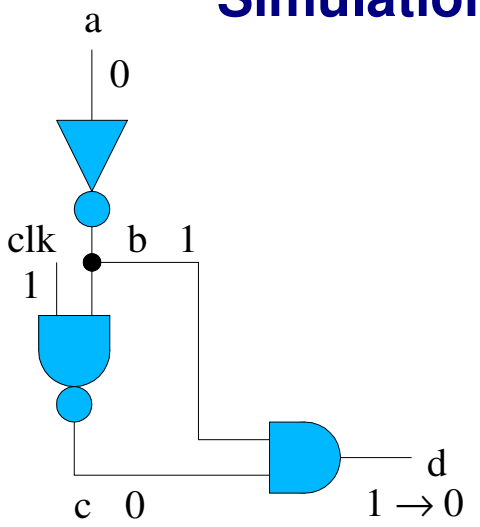


VHDL

1. a: 1 → 0
2. b: 0 → 1
3. c: 1 → 0

d: 0 → 1

Simulation (5)



VHDL

1. a: 1 → 0
2. b: 0 → 1
3. c: 1 → 0

   d: 0 → 1
4. d: 1 → 0

   fertig.

Delta-Delays

- Jeder solche "Elementarschritt" dauert 0 fs und wird Delta-Delay genannt.
- Zuweisungen an Signale werden nicht sofort ausgeführt, sondern erst einen Delta-Schritt später
- Zugrunde liegt das Modell eines diskreten Ereignis-gesteuerten Simulators
- Es wird simuliert, bis keine Ereignisse mehr anliegen

Modellfehler

- Ein fehlerhaftes Modell kann dazu führen, dass immer neue Ereignisse anliegen.
- Beispiel:

```
signal s : bit;
...
s <= not s;
```

- Auch in Hardware ist der Zustand nicht stabil.

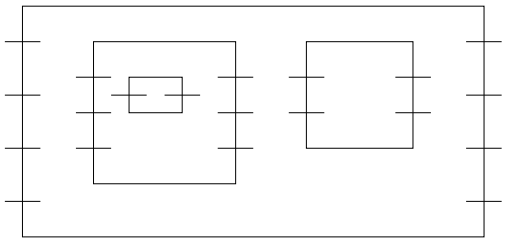
Delta-Schritte und Synthese

- Man sollte sich auf keinen Fall darauf verlassen, dass die **Synthese** Delta-genau übersetzt!
- Man sollte sich auf keinen Fall darauf verlassen, dass die **Verifikation** Delta-Schritte korrekt modelliert!

Komponenteninstanziierung

- Schaltungen werden üblicherweise in einzelne Komponenten zerlegt (strukturelle Beschreibung)
  - Komponenten können aus einzelnen Gattern bestehen (Bibliothek)
  - Komponenten können weitere Komponenteninstanzen enthalten
  - Komponenten können beliebig komplex werden

Komponenten



Component Declaration

```
architecture rtl of something is

   component multi_and
      port ( args : in std_logic_vector
                          (num_ops-1 downto 0) );
   end component;

   signal s : std_logic_vector(7 downto 0);

begin
   ...
```

Interfaceliste analog zu Entities

Component Instantiation

```
architecture rtl of some_entity is

   ...

begin
   ...
ci: multi_and
      port map(args => s);
   ...
end rtl;
```

In VHDL93 darf hier das Schlüsselwort "component" stehen

Syntax

- component_instantiation_statement <=
  label: *entity_name* [(*architecture_name*)]
  [**port map** (port_association_list)] ;
- port_association_list <=
  [*port_name* =>]
  (*signal_name* | **open**) { , ... }

Beispiele

- `signal a, b, c, d : integer;`
- `u1 : alu_int port map (a, b, c, d);`
- ```
  u2 : alu_int
       port map (a => a, b => b,
                 sel => c, result => d);
  ```
- ```
  u3 : alu_int
       port map (a => a, b => b,
                 sel => c, result => open);
  ```


Port Association List

- Verbindet Ports der Instanz (formal) mit Signalen der umgebenden Komponente (actual)
- Ähnlich zu Record Aggregates gibt es:
  - positional association
  - named association
- Ports können unverbunden bleiben ("open")
  - Eingänge bekommen dann den Default-Wert
  - Unconstrained Arrays dürfen nicht "open" sein

Array Ports

- Array Ports können als ganzes oder elementweise verbunden werden
  - Aber: einzelne Elemente dürfen nicht offen gelassen werden, d.h. entweder vollständig offen oder gar nicht
- Slices können sowohl für formal als auch actual-Parameter verwendet werden:

```
port map ( a(0 to 2) => b(3 downto 1),
           a(3) => b(7), a(4) => b(7) );
```

Erweiterungen in VHDL93

- Entities können direkt instantiiert werden, d.h. ohne Deklaration einer Component
  - `u4 : entity alu_int port map (a, b, c, d);`
- Ports der Richtung "in" können durch Ausdrücke sein
  - `u5 : alu_int port map (a, a+b, c, d);`
  - In VHDL87 muss stattdessen ein separates Signal verwendet werden
  - ```
    s <= a + b;
    u87 : alu_int port map (a, s, c, d);
    ```


Wahl der Architecture

- Wird eine Architecture angegeben, so wird diese verwendet
- Die Architecture kann auch in einer "configuration" angegeben werden (siehe später)
- *Design-Regel*: Die Architecture sollte entweder angegeben werden oder eindeutig sein

Assertions

- concurrent_assertion_statement <=
  [label:] **assert** boolean_expression
  [**report** string_expression]
  [**severity** severity_expression] ;
- Überprüfen von Bedingungen
- boolean_expression: zu überwachender Ausdruck
- string_expression: Text, der ausgegeben wird
- severity_expression: Aktion des Simulators

Severity Level

- Was soll der Simulator machen, falls die Bedingung falsch ist?
- Vordefiniert in VHDL:
  ```
  type severity_level is
        ( note, warning, error, failure );
  ```
- Abhängig vom Simulator
  ```

Beispiel

```
entity RS_flipflop is
   port (R, S : in bit; Q : out bit);
end RS_flipflop;
```

```
architecture check of RS_flipflop is
begin
   assert R = '1' nand S = '1'
      report "Both R and S are active.";
   ...
end check;
```

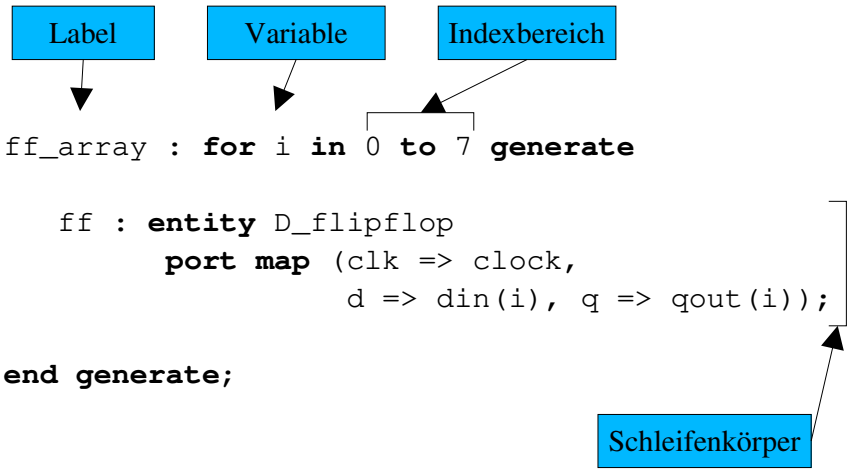

Assertions und Synthese

- Assert-Anweisungen brauchen nicht synthetisiert zu werden
- Sie können zur Optimierung der Logik verwendet werden
- In der formalen Verifikation können Assertions
  - verwendet werden, um unerreichbare Zustände zu erkennen, oder
  - formal bewiesen werden.

Generate-Statements

- Viele Schaltungen haben eine reguläre, iterativ beschreibbare Struktur
  - Beispiel: Speicher
- Generate-Statements erlauben es, Anweisungen wiederholt bzw. bedingt zu instanziieren

Iterative Strukturen



Iterative Strukturen (2)

- *generate_label* :
  for *identifier* **in** *discrete_range* **generate**
  { concurrent_statement }
  end generate [*generate_label*] ;
- Die Schleifenvariable braucht nicht (darf nicht) deklariert zu werden

Beispiel

```
cell_array: for i in 0 to width-1 generate
   signal data_unbuffered : std_logic;
begin
   cell_storage : entity D_flipflop
         port map(clk => clock, d => din(i),
                  q => data_unbuffered);
   cell_buffer : entity tristate_buffer
         port map(a => data_unbuffered,
                  en => enable, y => dout(i));
end generate cell_array;
```


Iterative Strukturen: VHDL93

- *generate_label* :
  for *identifier* **in** *discrete_range* **generate**
  { block_declarative_item }
  [**begin**]
  { concurrent_statement }
  end generate [*generate_label*] ;

wie bei einer Architecture

Konditionale Strukturen

- generate_*label* :
  if *boolean_expression* **generate**
  { concurrent_statement }
  end generate [*generate_label*] ;
- Kann verwendet werden, wenn bestimmte Elemente einer iterativen Struktur eine Ausnahme bilden
- Der Boolesche Ausdruck muss *statisch* sein

Konditionale Strukturen: VHDL93

- generate_*label* :
  if *boolean_expression* **generate**
  { block_declarative_item }
  [**begin**]
  { concurrent_statement }
  end generate [*generate_label*] ;

Beispiel

```
cell_array: for i in 0 to width-1 generate
   signal data_unbuffered : std_logic;
begin
   cell_storage : entity D_flipflop
         port map(clk => clock, d => din(i),
                  q => data_unbuffered);
g1: if i = 0 generate
        dout(i) <= data_unbuffered;
    end generate;
g2: if i /= 0 generate
        cell_buffer : entity tristate_buffer
              port map(a => data_unbuffered,
                       en => enable, y =>
   dout(i));
    end generate;
end generate cell_array;
```


Blocks

- Zusammenfassen von Befehlen
- Lokaler Deklarationsteil
- ```
  b0 : block
     signal s : bit;
  begin
     s <= a and b;
     o <= s xor c;
  end block;
  ```
- Blocks können Interfaces enthalten
- Blocks können geschachtelt werden

Prozesse
  ```

Prozesse

- Gehören zu den "concurrent statements"
  - können in einer Architecture definiert werden
  - können nicht in einem Prozess definiert werden
- Anweisungen des Prozesses werden sequentiell ausgeführt
- Sind alle Befehle abgearbeitet, startet die Auswertung von neuem

Beispiel

```
architecture rtl of mux is
begin

p : process(a, b, sel)
begin
   if (sel = '0') then
      z <= a;
   else
      z <= b;
   end if;
end process p;

end rtl;
```


Definition

- process_statement <=
  [label:]
  process [(sensitivity_list)]
  process_declarative_part
  begin
  { sequential_statement }
  end process [label] ;
- Wird am Ende ein label verwendet, muss es mit dem am Anfang übereinstimmen.

Semantik

- Enthält der Prozess eine **Sensitivitätsliste**, so wird er gestartet, wenn sich eines der Signale ändert. Er darf dann keine wait-Anweisung enthalten.
- Ansonsten wird er bis zur nächsten wait-Anweisung ausgeführt. Wir das Ende erreicht, wird er sofort wieder gestartet.

Sensitivity vs. Wait

- Prozesse ***ohne*** Sensitivity List ***sollten*** mindestens eine wait-Anweisung enthalten.
- Prozesse ***mit*** Sensitivity List ***dürfen*** keine wait-Anweisungen enthalten.
  - Die Sensitivity List wird als implizites
    wait on sensitivity_list ;
    am Ende des Prozesses modelliert.

Prozesse und Signale

- In Prozessen sind nur einfache Signalzuweisungen erlaubt
- Zuweisungen auf ein Signal werden zu einem Schreibbefehl zusammengefasst
- Die Werte werden erst beim nächsten wait zugewiesen, d.h. sie werden frühestens im nächsten Simulationszyklus sichtbar (delta delays)

exakte Semantik:
siehe LRM §9.2.1

Prozesse und Signale (2)

```
p : process(q)
begin
   if q = '1' then
      q <= '0';
   end if;
   if q = '0' then
      q <= '1';
   end if;
end process p;
```

werden nie im selben Durchlauf ausgeführt

Aufgabe

- Es soll eine "encoder"-Schaltung angegeben werden, die einen 4-Bit-Eingang "sel" in einen 16-Bit-Wert "y" konvertiert. In "y" sollen alle Bits den Wert '1' haben, nur der durch "sel" angegebene soll '0' sein.
- ```
  entity encoder is
  port(sel : in std_logic_vector(3 downto 0);
       y : out std_logic_vector(15 downto 0));
  end encoder;
  ```
  ```

Lösung 1

```
architecture behave of encoder is
begin
   with sel select
   y <= "1111111111111110" when "0000",
   y <= "1111111111111101" when "0001",
   y <= "1111111111111011" when "0010",
   y <= "1111111111110111" when "0011",
   y <= "1111111111101111" when "0100",
   y <= "1111111111011111" when "0101",
   y <= "1111111110111111" when "0110",
   y <= "1111111101111111" when "0111",
   y <= "1111111011111111" when "1000",
   y <= "1111110111111111" when "1001",
   ...
   y <= "0111111111111111" when others;
end encoder;
```


Lösung 2

```
library ieee;
use ieee.std_logic_unsigned.all;

architecture behave of encoder is
begin
  p : process(sel)
  begin
    y <= (y'range => '1');
    y(conv_integer(sel)) <= '0';
  end process;
end encoder;
```

für conv_integer

zweite Anweisung *überschreibt* die erste Anweisung

std_logic_vector konvertieren in integer

Deklarationsteil

- Im Deklarationsteil können definiert werden
  - Typen, Subtypen,
  - Konstanten,
  - Variablen,
  - ... und anderes.

Variablen

- Zuweisungen auf Variablen sind *sofort* sichtbar
- Variablen können nur innerhalb eines sequentiellen Kontextes deklariert und verwendet werden
- Variablen werden beim ersten Prozessaufruf initialisiert und behalten ihren Wert in den folgenden Aufrufen (Unterschied zu Software!)

Variablen (2)

- Beispiele:

```
variable v : integer range 2 to 11;
variable w : integer := 7;
```

- Zuweisungen:

```
v := 3;
w := 2 + 3;
```


Beispiel

-
```
p : process(q)
variable v : bit;
begin
   v := q;
   if v = '1' then
      v := '0';
   end if;
   if v = '0' then
      v := '1';
   end if;
   q <= v;
end process p;
```

- ... ist äquivalent zu `q <= '1';`

Prozesse und Synthese

- In der Regel werden nur Prozesse mit höchstens einer wait-Anweisung unterstützt
- Sensitivitätslisten werden ignoriert, d.h. es wird kombinatorische Logik erzeugt.
  Beispiel:

```
p : process(a)
begin
   o <= a and b;
end process;
```

o wird neu berechnet, wenn sich a ändert

```
q : process(a, b)
begin
   o <= a and b;
end process;
```

o wird immer neu berechnet

Variablen und Synthese

- Viele Synthese-Tools unterstützen es nicht, dass Variablen zu Registern werden können
- Es sollte nie davon ausgegangen werden, dass Variablen ihren Wert über mehrere Zeittakte behalten

Wait Statements

- Sensitivity Clause:
  - **wait on** signal_name { , ... } ;
  - Wartet, bis sich eines der Signale ändert
- Condition Clause:
  - **wait until** boolean_expression;
  - Wartet, bis der Ausdruck von false nach true wechselt
- Timeout Clause
  - **wait for** time_expression;

Beispiele

- `wait until clk = '1';`
  Ist clk schon zu Beginn '1', dann wird gewartet, bis clk auf '0' wechselt und dann weiter, bis es wieder '1' wird.
- `wait;`
  wartet bis in alle Ewigkeit.
- `wait on clk until reset = '0';`
  wartet bis sich clk ändert und gleichzeitig reset '0' ist.