



## 2. Praktikumstermin

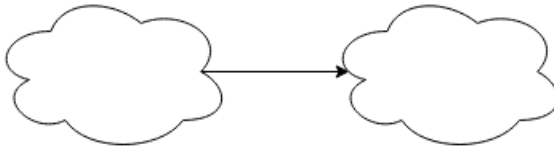
Rechnerorganisation Praktikum WiSe23/24 | Architektur eingebetteter Systeme |  
Neunwertige Logic, Hierarchien, Instanziierung, Signale & Arrays

---



## Neunwertige Logik

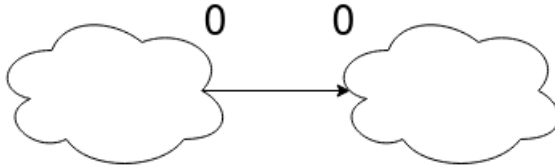
- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).





## Neunwertige Logik

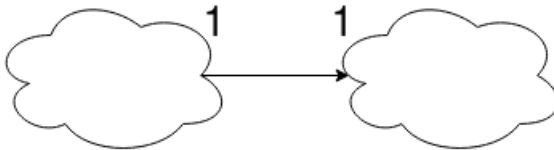
- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).





## Neunwertige Logik

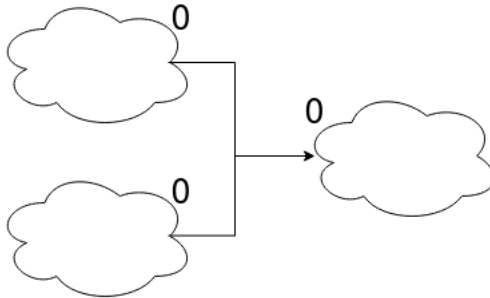
- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).





## Neunwertige Logik

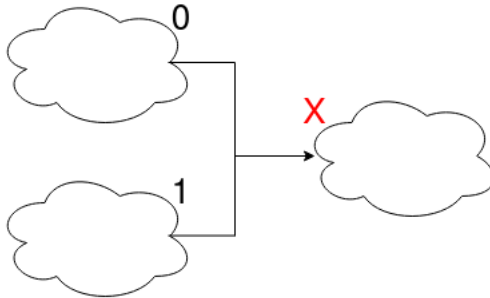
- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).





## Neunwertige Logik

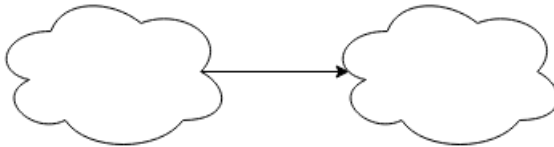
- Basis von **Zweiwertige Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik), mit dieser Abstraktion: Signalen können sich in Zwei Zustände befinden: 1 (Spannung) oder 0 (keine Spannung).





## Neunwertige Logik

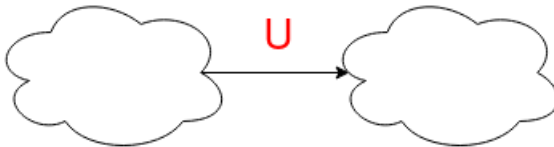
- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).



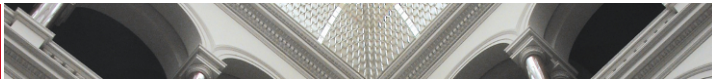


## : Neunwertige Logik

- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).

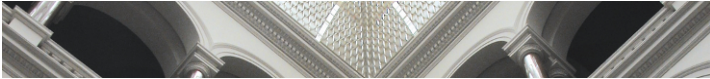




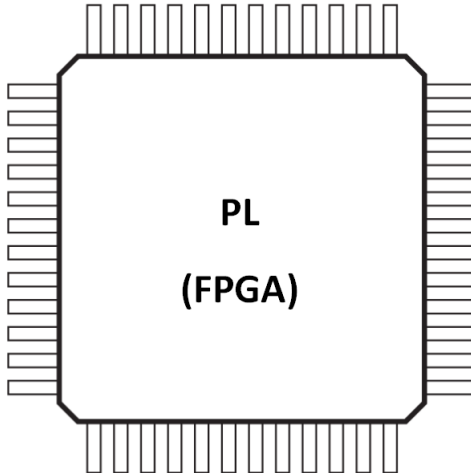


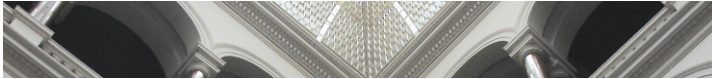
## : Neunwertige Logik

- Basis von **zweiwertiger Logik** sind Funktionen (Aussagen), welche entweder *wahr* oder *falsch* sind.
- Wir bauen digitalen Schaltungen (Digitaltechnik) mit dieser Abstraktion: Signalen können sich in zwei Zuständen befinden: 1 (Spannung) oder 0 (keine Spannung).
- Um elektrisches Verhalten exakter nachbilden zu können, arbeitet die neunwertige Logik (auch **multivalued logic**, MVL9) nicht nur mit den beiden Werten 0 und 1
- Sie enthält auch: **Unknown (X)**, **Uninitialized (U)**, High-Impedance (Z), Weak-Werte (L, H, W), und Don't-care (-).
- Hilfreich beim Debugging!

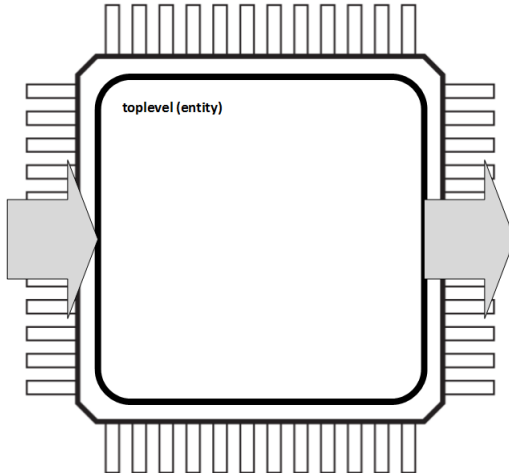


## Hierarchisches Design



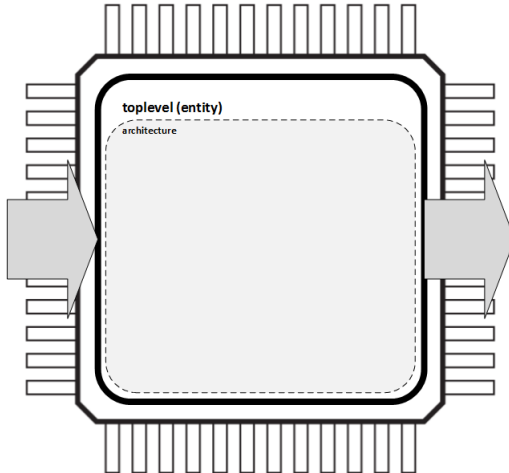


## Hierarchisches Design



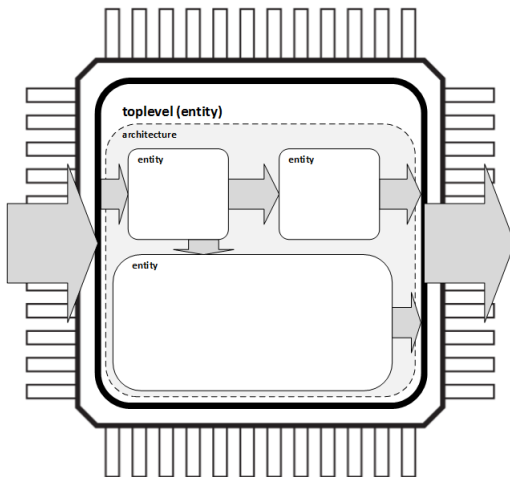


## Hierarchisches Design



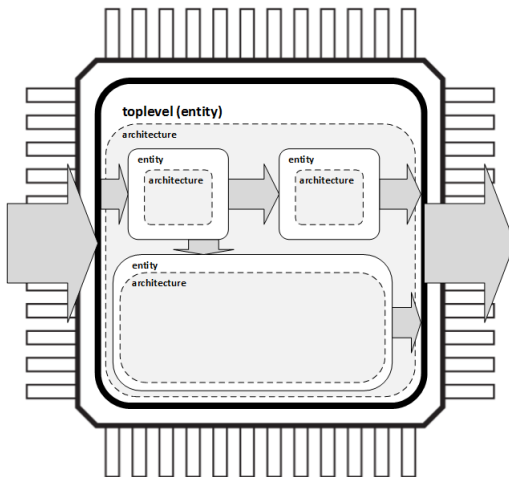


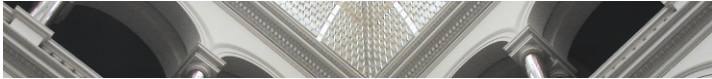
## Hierarchisches Design



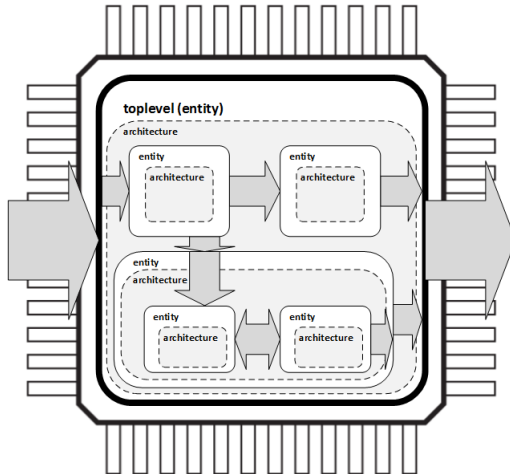


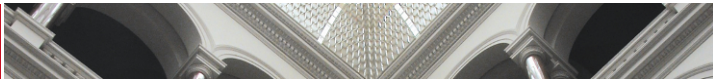
## Hierarchisches Design





## Hierarchisches Design





## Instanziierung Syntax

```
<instance> : entity <library>.<component>(<architecture>)  
  [ generic map (...) ]  
  port map( ... );
```

- **library**: Design-Bibliothek, die die Compiler-Ergebnisse aufnimmt
  - Verzeichnis mit dem Bibliotheksnamen im Arbeitsverzeichnis
  - *GHDL* verwendet als Standard-Bibliothek „work“
- **component**: zu instanziiierende Komponente
  - Muss mit exakt diesem Namen im Projektverzeichnis existieren!
- **architecture**: zu verwendene Implementierung
  - Zu einer *entity* können mehrere *architectures* gehören.
  - Muss mit exakt diesem Namen im Projektverzeichnis existieren!





## Instanziierung Beispiel

```
entity or3 is
  port (i1, i2, i3 : in std_logic;
        o1 : out std_logic);
end entity or3;

architecture netlist of or3 is
  signal s1 : std_logic;           -- Signalerzeugung
begin
  or_1: entity work.or2(behavior)  -- Instanziierung
  port map ( a => i1,              -- Signalzuweisung
            b => i2,
            y => s1);

  or_2: entity work.or2(behavior)  -- Instanziierung
  port map ( a => s1,              -- Signalzuweisung
            b => i3,
            y => o1);
end architecture netlist;
```

## Wichtig:

- *eindeutige* Bezeichner für Instanzen innerhalb einer architecture
- Erzeugung interner Signale mittels `signal` (vgl. nächste Folie)



## Signale

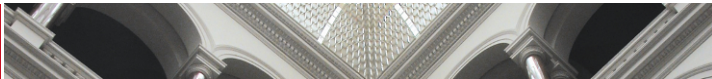
# Deklaration und Zuweisung

In VHDL unterscheidet man zwischen

- **externen Signalen:** symbolisieren die Verbindungen einer Entity nach außen, Definition durch port-Statements
- **internen Signalen:** symbolisieren die Verbindungen zwischen den Einzelteilen einer Entity, Definition wie folgt:

```
architecture <architecture_name> of <entity_name> is
...
-- Signaldeklaration
signal X, Y : std_logic := '0'; -- Interne Signale X und Y sind vom Typ std_logic
...                               -- und haben den Standardwert 0
begin
...
-- Signalzuweisung
X <= '1'; -- dem Signal X wird der Wert '1' zugewiesen
Y <= 'Z'; -- dem Signal Y wird der Wert 'Z' (Tristate) zugewiesen
...
end architecture;
```

- Zuweisung eines Standardwertes bei der Deklaration durch :=



Implementationsebenen

## Verhalten vs. Struktur

### Verhaltensebene (behavioral modelling)

- sequentielle Anweisungen (innerhalb eines Prozesses)  
VHDL-Schlüsselwörter: *process, if-else-elsif, case-when, function... , procedure ...*
- nebenläufige Anweisungen  
VHDL-Schlüsselwörter: *when-else, with-select, ... or, and ...*

### Strukturebene (structural modelling)

- Ein „sauberes“ hierarchisches Design deklariert auf dieser Ebene nur die Signale und Komponenten.
- Somit erstreckt sich die Modellierung ausschließlich auf das Verdrahten vorhandener Komponenten über die Entity.  
VHDL-Schlüsselwörter: *component, signal, ...*



Implementationsebenen

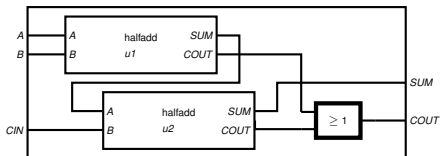
## Verhaltens- und Strukturbeschreibung

### Verhaltensbeschreibung

eines Halbaddierers auf Logikebene

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

architecture gatelevel of halfadd is
begin
    SUM  <= A xor B;
    COUT <= A and B;
end architecture gatelevel;
```

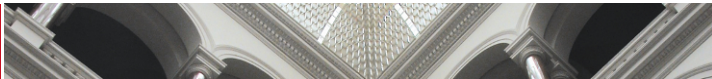


### Strukturelle Beschreibung

eines Volladdierers mit zwei Halbaddierern

```
entity fulladd is
    port (A, B, CIN : in  std_logic;
          SUM, COUT : out std_logic);
end entity fulladd;

architecture structural of fulladd is
    signal S1, S2, S3: std_logic;
begin
    u1 : entity work.halfadd(gatelevel)
        port map(A => A,
                 B => B,
                 SUM => S1,
                 COUT => S2);
    u2 : entity work.halfadd(gatelevel)
        port map(A => S1,
                 B => CIN,
                 SUM => SUM,
                 COUT => S3);
    u3 : entity work.or2(logic)
        port map(A => S2,
                 B => S3,
                 Y => COUT);
end architecture structural;
```



## Arrays

### Was ist das?

- Arrays sind Aufreihungen von Daten desselben Datentyps

```
type type_name is array (range) of element_type;           -- Syntax
```

- Array definieren (im *Deklarationsteil*):

```
type mv19 is array (0 to 8) of std_logic;                 -- Beispiel 1  
signal ex1 : mv19;
```

```
type byte is array (7 downto 0) of std_logic;             -- Beispiel 2  
signal ex2 : byte;
```

- Array belegen (im *Anweisungsteil*):

```
-- Gemeinsam ...  
ex1 <= ('0', '1', 'L', 'H', 'X', 'W', 'Z', '-', 'U');
```

```
-- Einzelne ...  
ex2(7) <= '1'; ex2(6) <= '0'; ex2(5) <= '0'; ex2(4) <= '0';  
ex2(3) <= '0'; ex2(2) <= '0'; ex2(1) <= '0'; ex2(0) <= '0';
```

```
-- Oder kurz ...  
ex2 <= ( 7 => '1', others => '0');
```



## Arrays

### std\_logic\_vector

- in `ieee.std_logic_1164` enthaltener Datentyp
- Arrays von beliebig vielen `std_logic`-Werten
- logische Operatoren (`and`, `or`, etc.) auch für Vektoren definiert

```
architecture <architecture_name> of <entity_name> is
...
    signal s1, s2, s3 : std_logic_vector(3 downto 0);
...
begin
...
    s1 <= "1100";           -- s1 = "1100"
    s2(0) <= '0';           -- 0-te Stelle von s2 zuweisen
    s2(2) <= '1';           -- 2-te Stelle von s2 zuweisen
    s2(1) <= '1';           -- 1-te Stelle von s2 zuweisen
    s2(3) <= '0';           -- 3-te Stelle von s2 zuweisen
    s3 <= s1 and s2;         -- s3 = "0100"
...
end architecture;
```

- **wichtig:** Werte zuweisen: `std_logic` in `' '`, `std_logic_vector` in `" "`



## Konvertierung

# std\_logic\_vector to integer

- **wichtig:** Bibliothek einbinden

```
library IEEE;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all;      -- Neu!
```

- Array-Indizes sind Ganzzahlen (Integer)
- Deshalb ist folgende Typumwandlung u.U. nützlich:

```
architecture foo of bar is  
...  
    signal int : integer;  
    signal slv : std_logic_vector(3 downto 0);  
...  
begin  
...  
    int <= to_integer(unsigned(slv));  
...  
end architecture;
```

- siehe auch: „cast / convert numeric\_std“ im ISIS-Kurs



## Kontrollstrukturen...

- ... fassen mehrere Anweisungen in einer zusammen
- ... können demnach situationsabhängig „entscheiden“
- Es existieren verschiedene Kontrollstrukturen
- Wir demonstrieren ein Beispiel im Folgenden anhand der Entity:

```
entity bcd is
  port(bcd      : in  std_logic_vector(3 downto 0);
        bitmask : out std_logic_vector(6 downto 0));
end entity bcd;
```





## Kontrollstrukturen

# Kontrollstrukturen

## Conditional assignment:

```
architecture when_example of bcd is
begin
    output <= "0111111" when bcd = "0000" else
               "0000110" when bcd = "0001" else
               -- ...
               "1000000";
end;
```

## With-select:

```
architecture with_example of bcd is
begin
    with bcd select
        bitmask <= "0111111" when "0000",
                   "0000110" when "0001",
                   -- ...
                   "1000000" when others;
end architecture with_example;
```