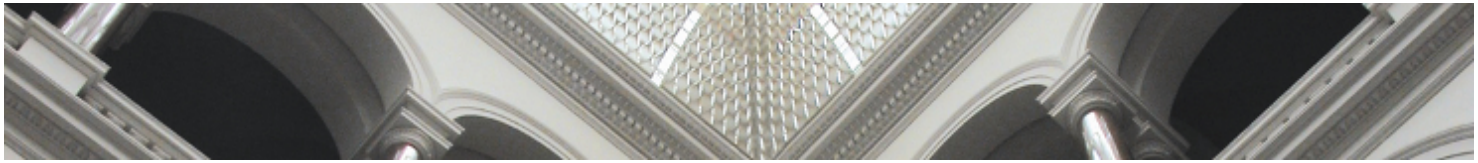
Technische Universität Berlin

9. Praktikumstermin

Rechnerorganisation Praktikum WiSe23/24 | Architektur eingebetteter Systeme | Testbenches

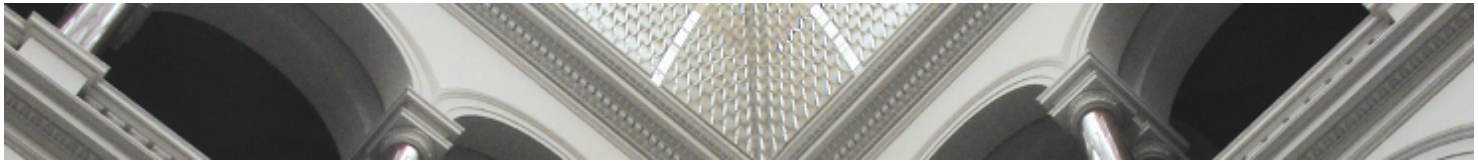


Technische Universität Berlin

Wozu Testbenches?

Eine Testbench (TB) . . .

- . . . belegt Eingänge einer Schaltung mit verschiedenen Werten
- . . . prüft resultierende Ausgangsbelegung über Vergleichswerte
- . . . gibt auftretende Unterschiede (Fehler) an den Nutzer zurück
- . . . und stellt so sicher, dass ein Modul wie gewünscht arbeitet



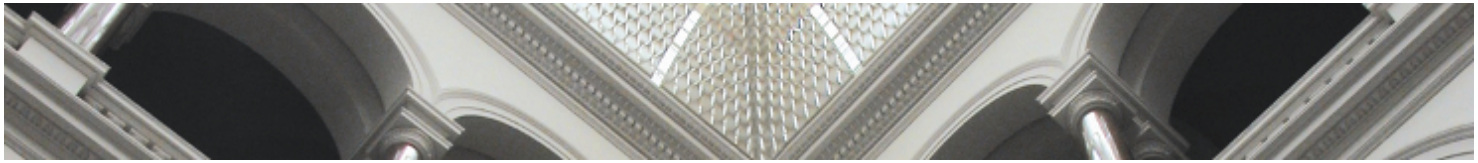
Technische Universität Berlin

Testbenches - Aufbau (1)

Entity

- TBs brauchen keine Schnittstellen nach außen
- Ihre `entity`-Beschreibung ist folglich leer, sofern keine Generics verwendet werden

```
entity testbench is
  -- (optionale Generics)
  -- keine Ports
end testbench;
```



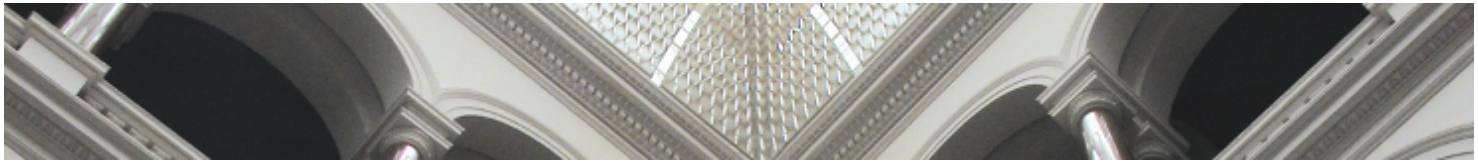
Technische Universität Berlin

Testbenches - Aufbau (2)

Architecture

- TBs müssen anders als normale Module nicht synthetisierbar sein
- In der `architecture` funktioniert also alles wie bekannt (und mehr!)

```
architecture testbench is
  -- Deklarationsteil
begin
  -- Anweisungsteil
end testbench;
```



Technische Universität Berlin

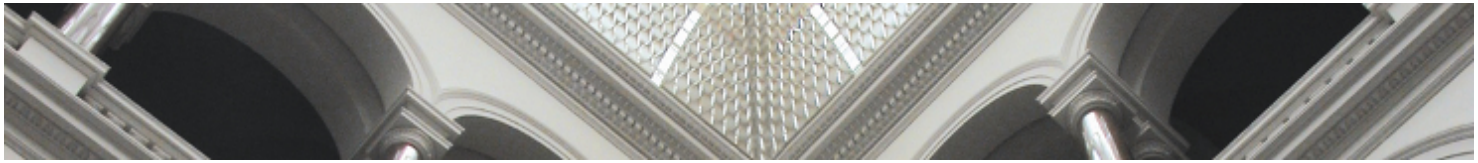
Testen - Schritt für Schritt (1)

Schritt 1: DUT instanziieren

- DUT („device under test“) = zu testendes Modul
- Zunächst muss das DUT instanziiert werden, z.B. so:

```
-- im Anweisungsteil der architecture
dut: entity work.mux2(behavioral)
     port map(a => s1,
              b => s2,
              s => s3,
              y => s4);
```

- **Achtung:** Dazu muss es bereits kompiliert vorliegen!



Technische Universität Berlin

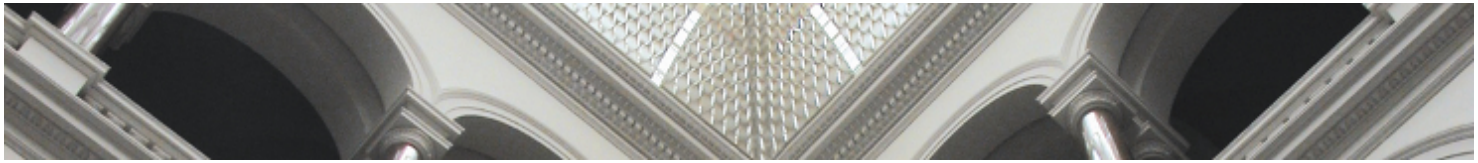
Testen - Schritt für Schritt (2)

Schritt 2: Testprozess anlegen

- Komplexere Module werden meist in einem Prozess getestet
- So können *Schleifen*, *if-Abfragen* und *Variablen* genutzt werden
- Der dabei verwendete Prozess besitzt *keine* Sensitivitätsliste

```
-- im Anweisungsteil der architecture
test: process
  -- Deklarationsteil
begin
  -- Anweisungsteil
  wait;
end process;
```

- Er wird dann permanent, wie eine Endlosschleife, durchlaufen
- Um das zu verhindern, steht am Ende in der Regel ein `wait`



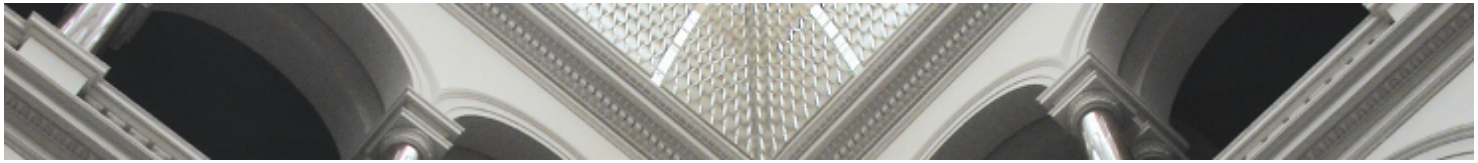
Technische Universität Berlin

Testen - Schritt für Schritt (3)

Schritt 3: Testdaten anlegen

- Verschiedene Testdaten am DUT anlegen und Ausgänge prüfen
  - Testdaten können bspw. in Form eines Arrays vorliegen
  - und in einer Schleife nacheinander angelegt werden
  - Die Überprüfung ist dann bloß ein simpler Vergleich
- **Wichtig:** Zwischen Anlegen und Prüfen kurz warten

```
-- im Anweisungsteil des process

-- Testdaten anlegen

wait for 5 ns;

-- Ausgänge prüfen
```



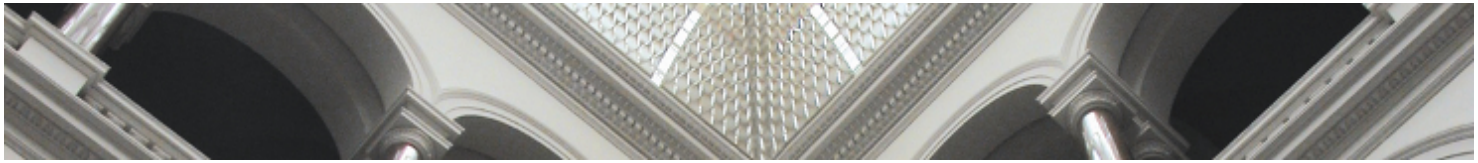
Technische Universität Berlin

Testen - Schritt für Schritt (4)

Schritt 4: Fehler ausgeben

- Ergibt die Überprüfung Unterschiede, sollte die TB diese ausgeben
- Hierbei ist die `textio`-Bibliothek hilfreich, dazu aber später mehr

- Außerdem praktisch: Anzahl der Fehler in einer Variable speichern
- So können dann auch nur die ersten *n* Fehler angezeigt werden

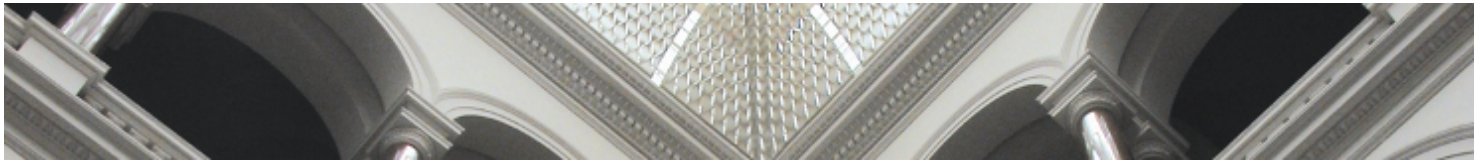


Technische Universität Berlin

Testen - Schritt für Schritt (5)

Schritt 5: Testergebnis ausgeben

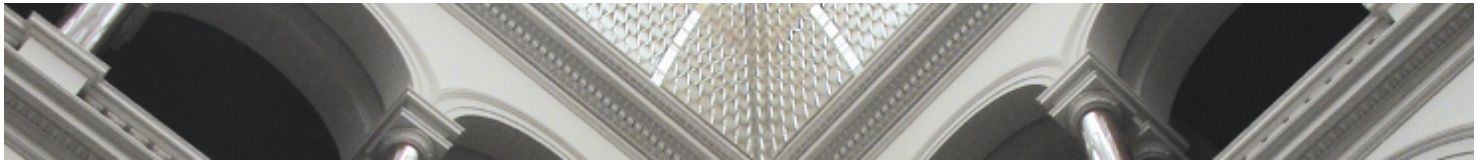
- Eine abschließende Auswertung rundet die Testbench ab
- Die Auswertung kann aus der Anzahl der Fehler resultieren
- Sie sollte das Ergebnis in einem Satz eindeutig zusammenfassen



Technische Universität Berlin

Testen - Zusammenfassung

```
entity testbench is
  -- (optionale Generics)
  -- keine Ports
end testbench;

architecture testbench is
  -- Deklarationsteil
  -- Hier Signale definieren
begin

  -- DUT instanziieren

  test: process
    -- Deklarationsteil
    -- Hier Variablen definieren
    -- z.B. einen Fehler-Zähler
  begin
    -- Phase 1: üblicherweise in einer Schleife:
    -- verschiedene Testdaten anlegen & überprüfen

    -- Phase 2: Abschließende Ausgabe des Testergebnisses

    wait; -- Wichtig! Sonst: Endlosschleife
  end process;
end testbench;
```



Technische Universität Berlin

I/O: Grundlegendes

image-Attribut

- mit VHDL 1993 eingeführt, um Textausgaben zu vereinfachen:

```
T'image(X)
```

- Dabei ist `X` der auszugebene Ausdruck vom Typ `T`, z.B.:

```
integer'image(error_count)
std_logic'image(one_bit_value)
```

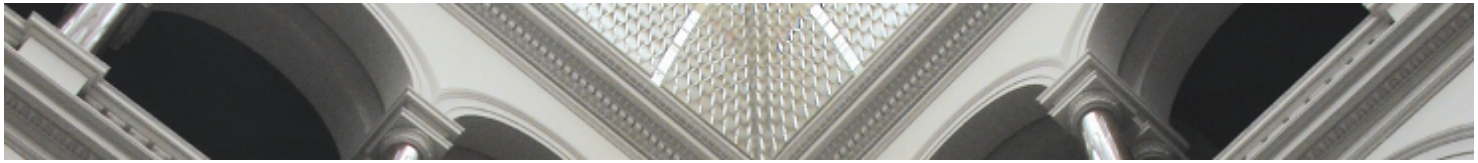
- **Achtung:** `std_logic_vector'image` funktioniert *nicht*!

Konkatenation

- In VHDL ist das Kaufmanns-Und `&` der Konkatenations-Operator
- So lassen sich mehrere Werte zu einem einzigen zusammenfügen:

```
A : std_logic_vector(3 downto 0) := "1111";
B : std_logic_vector(3 downto 0) := "1111";
C : std_logic_vector(7 downto 0);
C <= A & B;
```

- Dies funktioniert auch mit mehreren Strings



Technische Universität Berlin

I/O: Report-Statement

– Für einfache String-Rückgaben existiert das `report`-Statement

```
[assert <condition>] report <string> [severity <level>];
```

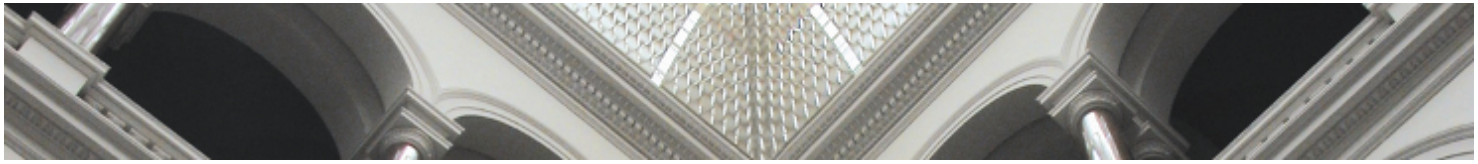
– Die Teile in eckigen Klammern sind optional

– Der String muss in Anführungszeichen stehen

– Mögliche `severity` Level:

- `note` (Hinweis)
- `warning` (Warnung)
- `error` (Fehler, Standard-Level)
- `failure` (Führt zum Abbruch der Simulation)

– Der `report` wird angezeigt, falls die `assert`-Bedingung *nicht* gilt!



Technische Universität Berlin

I/O: Report-Statement - Beispiel

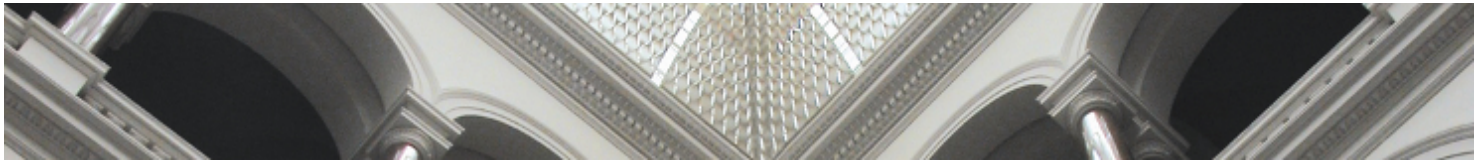
– Für das nachstehende Beispiel . . .

```
variable error_count : integer := 0;
-- ...
if error_count > 0 then
  report "Das DUT funktioniert nicht einwandfrei! Es gab "
         & integer'image(error_count) & " Fehler." severity failure;
end if;
```

– . . . ergäbe sich u.U. folgende Ausgabe:

```
# ** Failure: Das DUT funktioniert nicht einwandfrei! Es gab 3 Fehler.
#    Time: 100 ns  Iteration: 0  Process: /ram_tb/testbench File: ram_tb.vhd
```

– Durch das erzwungene Format ist diese aber etwas hässlich . . .



Technische Universität Berlin

I/O: textio-Bibliothek

– . . . Abhilfe schafft hier die `textio`-Bibliothek:

```
library std;
use std.textio.all;
```

– Diese ermöglicht u.a. das Einlesen und Schreiben externer Dateien

– bietet aber auch Möglichkeiten zum Schreiben nach `STDOUT`

– Dazu deklarieren wir eine neue Variable vom Typ `line`:

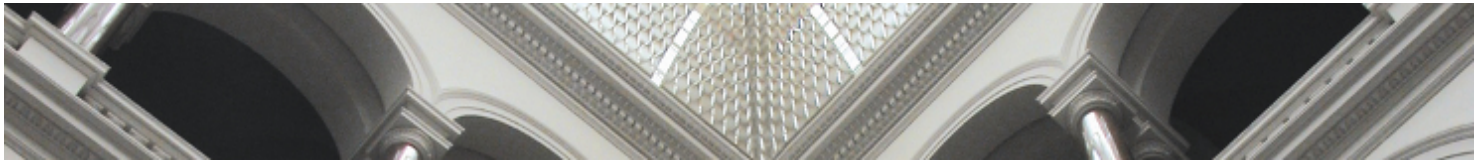
```
variable l : line;
```

– Schreiben über diesen dann die gewünschte Ausgabe:

```
write(l, <Ausgabe>)
```

– Und bestätigen am Ende die Ausgabe nach `STDOUT` mit:

```
writeline(OUTPUT, l)
```



Technische Universität Berlin

I/O: textio-Bibliothek - Beispiel

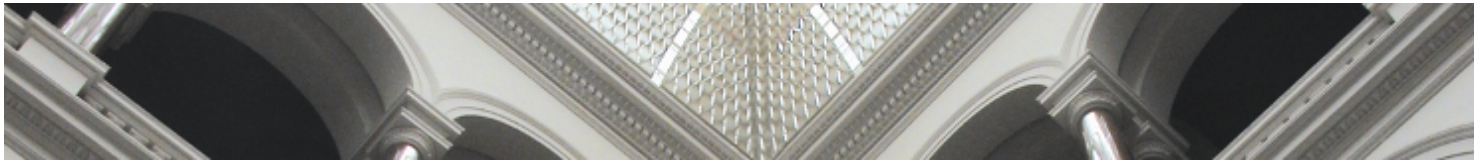
– Für das nachstehende Beispiel . . .

```
library std;
use std.textio.all;
-- ...
variable errour_count : integer := 0;
variable l: line;
-- ...
if error_count > 0 then
  write(l, time'image(now));
  write(l, string'("Das DUT funktioniert nicht einwandfrei! Es gab "));
  write(l, integer'image(error_count));
  write(l, string'(" Fehler."));
  writeline(OUTPUT, l)
end if;
```

– . . . ergäbe sich u.U. folgende Ausgabe:

```
# 100000 ps: Das DUT funktioniert nicht einwandfrei! Es gab 3 Fehler.
```

– Das ist schon deutlich schöner, aber . . .



Technische Universität Berlin

I/O: std_logic_textio

- . . . wie geben wir nun einen `std_logic_vector` aus?
- Wir könnten alle Bits einzeln über `std_logic'image` ausgeben
- Einfacher macht es uns die `std_logic_textio`-Bibliothek:

```
library ieee;
use ieee.std_logic_textio.all;
```

- Diese überlagert einen Teil der `textio`-Funktionen
- sodass nun auch folgendes möglich ist:

```
signal slv : std_logic_vector(3 downto 0) := "1010";
-- ...
variable l: line;
-- ...
write(l, slv);
```

- Ebenfalls praktisch: Die Ausgabe als Hex-Wert

```
hwrite(l, slv);
```

- Dieser ist für lange Vektoren leichter zu vergleichen