# VHDL

**V**ery High Speed
Integrated Circuit **H**ardware
**D**escription **L**anguage




# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench

Wiederholung

# Design-Entry: Möglichkeiten

| | *grafisch* | *textuell* |
|---|---|---|
| *high-level* | State-Chart (Zustandsgraph) | VHDL, Verilog, System C |
| *low-level* | Schematic Entry (Schaltplan) | ABEL, CUPL PALASM |

- Kriterien:
  - Unterstützung der menschlichen Intuition
  - Effizienz der Darstellung
  - Weiterverarbeitbarkeit durch Computer




# Historisches

≈1980 implementierungsunabh. Dokumentations-sprache für elektronische Systeme (DoD)

1987 Standardisierung als IEEE 1076: „VHDL-87"

dann MIL-STD-454 Dokumentation, Simulation und Verifikation für ASICs

1993 Überarbeitung von IEEE 1076: „VHDL-93"

VHDL-93 ist der aktuelle gültige Standard

Unterschiede zwischen VHDL-87 und VHDL-93 sind bei „konservativer" Programmierung vernachlässigbar

# Implementierbarkeit

- Standard beschreibt Syntax, nicht Implementierung
- Viele VHDL-Konstrukte nicht in HW umsetzbar
- Synthesetools erwarten bestimmten Codierstil
  - Beim Programmieren schon „in HW denken"





# Anwendungen der Sprache

- Spezifikation & Dokumentation (Mensch/Mensch)
- Simulation & Synthese (Maschine/Maschine)
- Design Entry für komplexe digitale HW (Mensch/Maschine)

⇨ Features:
- Modularität & Hierarchie
- unterschiedl. Abstraktionsebenen
- unterschiedliche Datentypen
- vielfältige Operationen
- parallele und sequentielle Abläufe

# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench

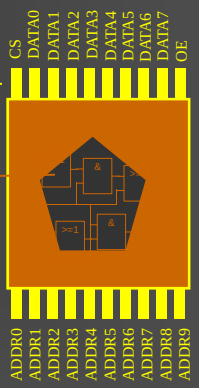



# Grundelemente eines Design

- Aufbau einer Design Unit
  - Entity
  - Architecture
  - Configuration
- Package
  - Package
  - Package Body
- Library

# Struktur einer Design-Unit

- **Entity**
  Schnittstelle zur Umwelt
  (Anschlüsse)
- **Architecture**
  Funktion, Innenleben
- **Configuration**
  Zuordnung Architecture - Entity

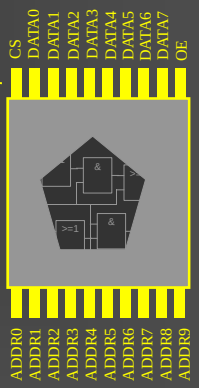



9



# Struktur einer Design-Unit

- **Entity**
  Schnittstelle zur Umwelt
  (Anschlüsse)
- Architecture
  Funktion, Innenleben
- Configuration
  Zuordnung Architecture - Entity



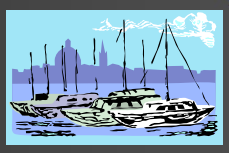

10

# Entity

- beschreibt das Interface (= Anschlüsse) der Design-Unit
- enthält keine Beschreibung der Funktion
- entspricht Schaltsymbol eines Bauteils
- wichtigstes Element sind die Ports
- zusätzlich Deklaration von Parametern möglich („Generics")




# Ports



- … sind beschrieben durch
  - Namen
  - Richtung
  - Datentyp

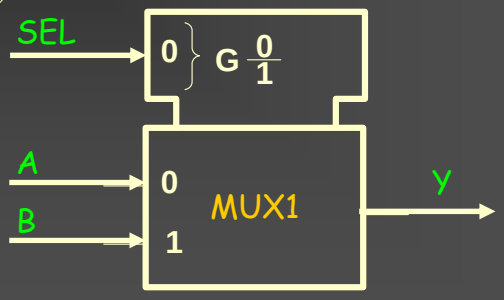
| Richtung | RD | WR | Bemerkung |
|---|---|---|---|
| IN | ja | nein | |
| OUT | nein | ja | |
| INOUT | ja | ja | |
| BUFFER | ja | ja | nur von einer Quelle beschreibbar |

# Entity - ein Beispiel

```
entity MUX1 is
port (A,B,SEL: in std_logic;
      Y: out std_logic);
end MUX1;
```

Name der Entity

Port-Name
Richtung
Datentyp







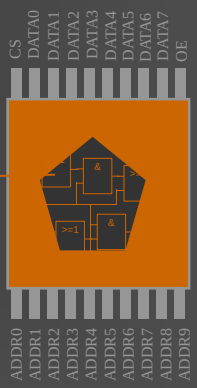
# Entity - noch ein Beispiel

```
entity MY_HALFADD is
port (A,B: in std_logic;
      SUM,CARRY: out std_logic);
end MY_HALFADD;
```

# Struktur einer Design-Unit

- ✓ Entity
  Schnittstelle zur Umwelt
  (Anschlüsse)
- Architecture
  Funktion, Innenleben

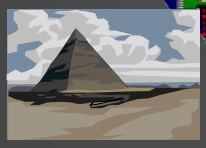
  Configuration
  Zuordnung Architecture - Entity





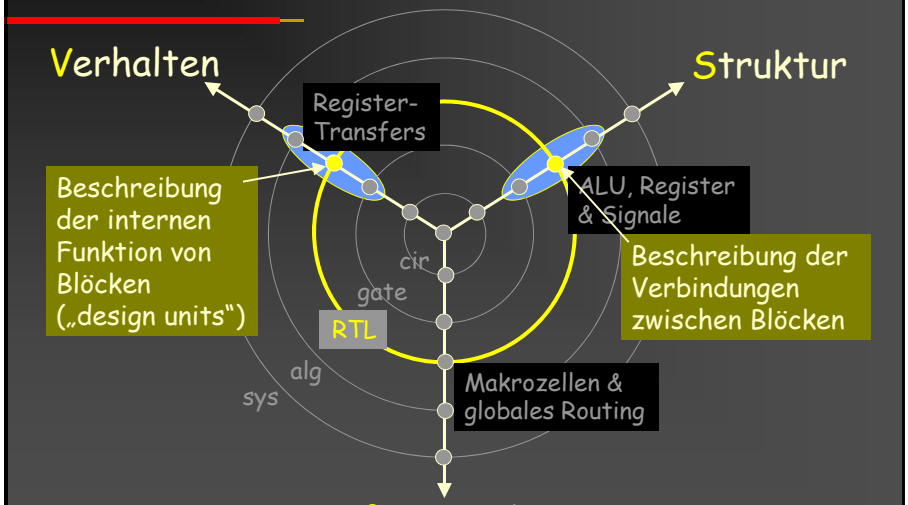


# Architecture



- beschreibt die Funktion einer Design-Unit;
- dies kann auf unterschiedl. Abstraktions-ebenen erfolgen (auch gemischt),
- und zwar entweder hierarchisch durch Instanzierung bestehender Design-Units.
- und/oder durch eine Verhaltens-beschreibung in Form paralleler und/oder sequenzieller Abläufe.

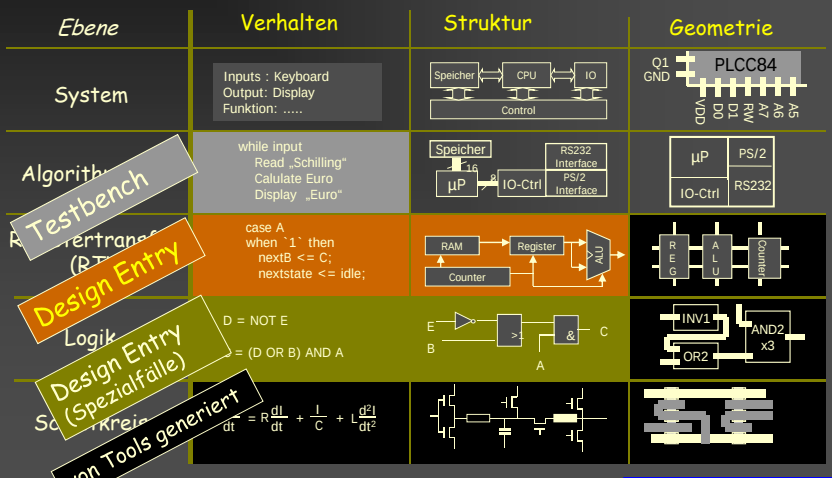
# VHDL-Entry im Y-Diagramm








# Beschreibung in VHDL



# Architecture - RTL/Verhalten

Name der Architecture

Name der Entity

```
architecture MUX_RTL of MUX1 is
begin -- MUX_RTL
  slct: process(A, B, SEL)
  begin
    if SEL ='0' then
      Y <= A;
    else
      Y <= B;
    end if;
  end process slct;
end MUX_RTL;
```

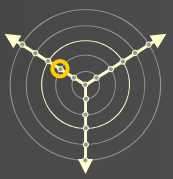




# Architecture - Logik/Verh.

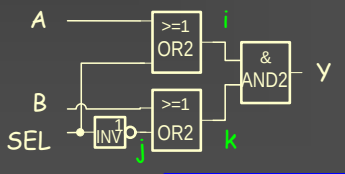
```
architecture MUX_equn of MUX1 is
begin -- MUX_behav
  Y <= (SEL or A) and (not(SEL) or B);
end MUX_equn;
```

$Y = (SEL \vee A) \wedge (\neg SEL \vee B)$



# Architecture – Logik/Strukt.

```
architecture MUX_struc of MUX1 is
  -- need component declarations here
  signal i,j,k: std_logic;
  begin -- MUX_struc
    u1: OR2      port map(A,SEL,i);
    u2: INV      port map(SEL,j);
    u3: OR2      port map(j,B,k);
    u4: AND2     port map(i,k,Y);
  end MUX_struc;
```

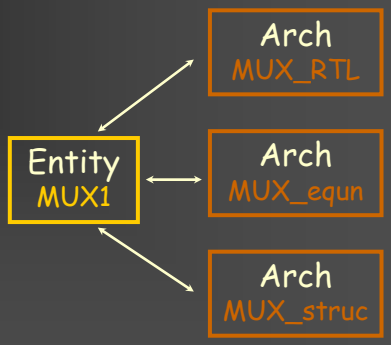






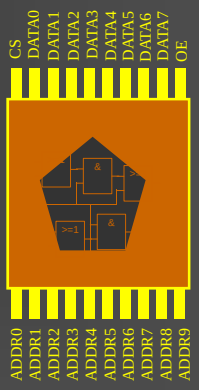

# Architecture – Varianten

- Entity ist fix
- zugeh. Architecture kann variieren:
  - Abstraktionsebenen
    - Simulationsaufwand
    - „Reife" des Designs
    - Testbench
    - IP-Cores
  - Implementierungsvarianten



# Struktur einer Design-Unit

- ✓ Entity
  Schnittstelle zur Umwelt
  (Anschlüsse)
- Architecture
  Funktion, Innenleben
- Configuration
  Zuordnung Architecture - Entity







# Configuration

Zuordnung einer bestimmten Architecture (MUX_RTL) zu Entity MUX1 :

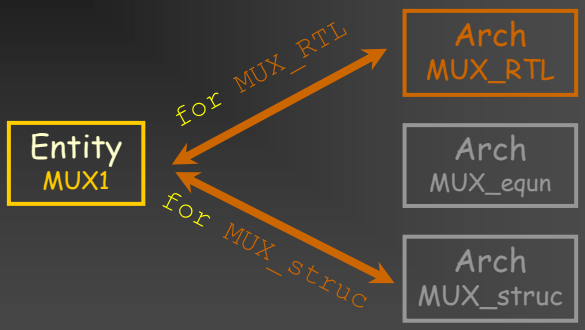
Name der Configuration

```
Configuration CFG_A of MUX1 is
  for MUX_RTL
  end for;
end CFG_A;
```

Name der Entity

Name der aktuell zugewiesenen Architecture

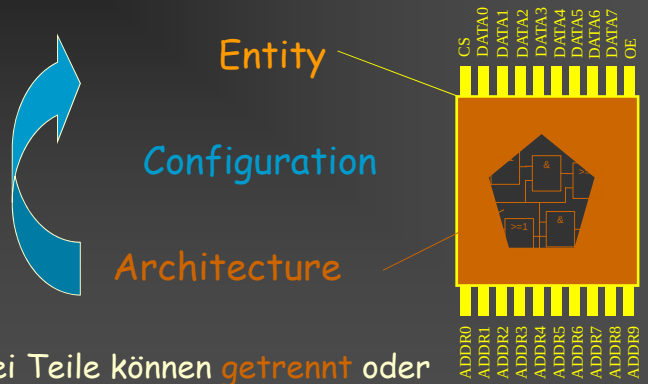
# Configuration




unterschiedliche Realisierungen einer Funktion können parallel existieren und einfach ausgetauscht werden




# Struktur einer Design Unit




Diese drei Teile können getrennt oder in einer gemeinsamen Datei abgelegt werden

# Grundelemente eines Design

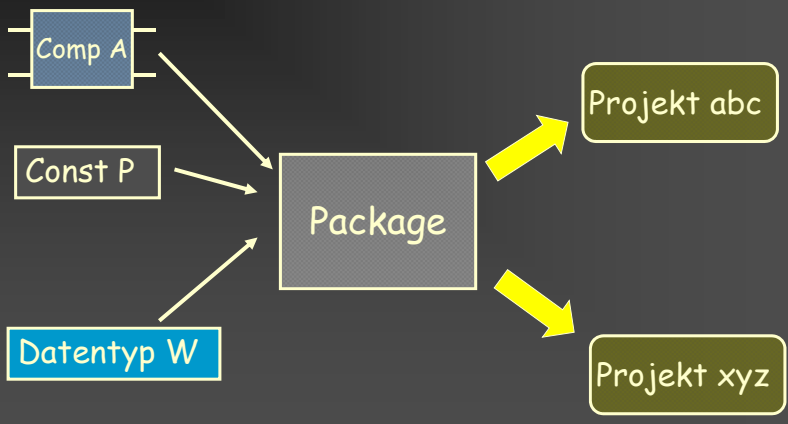
- ✓ Aufbau einer Design Unit
  - Entity
  - Architecture
  - Configuration
- Package
  - Package
  - Package Body
- Library




# Package

- Zentrale Stelle wo globale Deklarationen & Definitionen zu finden sind (vgl. include-File)
  - Konstante
  - User defined Types
  - Component Declarations
  - Sub-Programs
- Kann von Design-Units aus unterschiedlichen Projekten referenziert werden
- Ermöglicht konsistente Vorgehensweise in Design-Teams

# Package: Anwendung





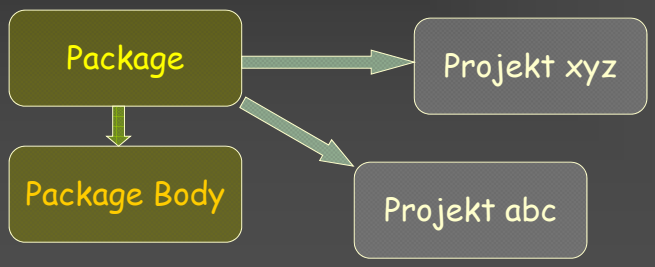


# Package – ein Beispiel

```
package my_constants is

-- define levels to be active high
constant ACTIVE: std_logic  := '1';

-- define width of counter
constant WIDTH : integer    := 8;

-- define maximum count
constant MAXCNT:
  std_logic_vector(WIDTH-1 downto 0)
  := (others => '1');
end my_constants;
```

# Package vs. Package Body

- Package: Schnittstelle für Funktionen, Typen, Komponenten, ... (vgl. *include*.h -Datei in C)
- Package Body: Funktionskörper, Wertzuweisungen für Konstanten, ... (vgl. *include*.c -Datei in C)







# Grundelemente eines Design

- ✓ Aufbau einer Design Unit
  - Entity
  - Architecture
  - Configuration
- ✓ Package
  - Package
  - Package Body
- Library

# Einbinden von Libraries

- Mit der `library` Anweisung wird eine Bibliothek sichtbar gemacht:

```
library <libname>
```

- Standardmäßig legt der Compiler alle compilierten Design-Units in der Library **work** ab.
- Die `library` Anweisung entspricht somit einer Pfadangabe.
- Die Library **work** ist per default sichtbar und muss nicht explizit angegeben werden.




# Zugriff auf Library-Elemente

- Mit der `use` Anweisung werden Elemente aus (sichtbaren!) Libraries bekannt gemacht:

```
use work.my_pack.HA
```

*library name* *package name* *element name*

- Die Angabe von `.all` als element name macht alle Elemente der Library bzw. des Packages bekannt.

# Standard Library (STD)

- Festgelegt durch IEEE-1076
- Enthält die Packages standard und textio
- Package standard (automatisch sichtbar)
  - Grundlegende Datentypen
  - Operatoren
- Package textio (nicht autom. sichtbar)
  - Dateitypen
  - Ein-/ Ausgabeoperatoren




# Package IEEE.std_logic_1164

- Typen
  - std_logic, std_ulogic, ....
- Operatoren
  - and, or, not, ....
- Konvertierungsfunktionen:
  - z.B.: To_bit, To_stdlogic, ...
- Funktionen
  - rising_edge, falling_edge, ...

# Package IEEE.std_logic_arith

- Typen
  - signed, unsigned
- Operatoren
  - +, -, *, /, ABS, ....
  - <, >, <=, ...
- Konvertierungsfunktionen

  CONV_INTEGER(arg: unsigned) return INTEGER

  ....




# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench

# Signale und Datentypen

- Signale dienen der Kommunikation im Design. Sie sind vergleichbar mit Leitungen in einer Schaltung.
- Der Datentyp eines Signals beschreibt die Menge von Werten die das Signal annehmen kann.
- Bei Signalzuweisungen und Operationen müssen die Datentypen zusammenpassen.
- Die Deklaration des Datentyps erfolgt beim Port

```
port (A,B,SEL: in  std_logic; ...);
```

- …oder bei der Signaldeklaration

```
signal x: std_logic;
```




# Vordefinierte Datentypen

| | |
|---|---|
| Boolean | {true, false} |
| Bit | {'0', '1'} |
| Bit_vector | "00100011" |
| Character | 'a', 'z', 'E', '0', '9' |
| String | "I like VHDL" |
| Integer | 27, -237, 0, 12 |
| Real | -11.23, 7.01E-4, 221.09 |
| Time | 23.5 ns, 17 ps |

# Der Datentyp "Enumerated"

- Die Menge der zulässigen Werte ist vom Anwender selbst definierbar

```
type my_state is (reset, idle, rd, wr);
signal state: my_state;
state <= "00";
state <= "reset";
```

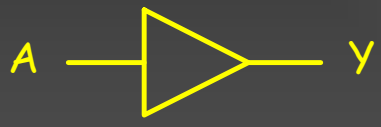
- Nur die definierten Werte dürfen zugewiesen werden




# Signalzuweisung

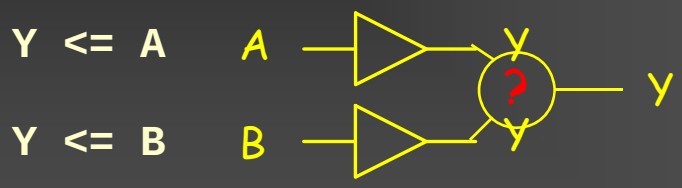
**Y <= A**

Entspricht einem Treiber auf Y



# Mehrfache Signalzuweisung

- Eine mehrfache Signalzuweisung auf das gleiche Signal bedeutet dessen Ansteuerung durch mehrere Treiber:




- Für die Simulation ist in diesem Fall eine Signal Resolution nötig.




# 9-wertige Logik (IEEE Std 1164-1993)

| | | |
|---|---|---|
| 0 | strong low | *Treiberausgang, definiert* |
| 1 | strong high | *"* |
| L | weak low | *Pull-down* |
| H | weak high | *Pull-up* |
| X | strong unknown | *Treiberausgang, undef´d.* |
| W | weak unknown | *bus-keeper, uninitialisiert* |
| Z | high impedance | *Treiberausgang, tri-state* |
| - | don't care | *Pegel bedeutungslos* |
| U | uninitialized | *FF-Ausgang, uninitialisiert* |

# Der Datentyp "std_logic"

- Die 9-wertige Logik nach IEEE 1164 ist im Datentyp Standard Logic (`std_logic`) implementiert (Package IEEE.std_logic_1164).
- Standard Logic bietet eine Signal-Resolution. Diese ermöglicht – im Gegensatz zu anderen Signaltypen wie z.B. Bit oder Integer – eine dem realen Verhalten der elektrischen Signale weitgehend entsprechende Simulation.
- Standard Logic ist daher für Signale, die in HW realisiert werden sollen, bevorzugt zu verwenden.




# Arrays (Vektoren)

- Zusammenfassung mehrerer Objekte des gleichen Typs
  - bit ➡ bit_vector
  - std_logic ➡ std_logic_vector
  - character ➡ string
- aufsteigender oder fallender Index

```
signal regA: bit_vector(1 to 3);
signal addr: bit_vector(2 downto 0);
```

# Arrays: „Concatenation"

- Zusammenfügen von Vektoren (oder Teilen davon)

```
signal addrL: bit_vector(7 downto 0);
signal addrH: bit_vector(4 downto 0);
signal Waddr: bit_vector(12 downto 0);

Waddr <= addrH & addrL(4 downto 1) & '0'
```




# Arrays: Zuweisung

- Zuweisung erfolgt positionsweise

```
signal addr: bit_vector(2 downto 0);
signal regA: bit_vector(1 to 3);

                    regA(1) <= addr(2);
regA <= addr;       regA(2) <= addr(1);
                    regA(3) <= addr(0);
```

# Elementweise Zuweisung

- Für die Zuweisung von Elementen in ein Array gibt es vielfältige Möglichkeiten:

```
signal X: bit_vector(0 to 3);
signal A: bit;
X <= (0=>'1', 2=>A, others =>'0');
X <= ('1' & '0' & A & '0');
X <= ('1','0',A,'0');
X(0 to 1) <= („10“); X(2) <= A;
```

unabh. von der Größe des Array !




# Operatoren

- Logische
  **AND, OR, NAND, NOR, NOT, XOR**
- Relationale
  **>, <, >=, <=, =, /=**
- Schiebeoperatoren
  **sll, srl, sla, sra, rol, ror**
- Arithmetische
  **+, -, *, /, **, abs, mod, rem**

# Logische Operatoren

**NOT, AND, OR, NAND, NOR, XOR, XNOR**

- Definiert für
  - bit
  - boolean
  - std_logic
  - std_ulogic

```
Y <= (SEL or A) and (not(SEL) or B);
```

- Bei Vektor Anwendung bitweise für jeweils entsprechende Elemente




# Relationale Operatoren

**>, <, >=, <=, =, /=**

- Liefern als Ergebnis Datentyp Boolean
- Anwendung meist für if / then /else:

```
if SEL = '0' then
```

- Bit-Vektoren haben keine numerische Bedeutung:
  - es wird von links kommend bitweise verglichen
  - "1001" ist kleiner als "111" !

# Arithmetische Operatoren

**+, -, *, /, ** [Exponent],**
**abs, mod, rem** [Rest]

Definiert für
- integer
- real (nicht mod, rem)
- time

```
Y <= A * abs(B-C) + (F**2);
```

- Im allgemeinen müssen Operanden (und Ergebnis) gleichen Typ haben




# Kleiner Unterschied …

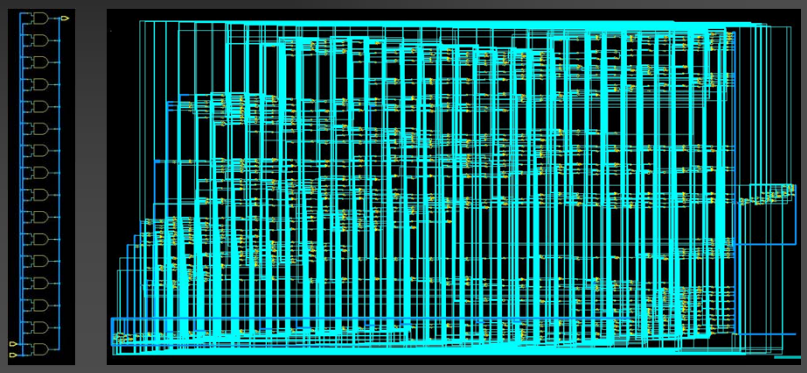
```
entity OPERATION is
  port(a,b: in std_logic_vector(0 to 15);
       y:  out std_logic_vector(0 to 15));
end OPERATION;
```

```
architecture F_NAND of
   OPERATION is
begin
  y <=  not (a and b);
end F_NAND;
```

```
architecture F_MULT of
   OPERATION is
begin
  y <=  a * b;
end F_MULT;
```

# … große Wirkung

NAND: 16GE MULT: ca. 2800 GE



55



# Attribute

- Feldbezogene Attribute (Beispiele)
  - a‘left, a‘right
  - a‘low, a‘high
  - a‘range
- Signalbezogene Attribute (Beispiele)
  - s‘event
  - s‘stable

56

# Sequentielle Anweisungen

- nur innerhalb Process zulässig
  - Wait Anweisung
  - If-Then-Else Anweisung
  - Case Anweisung
  - Loop Anweisung
  - Assert Anweisung




# Wait-Anweisungen

nicht synthetisierbar

- Wait for <time>

```
wait for 100 ms;
```

- Wait on <event on signal from list>

```
wait on A, B, C;
```

- Wait until <condition>

```
wait until trig = '1';
```

- Wait [forever]

```
wait;
```

# If-Then-Else Anweisung

Bedingung

```
if A < 0 then              --  A < 0
  B <= 0;
elsif A < 3 then           --  0 ≤ A < 3
  B <= 1;
elsif A > 2 then           --  A ≥ 3
  B <= 2;
else                       --  ??
  B <= 3;
end if;
```

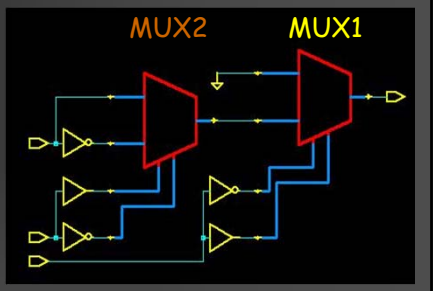
sequentielle Statements

- Bei `elsif` wird der erste gültige Zweig gewählt
  => Reihenfolge wichtig



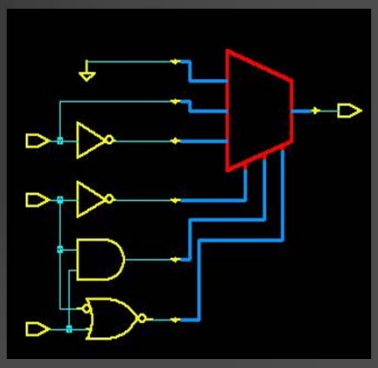

# If-Then-Else Beispiel 1

```
process(in1,in2,in3)
begin  -- process
  if in1 = '0' then
    out1 <= '0';
  else
    if in2 = '1' then
      out1 <= in3;
    else
      out1 <= not in3;
    end if;
  end if;
end process;
```



# If-Then-Else Beispiel 2

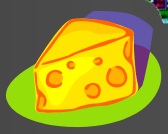
```
process (in1, in2, in3)
 begin  -- process
   if in1 = '0' then
     out1 <= '0';
   elsif in2 = '1' then
     out1 <= in3;
   else
     out1 <= not in3;
   end if;
 end process;
```



61



# Case-Anweisung



```
case (a + b) is
  when 0 =>
    y <= 1;
  when 1 to 3 =>
    y <= 2;
  when 4 | 6 =>
    y <= 3;
  when others =>
    y <= 4;
end case;
```

"Objekt"

sequ. Statemt.

Bereich

Liste

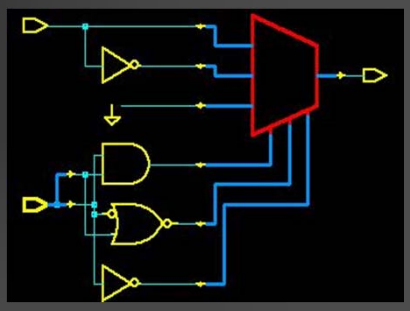
Rest (obligat)

Jeder mögliche Wert des Objekts muss exakt einmal vorkommen

62 A

# Case Beispiel

```
process (sel, in3)
  begin  -- process
    case (sel) is
      when "11"   =>
  out1 <= in3;
      when "10"   =>
  out1 <= not in3;
      when others =>
  out1 <= '0';
    end case;
end process;
```







# Loop-Anweisung

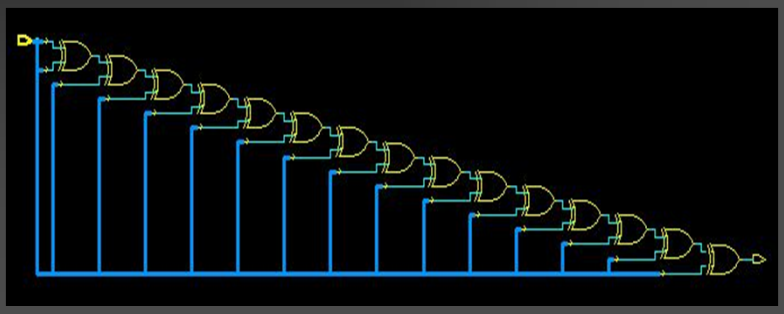
```
process (in_vec)
  variable parity:std_logic:='0';
begin  -- process
  parity:='0';
  for i in 15 downto 0 loop
    parity:=parity xor in_vec(I);
  end loop;  -- i
  out1 <= parity;
end process;
```

Iterations-vorschrift

sequenzielle Anweisung(en)

# Loop: Syntheseergebnis

- Die sequenzielle Anweisung in der Schleife wird mehrfach ausgeführt und generiert daher mehrfach Hardware







# Assert-Anweisung

- zur Überprüfung von Bedingungen während der Simulation

```
assert a /= 0
   report „Dumm gelaufen“
   severity error
```

Bedingung, wenn „false“:

Ausgabestring (optional)

Severity level [note, warning, error, failure]

Abbruchlevel Simulator-abhängig

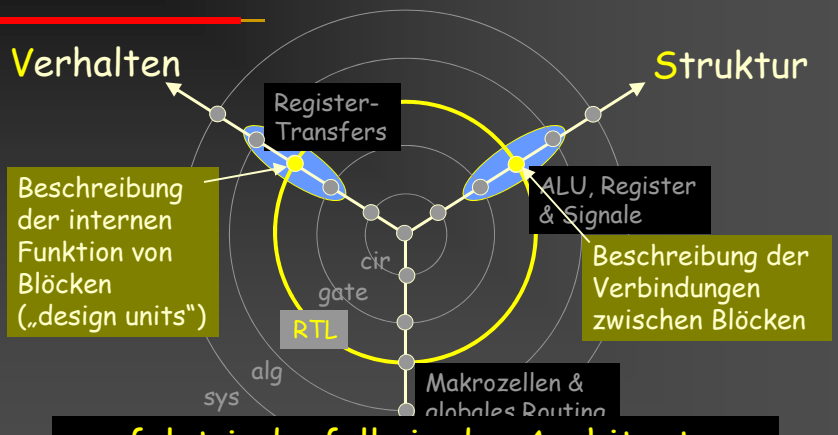
# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench




Wiederholung

# Modellierung in VHDL




…erfolgt jedenfalls in der Architecture

Geometrie

# Strukturmodellierung

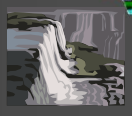
- Deklaration:
  - Welche Komponenten sollen ins … eingefügt werden (Anschlüs…
- Instanzierung:
  - Komponenten wer… …, auch mehrfach, w… …interne Funktion wird nich… …rachtet (black box)
- Portm…
  - …n zwischen den Komponenten (vgl. B… …) werden hergestellt.

Diese Art der Modellierung ermöglicht ein hierarchisches Design!

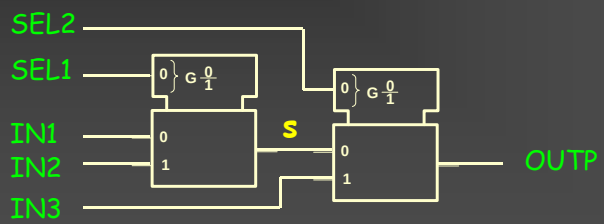




# Beispiel MUX-Kaskade



Für die folgende Schaltung ist durch geeignete Instanzierung der Component MUX1 eine strukturelle Darstellung anzugeben.



# Komponentendeklaration

```
entity MUX1 is
  port (A,B,SEL: in  std_logic;
                Y: out std_logic);
end MUX1;
```

Alternativ: Deklaration im Package

```
architecture DEMO of MUX_CASCADE is
  …
  component MUX1
    port (A,B,SEL: in  std_logic;
                  Y: out std_logic);
  end component;
  …
end DEMO
```





# Instanzierung & Port Map

```
architecture DEMO of MUX_CASCADE is
  signal s: std_logic;
  component MUX1
    port (A,B,SEL: in  std_logic;
                  Y: out std_logic);
  end component;
begin -- DEMO
  m1: MUX1 port map(IN1,IN2,SEL1,s);
  m2: MUX1 port map(s,IN3,SEL2,OUTP);
end DEMO;
```

✓

# Signaldeklaration

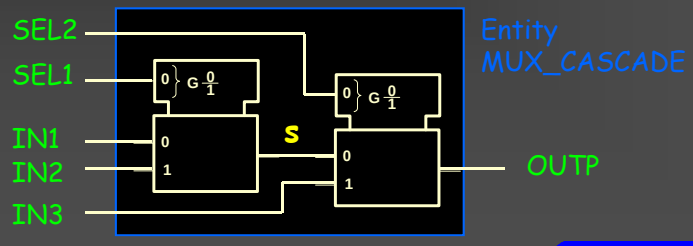
```
architecture DEMO of MUX_CASCADE is
  signal s: std_logic;
  component MUX1
    port (A,B,SEL: in  std_logic;
                Y: out std_logic);
  end component;
begin -- DEMO
  m1: MUX1 port map(IN1,IN2,SEL1,s);
  m2: MUX1 port map(s,IN3,SEL2,OUTP);
end DEMO;
```





# Lokale Signale

- Signale an den Anschlüssen sind in der Entity deklariert => in der Architecture keine Deklaration nötig
- Lokale (=interne) Signale müssen in der Architecture zu Beginn explizit deklariert werden.



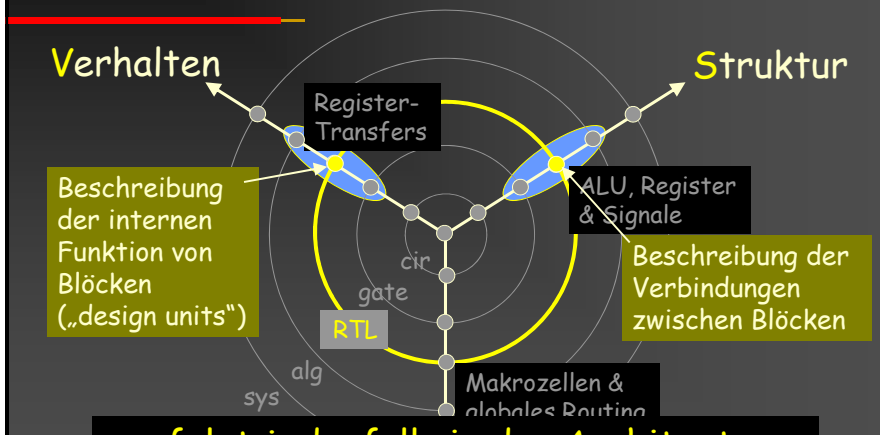
# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench




Wiederholung

# Modellierung in VHDL



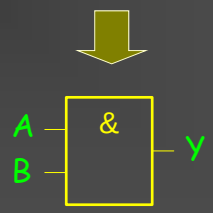

…erfolgt jedenfalls in der Architecture

Geometrie

# Abbildung eines VHDL-Befehls

- Jeder Befehl wird bei der Synthese auf entsprechende Hard-ware abgebildet.
- Diese Hardware arbeitet permanent.
- Ein Befehl wird daher nicht nur einmal abgearbeitet, sondern alle Befehle sind ständig aktiv (vgl. HW)

```
architecture X of F_AND is
begin
  Y <= A and B;
end X;
```





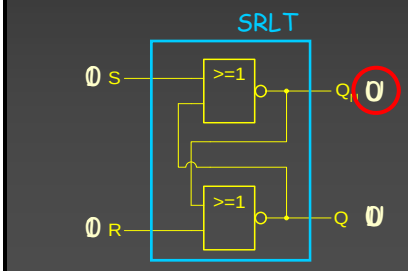


# Concurrency

- VHDL-Statements beschreiben Hardware-Funktionseinheiten und sind in der Lage, alle gleichzeitig zu arbeiten.
- VHDL-Anweisungen werden daher grundsätzlich parallel ("concurrent") ausgeführt.
- Im Gegensatz dazu werden in der Software normalerweise alle Befehle nacheinander ("sequentiell") abgearbeitet.
- Ein Simulator führt daher in einem Zeitschritt zunächst alle parallelen Anweisungen aus („Delta-Time"), bevor er die Zeit inkrementiert.

# Beispiel zur Concurrency

```
architecture X of SRLT is
begin
  QN <= not(S or Q);
  Q <= not(R or QN);
end X;
```




- Funktion in HW:
  - Statements beschreiben die Struktur
  - Reihenfolge irrelevant
  - Beide Zuweisungen werden ständig und parallel vorgenommen
- Bei sequentieller Bearbeitung wie in SW:
  - Reihenfolge ist wichtig !
  - Anderes Verhalten !




# Concurrency: weitere Beispiele

| | |
|---|---|
| `Y <= Y + 1;` | SW: Incrementer (Y in Register!)<br>HW: Timing Loop / Overflow |
| `Y <= not(Y);` | SW: Inversion<br>HW: Timing Loop / Oszillator |
| `Y <= 0;`<br>`Y <= B + C;` | SW: Initialisierung von Y<br>HW: Treiberkonflikt |
| `Y <= A + B;`<br>`A <= C + D;` | SW: Y wird mit altem Wert von A berechnet<br>HW: 2 Addierer in Kaskade; Y verwendet aktuelles A |

# Sequenzielle Abläufe

- Sequenzielle Abläufe werden in VHDL als Process formuliert.
- Manche Statements sind grundsätzlich nur innerhalb eines Process zulässig:
  - IF / THEN / ELSE
  - CASE
- Ein Simulator führt die Anweisungen im Process der Reihe nach durch.

```
<label>: process(<sens>)
begin
  statement 1
      . . .
  statement n
end process <label>
```




# Signalzuweisungen im Process

- Die Auswertung der Signalzuweisungen erfolgt ausschließlich am Ende des Process.
- Es sind daher auch mehrfache Zuweisungen auf das selbe Signal zulässig, ohne dass sich Treiberkonflikte ergeben.
- Zur Wirkung kommt nur die jeweils letzte Zuweisung auf ein Signal, alle anderen werden ignoriert (d.h. kommen nicht einmal kurzzeitig zur Wirkung).
- Zuweisungen von verschiedenen Processes auf das selbe Signal führen jedoch zu Treiberkonflikten.

# Process: ein Beispiel

```
architecture MUX_RTL2 of MUX1 is
begin -- MUX_RTL2
  slct: process(A, B, SEL)
  begin
    Y <= B;
    if SEL = '0' then
      Y <= A;
    end if;
  end process slct;
end MUX_RTL2;
```

Sensitivity List

sequentielle Statements

Letzte ausgeführte Zuweisung wird wirksam





# Signalzuweisung: ein Beispiel

Gegeben ist folgender Process, wobei zu Beginn gilt A = 2, B = 5, Y = 9, Z = 0:

```
signal A, B, Y, Z: integer;
assigmt: process(A, B, Y)
begin
  Y <= B;
  Y <= A;
  Z <= Y-1;
end process assigmt;
```

Welche Werte haben Y und Z nach einem Durchlauf des Process?

2 1

# Sensitivity List

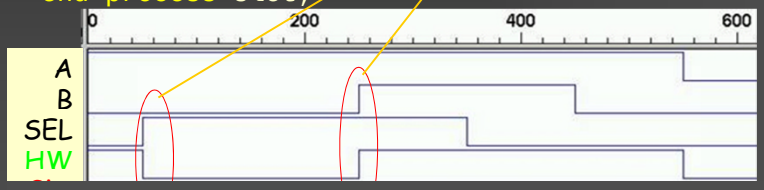
- Processes werden nicht ständig durchlaufen.
- In der Sensitivity List sind jene Signale anzugeben, bei deren Änderung ("Event") der Simulator den Process von neuem durchlaufen soll.
- Dies sind bei kombinatorischer Logik alle Signale, die im Process gelesen werden.
- Die Sensitivity List ist nur für die Simulation relevant und hat keinen Einfluss auf das Ergebnis der Synthese (nur Warning wenn unvollständig).




# Sensitivity List: ein Beispiel

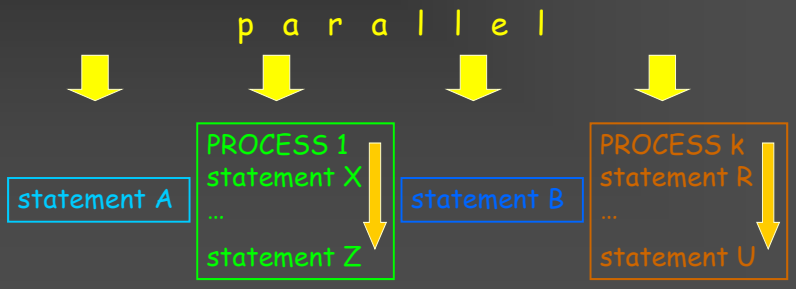
```
slct: process(A, B, SEL)
 begin
   if SEL = '0' then
     Y <= A;
   else
     Y <= B;
   end if;
 end process slct;
```

HW und Simulation verhalten sich unterschiedlich!



# Processes und Concurrency

- Anweisungen im Process: sequentiell
- unterschiedliche Processes und Anweisungen außerhalb von Processes: parallel







# Processes in der Architecture

```
architecture ABSTRACT of EXAMPLE is
begin -- ABSTRACT
  <concurrent statements>;
  P1: process(<sensitivity list>)
  begin
    <sequential statements>;
  end process P1;
  <concurrent statements>;
  P2: process(<sensitivity list>)
  begin
    <sequential statements>;
  end process P2;
  <concurrent statements>;
end ABSTRACT;
```

beliebig viele Processes

beliebig viele concurrent Statements

Reihenfolge irrelevant

# Variable

- Lokale Gültigkeit innerhalb eines Process.
- Nach außen nicht sichtbar.
- Wertzuweisung erfolgt sofort.
- Operator für Zuweisung ist ":=" anstelle von "<=".
- Wert wird zwischen zwei Aufrufen des Process beibehalten.
- Wechselseitige Zuweisung zwischen Variablen und Signalen zulässig, sofern Datentyp übereinstimmt.




# Variable versus Signal

Werte zu Beginn: A = 2, B = 5, Y=9, Z=0

```
signal A,B,Y,Z: integer;
assigmt: process(A,B,Y)

begin
  Y <= B;
  Y <= A;
  Z <= Y-1;
end process assigmt;
```

Werte nach dem Process:

(1) Y = 2, Z = 8
(2) Y = 2, Z = 1

```
signal A,B,Z: integer;
assigmt: process(A,B)
variable Y: integer;
  begin
    Y := B;
    Y := A;
    Z <= Y-1;
  end process assigmt;
```

Y = 2, Z = 1

# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench



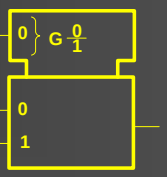

# Codierung eines Multiplexers

```
architecture GOOD_ONE of MUX1 is
begin -- GOOD_ONE
slct: process(A, B, SEL)
  begin
    if SEL ='0' then
      Y <= A;
    else
      Y <= B;
    end if;
  end process slct;
end GOOD_ONE;
```

Bedingung

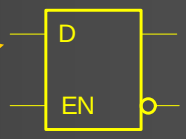
IF-Zweig

ELSE-Zweig



# Codierung eines Latch

```
architecture BAD_ONE of MUX is
begin -- BAD_ONE
slct: process(A, SEL)
  begin
    if SEL ='0' then
      Y <= A;
  -- KEIN ELSE-ZWEIG
    end if;
  end process slct;
end BAD_ONE;
```




Bedingung

d.h. alten Wert halten, wenn Bedingung nicht erfüllt ist.





# Codierung eines Flip-Flop

```
architecture GOOD_ONE of FLIPFLOP is
begin -- GOOD_ONE
sync: process(clk)
  begin
    if clk'event and clk = '1' then
      outp <= data;
    end if;
  end process sync;
end GOOD_ONE;
```

nur clk in der Sensitivity-List !

kein ELSE-Zweig

# Synchr. & Asynchr. Reset

```
if asyn_rst = '0' then
  outp <= '0';
else
  if clk'event and clk = '1' then
    if syn_rst = '0' then
      outp <= '0';
    else
      outp <= data;
    end if;
  end if;
end if;
```

asynchroner Reset

synchr. Reset

Sensitivity List enthält **clk** und **asyn_rst**, nicht jedoch **syn_rst**

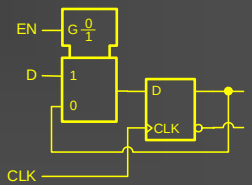




# Clock Enable

```
sync: process(clk)
  begin
    if clk'event and clk = '1' then
      outp <= next_data;
    end if;
  end process sync;
new_data: process(data,enable,outp)
  begin
    if enable = '1' then
      next_data <= data;
    else
      next_data <= outp;
    end if;
  end process new_data;
```

Hilfssignal, wird asynchron ermittelt



# Asynchroner Teiler

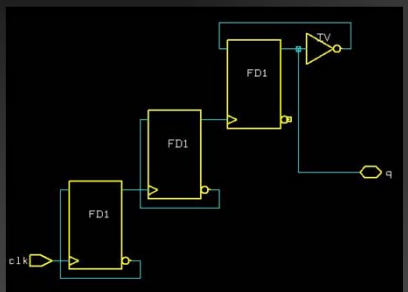
```
entity tff is
  port(clk: in std_logic;
       q: inout std_logic);
end tff;

architecture A of tff is
begin -- Toggle Flip-Flop
work: process(clk)
  begin
    if (clk'event and
        clk = '1') then
      q <= not(q);
    end if;
  end process work;
end A;
```

```
entity takt is
  port(clk: in std_logic;
       q: inout std_logic);
end takt;

architecture B of takt is
signal x,y : std_logic;
-- component declaration tff
begin
tff1: tff
  port map(clk=>clk, q=>x);
tff2: tff
  port map(clk=>x, q=>y);
tff3: tff
  port map(clk=>y, q=>q);
end B;
```

97

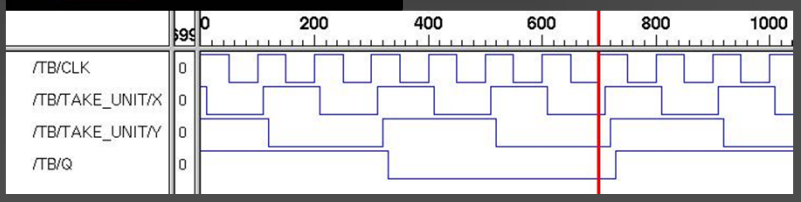


# Asynchroner Teiler – Probleme




Wird Q in ein Takt-netz eingespeist ?

Q ist gegenüber CLK stark verschoben, Delay ist jedoch unbestimmt.




98

# Binärer Zähler: Entity

Zu entwerfen ist ein synchroner 8-Bit Zähler:

```
library IEEE;
use     IEEE.std_logic_1164.all;
use     work.my_constants.all;
entity syn_counter is
  port(
    clk:  in    std_logic;
    cnt:  inout
              std_logic_vector(WIDTH-1 downto 0);
    oflo: out   std_logic;
    rst:  in    std_logic;
     );
end syn_counter;
```

Libs und packages sichtbar machen





# Binärer Zähler: Package

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

package my_constants is

-- define levels to be active high
constant ACTIVE: std_logic  := '1';

-- define width of counter
constant WIDTH : integer    := 8;

-- define maximum count
constant MAXCNT:
      std_logic_vector(WIDTH-1 downto 0)
      := (others => '1');

end my_constants;
```

Lib und packages sichtbar machen

# Binärer Zähler: Architecture 1

```
library IEEE;
use     IEEE.std_logic_1164.all;
use     IEEE.std_logic_UNSIGNED.all;
use     work.my_constants.all;

architecture nice of syn_counter is

signal nextCnt: std_logic_vector(WIDTH-1 downto 0);

begin
-- <process count_syn: synchronous capture of next state>
-- <process value_update: asynchronous next-state-logic>
end nice;
```

Lib und packages sichtbar machen

Signaldeklaration

Hier werden noch 2 Prozesse eingefügt





# Binärer Zähler: Architecture 2

```
count_syn: process(clk, rst)

begin
  if rst = ACTIVE then
    cnt <= (others => '0');
  else
    if clk'event and clk = '1' then
      cnt <= nextCnt;
    end if;
  end if;
end process count_syn;
```

asynchroner Reset

synchron übernehmen

# Binärer Zähler: Architecture 3

```
value_update: process(cnt)

begin
  nextCnt <= cnt + '1';
  oflo    <= not ACTIVE;
  if cnt = MAXCNT then
    nextCnt <= (others => '0');
    oflo    <= ACTIVE;
  end if;

end process value_update;
```

Kein else-Zweig
Aber Default-Zuweisung

Zählen ohne Overflow

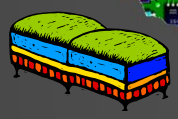
Overflow behandeln




# Überblick

- Historie & Anwendung
- Grundelemente des Design
- Syntax: Signale, Datentypen & Operatoren
- Strukturelle Modellierung & Hierarchie
- Verhaltensmodellierung:
  parallele und sequenzielle Abläufe
- Codierungsbeispiele
- Testbench

# Zweck einer Testbench

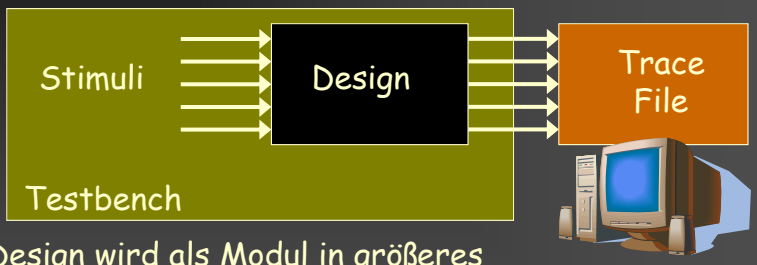


- Jede sinnvolle Hardware muss in ihrer Anwendung mit der Umwelt interagieren (Inputs & Outputs).
- Um ein Design funktional zu testen noch bevor die HW gefertigt ist, muss es in eine (virtuelle) Umgebung eingebettet werden, die den Bedingungen im Betrieb möglichst gleicht.
- Diese Einbettung zu schaffen, ist Aufgabe der Testbench. Sie hat typisch folgende Aufgaben:
  - Generieren des Taktes
  - Anlegen eines Reset und Initialisierung der Eingänge
  - Simulation von Eingangssignalen bzw. Sequenzen davon
  - ggf. Auswertung von Outputs




Wiederholung

# Simulation & Testbench




Design wird als Modul in größeres Design (Testbench) eingebettet und erhält so Stimuli für die Eingänge bei der Simulation.

Die Testbench ist stets auf dem Top-Level, und das zu testende Design ist als Component eingebettet

# Aufbau einer Testbench

- Die Testbench besteht wie jede Design Unit aus Architecture, Entity und Configuration.
- Die Testbench ist hierarchisch auf dem Top Level und hat keine Inputs und Outputs.
- Das zu testende Design wird als Component instanziert und die Signale entsprechend angeschlossen (port map).
- Die Testbench dient nur der Simulation und wird später nicht in Hardware gefertigt. Sie muss daher nicht synthetisierbar sein.




# Reset und Initialisierung

- Reset muss zu Beginn lang genug anliegen
  - mindestens 1 Taktperiode + Einschwingzeiten
  - wirkt bei Simulation und Hardware gleich
- Initialisierung von Eingängen/Signalen
  - direkt z.B. über Maskierung

```
y <= a and reset;
```

  - über Default bei der Deklaration wirkt NUR bei Simulation, später keine Auswirkung auf die Hardware!

```
signal x : std_logic := '0';
```

# Testbench – ein Beispiel (1)

```
library IEEE;
use     IEEE.std_logic_1164.all;
use     IEEE.std_logic_UNSIGNED.all;
use     work.my_constants.all;

entity testbench is
end testbench;
```

Keine Ports => Entity trivial

```
architecture behav of testbench is
  constant clock_period : time := 100 ns;
  signal clock, reset, overflow : std_logic;
  signal count : std_logic_vector(WIDTH-1 downto 0);
```





# Testbench – ein Beispiel (2)

```
component syn_counter is
  port(
    clk:      in    std_logic;
    cnt:      inout std_logic_vector(WIDTH-1 downto 0);
    oflo:     out   std_logic;
    rst:      in    std_logic
      );
end component ;
```

Design als Component deklariert

```
begin
DUT: syn_counter port map (clock,count,overflow,reset);
```

Instanzierung des Design

# Testbench – ein Beispiel (3)

```
ClockGenerator: process
begin
  clock <= '1';
  wait for clock_period/2;
  clock <= '0';
  wait for clock_period/2;
end process ClockGenerator;
```

Takt generieren
(Process läuft endlos !)




# Testbench – ein Beispiel (4)

```
  Waveforms: process
  begin
    reset <= ACTIVE;
    wait for (2.1*clock_period);
    reset <= not ACTIVE;
    -- bei Bedarf hier weitere Zuweisungen auf Signale
    wait for (2**WIDTH + 1) * clock_period;
    assert false
    report "test finished" severity error;
  end process Waveforms;
end behav;
```

Reset aktivieren
für > 2 Taktzyklen

Zähler-Überlauf abwarten

Simulation abbrechen

# Modellierung von Delay

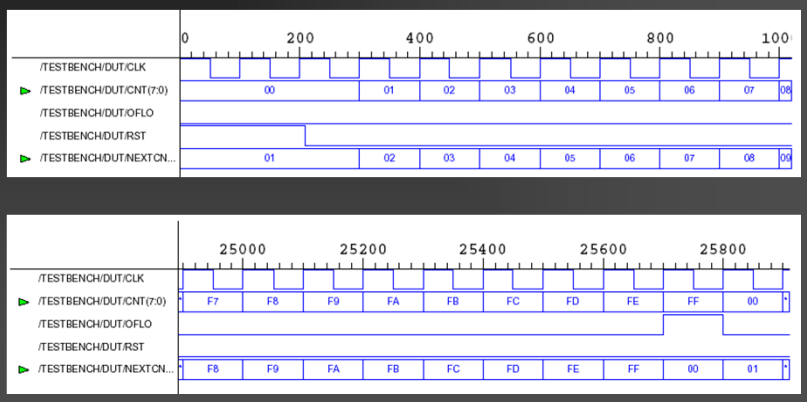
- Zuweisung einer Verzögerung in VHDL

```
Y <= A after 25ns;
```

- Wird vom Simulator für die Behavioral Simulation berücksichtigt
- Ist jedoch nicht synthetisierbar und wird daher bei der Synthese verworfen (Warning)
- Nach der Synthese bzw. dem PPR werden ohnedies Default-Werte bzw. backannotierte Delays verwendet.

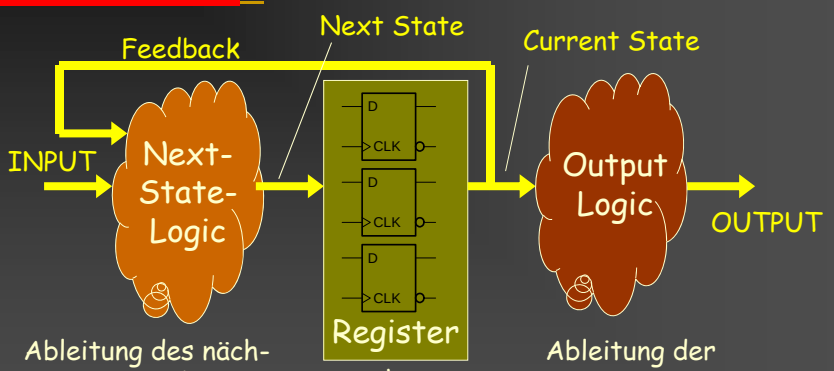



# Testbench - Simulation



Wiederholung

# Moore-State Machine




Ableitung des nächsten Zustands aus aktuellem Zustand und Eingängen

synchroner Zustandswechsel

Ableitung der Ausgänge aus aktuellem Zustand





# State-Machines in VHDL

## Funktionen der FSM

(1) synchroner Zustandswechsel

(2) Berechnung des nächsten Zustandes

(3) Berechnung der Ausgänge

## Prozesse in VHDL

(1) Synchroner Teil

(2) Next-State-Logic (asynchroner Process)

(3) Output Logic (asynchroner Process)

# Ampel: Package

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

package ampel_const is
type state is (Xgrn, Xyel, Xred, Xry);
constant GREEN:  std_logic_vector(0 to 2):="100";
constant YELLOW: std_logic_vector(0 to 2):="010";
constant RED:    std_logic_vector(0 to 2):="001";
constant REDYEL: std_logic_vector(0 to 2):="011";
constant ACTIVE: std_logic:='1';
end ampel_const;
```



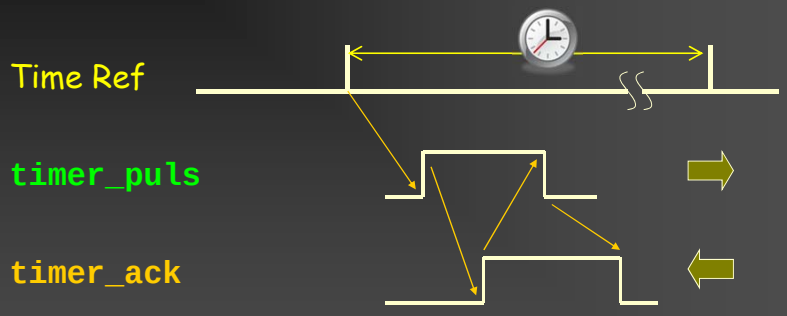


# Ampel: Entity

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

entity ampel is
port(
     clk       : in std_logic;
     reset     : in std_logic;
     timer_puls: in std_logic;
     timer_ack: inout std_logic;
     lights_X:  out std_logic_vector(0 to 2);
     lights_Y:  out std_logic_vector(0 to 2)
     );
end ampel;
```

Handshake mit externem Timer

# Externer Timer: Handshake




Der externe Timer generiert, abgeleitet von einer Zeitreferenz, Pulse **timer_puls** die jeweils bis zur Quittierung durch **timer_ack** aktiv bleiben.




# Ampel: Architecture (1)

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

architecture simple of ampel is
  signal phase, nextPhase: state;
  signal nextTimer_ack: std_logic;

begin

 --< process syn_phase: synchroner Teil >
 --< process next_phase: Next-State-Logic >
 --< process output: Output Logic >

end simple;
```

# Ampel: Architecture (2)

```
syn_phase: process(clk,reset)
  begin
    if reset = ACTIVE then
      phase <= Xyel;
      timer_ack <= not ACTIVE;
    else
      if clk'event and clk = '1' then
        phase <= nextPhase;
        timer_ack <= nextTimer_ack;
      end if;
    end if;
  end process syn_phase;
```





# Ampel: Architecture (3)

```
next_phase: process(timer_puls, phase, timer_ack)
begin
  nextPhase <= phase;
  nextTimer_ack <= not ACTIVE;
  if timer_puls = ACTIVE then
    nextTimer_ack  <= ACTIVE;
    if timer_ack = not ACTIVE then
      case phase is
        when Xyel   => nextPhase <= Xred;
        when Xred   => nextPhase <= Xry;
        when Xry    => nextPhase <= Xgrn;
        when others => nextPhase <= Xyel;
      end case;
    end if;
  end if;
end process next_phase;
```

Default Assignments

# Ampel: Architecture (4)

```
output: process(phase)
begin
  case phase is
    when Xgrn   =>  lights_X <= GREEN;
                    lights_Y <= RED;
    when Xred   =>  lights_X <= RED;
                    lights_Y <= GREEN;
    when Xry    =>  lights_X <= REDYEL;
                    lights_Y <= YELLOW;
    when others =>  lights_X <= YELLOW;
                    lights_Y <= REDYEL;
  end case;
end process output;
```




# Zusammenfassung (1)

- VHDL ist eine standardisierte Sprache, die für die Beschreibung von Hardware verwendet wird.
- Der Standard bezieht sich auf die Syntax. Viele syntaktisch richtige Konstrukte sind zwar simulierbar jedoch nicht in Hardware implementierbar.
- Ein Design wird in VHDL beschrieben durch
  - Entity (Anschlüsse)
  - Architecture (Funktionsbeschreibung) und
  - Configuration (Zuordnung Entity/Architecture)

# Zusammenfassung (2)

- Mittels eines Package können Konstante, Funktionen etc. global definiert werden. Ein Package muss vor seiner Verwendung sichtbar gemacht werden.
- Über Libraries können vordefinierte Packages eingebunden werden, wie z.B. std_logic_1164.
- Die Funktionsbeschreibung in der Architecture kann entweder in der Struktur- oder in der Verhaltensdomäne erfolgen.
- Die Möglichkeit in der Struktubeschriebung, bestehende Entities als Components einzubinden, erlaubt ein hierarchisches Design.




# Zusammenfassung (3)

- Die Kommunikation innerhalb des Designs erfolgt über Signale.
- Jedes Signal muss bei der Deklaration ein Signaltyp zugewiesen werden, der die Menge jener Werte beschreibt, die das Signal annehmen darf.
- Neben den im Standard vordefinierten Datentypen gibt es auch den Typ „enumerated".

# Zusammenfassung (4)

- Ein wesentliches Konzept in VHDL ist die Concurrency: Im Normalfall laufen alle Anweisungen parallel ab.
- Einen sequentiellen Ablauf von Anweisungen erreicht man nur innerhalb eines Process.
- Im Process erfolgt die Zuweisung von Signalen erst ganz am Ende, die von Variablen jedoch sofort.
- Die Sensitivity-List des Process führt jene Signale an, bei deren Änderung der Process im Simulator erneut durchlaufen wird.




# Zusammenfassung (4)

- Die Testbench schafft eine virtuelle Umgebung, in die das Design für den Test eingebettet wird. Sie wird ebenfalls in VHDL formuliert, braucht jedoch nicht synthetisierbar zu sein.