



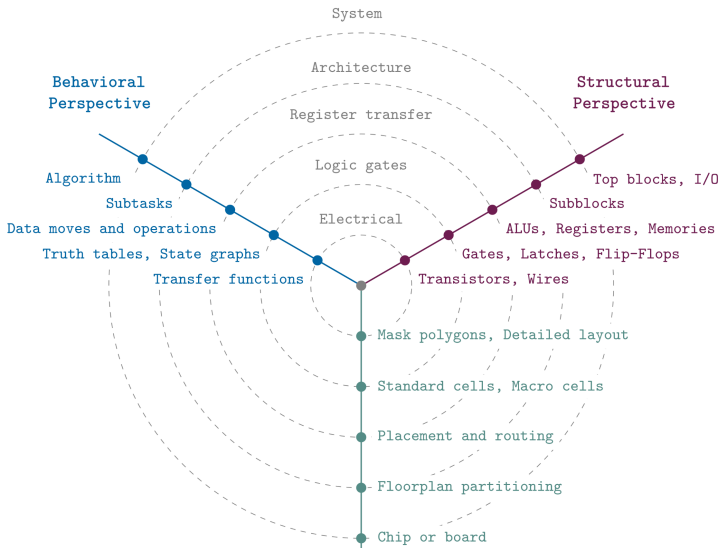
# CheatSheet Digital Design

Fs 2025 – Prof. Dr. Paul Zbinden  
Autoren: Ricca Aaron  
<https://github.com/Rin-Ha-n/DigDes>

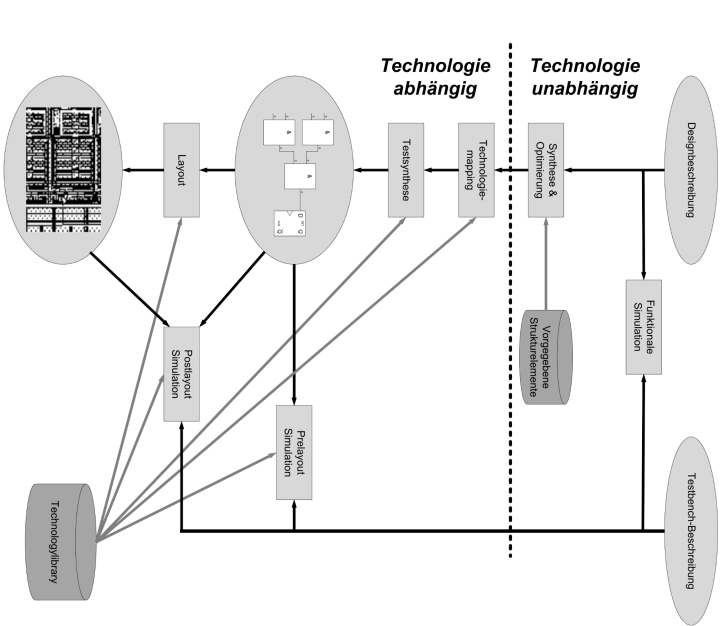
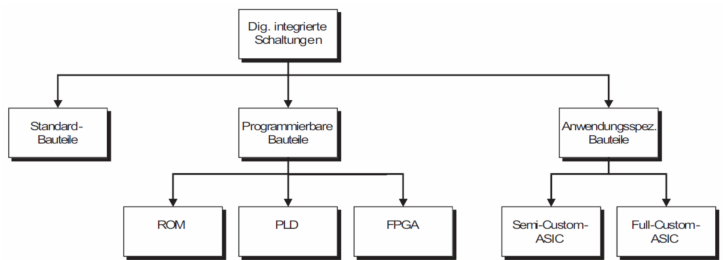
V 1.0 June 22, 2025

## 1 Introduzione

### 1.1 Scelta/caratteristiche dei componenti

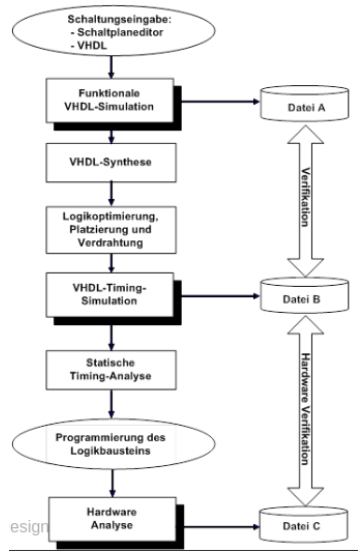


| Kriterien         | Standard Bauteile | ROM | PLD | FPGA | Semi-Custom | Full-Custom |
|-------------------|-------------------|-----|-----|------|-------------|-------------|
| Machbarkeit       | ++                | --  | --  | +    | +           | +++         |
| Zeit Realisierung | +                 | ++  | ++  | ++   | -           | --          |
| Iterationszeit    | -                 | ++  | ++  | ++   | -           | --          |
| NRE               | ++                | +   | +   | +    | -           | ---         |
| Stückpreis        | --                | +   | +   | -    | +           | +++         |



### 1.2 Guida al design

- Design / Entry
- Funktionale Simulation
- Synthese
  - Logikoptimierung
  - Platzierung
  - Verdrahtung
- Timing Simulation
- Statische Timing Analyse
- Herstellungsdaternezeugen



## 2 Programmazione VHDL

### 2.1 Library

Una libreria può contenere componenti e o pacchetti. I componenti sono descrizione di circuiti e realizzazione specifiche, vengono memorizzati nella libreria in modo da

poter essere riutilizzati più volte e da più progettisti contemporaneamente. I blocchi di codice di una libreria sono memorizzati in forma compilata, direttamente eseguibile. Contenuto di una libreria: Components, Packages, Functions, Procedures, Declarations.

```
1 library ieee;  
2 use ieee.std_logic_1164.all; -- CPP: using namespace std;  
3 use ieee.numeric_std.all; -- Solo per operazioni aritmetiche per vettori  
4 use ieee.math_real.all; -- Solo per operazioni aritmetiche per scalari
```

### 2.2 entity dichiarazione

L'entità descrive il componente del progetto. In primo luogo l'entità descrive l'interfaccia (schnittstelle) del componente.

```
1 entity <entity name> is  
2     port (  
3         {port_name : <mode> <type>;} -- <mode> = in | out | inout  
4     );  
5 end <entity name>;
```

L'architettura descrive il comportamento del componente, come funziona e come è realizzato.

### 2.3 architecture

```
1 architecture <architecture_type> of <Entity_name> is  
2     [type_declaration]  
3     [component_declaration]  
4     [subtype_declaration]  
5     [constant_declaration]  
6     [signal_declaration]  
7 begin  
8     -- codice di architettura  
9 end <architecture_type>;
```

### 2.4 component dichiarazione

I componenti sono utilizzati per definire le porte di un'entità, in modo da poterla utilizzare in altre entità.

```
1 component <component_name>  
2     port (  
3         {port_name : <mode> <type>;}  
4     );  
5 end component <component_name>;
```

### 2.5 Port mapping

Il port mapping è utilizzato per collegare le porte dell'entità con i segnali dell'architettura.

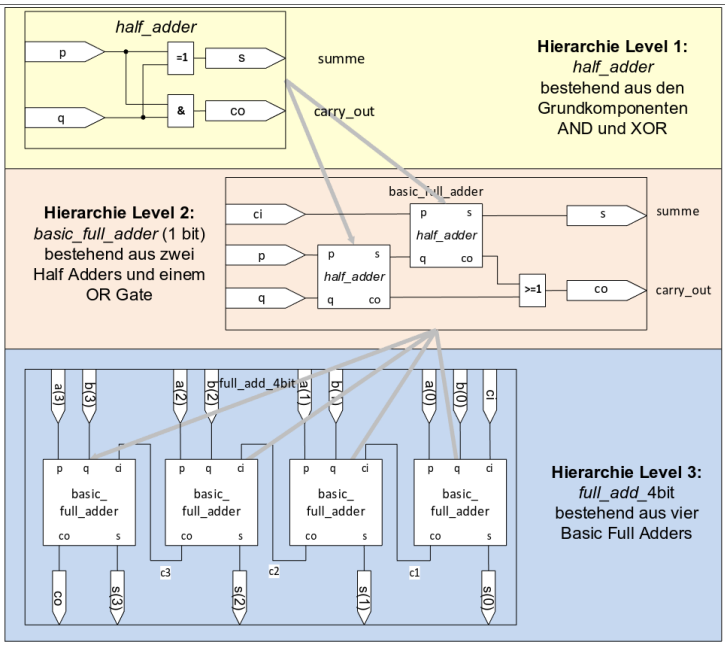
```
1 U1: entity_name  
2     port map (  
3         <port_name> => <signal_name>,  
4         <port_name> => <signal_name>  
5     );
```

#### 2.5.1 Esempio

```
1 architecture structural of half_adder is  
2     -- dichiarazione del componente xor2  
3     component xor2  
4     port (  
5         in1, in2 : in bit;  
6         oup      : out bit  
7     );  
8 end component;  
9  
10    -- dichiarazione del componente and2  
11    component and2  
12    port (  
13        in1, in2 : in bit;  
14        oup      : out bit
```

```
15 );
16 end component;
17
18 begin
19 -- instantiation of ocmponents XOR2 and AND2
20
21 -- Mappatura esplicita
22 U1 : xor2
23     port map (
24         in1 => q,
25         in2 => p,
26         oup => s
27     );
28
29 -- Mappatura implicita
30 U2 : and2
31     port map (p, q, s) -- L'ordine delle porte segue quello della dichiarazione del
32                          ↳ componente!
```

2.6 Hierarchie Level



2.7 Tipi

- <architecture\_type> = Behavioral | Structural | RTL | Dataflow | Tb | ...
- <mode> = in | out | inout
- <type> = bit | bit\_vector | std\_ulogic | std\_ulogic\_vector | integer | boolean

2.7.1 <architecture\_type>

**Behavioral:** si occupa di descrivere il comportamento del circuito, senza preoccuparsi della struttura fisica. Alto livello di astrazione concetto Wahrheitstabelle.

```
1 if rising_edge(clk) then
2   if A = '1' then
3     Y <= B;
4   end if;
5 end if;
```

**Structural:** si occupa di descrivere la struttura fisica del circuito, utilizzando componenti e connessioni tra di essi. Medio livello di astrazione.

```
1 U1: and_gate port map (A => A, B => B, Y => Y1);
2 U2: or_gate port map (A => A, B => C, Y => Y);
```

**RTL:** si occupa di descrivere il circuito a livello di registro e logica combinatoria, utilizzando registri e porte logiche. Basso livello di astrazione. Concetto Boole'sche Ausdrücke.

```
1 if rising_edge(clk) then
2   reg1 <= A and B;
3   reg2 <= reg1 xor C;
4 end if;
```

**Dataflow:** si occupa di descrivere il circuito a livello di flusso di dati, utilizzando porte logiche e segnali. Basso livello di astrazione.

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

**Tb:** si occupa di descrivere il circuito a livello di testbench, utilizzando segnali di test e componenti di test.

```
1 A <= '0'; wait for 10 ns;
2 A <= '1'; wait for 10 ns;
3 assert (Y = expected_value) report "Test failed" severity error;
```

2.7.2 <type> (dichiarazione: segnali, variabili, ...)

Come vanno dichiarati tutti i segnali utilizzati internamente all'architettura. Nella sintesi del codice, è vietato inizializzare i segnali nella dichiarazione!

- bit: rappresenta un singolo bit, con valori '0' e '1'.  
1 signal A : bit;
- bit\_vector: rappresenta un vettore di bit, con valori '0' e '1'.  
1 signal B : bit\_vector(7 downto 0); -- vettore di 8 bit
- std\_logic: rappresenta un singolo bit con valori '0', '1'.  
1 signal C : std\_logic;
- std\_logic\_vector: rappresenta un vettore di std\_logic, con valori '0', '1'.  
1 signal D : std\_logic\_vector(7 downto 0);
- std\_ulogic: rappresenta un singolo bit con valori '0', '1', 'Z' (alta impedenza) e 'X' (indeterminato).  
1 signal E : std\_ulogic;
- std\_u logic\_vector: rappresenta un vettore di std\_ulogic, con valori '0', '1', 'Z' e 'X'.  
1 signal F : std\_u logic\_vector(7 downto 0);
- integer: rappresenta un numero intero, con valori compresi tra  $-2^{31}$  e  $2^{31} - 1$  (è necessario definire l'intervallo di utilizzo).  
1 signal G : integer range 0 to 255; -- intervallo di utilizzo
- boolean: rappresenta un valore booleano, con valori true e false.  
1 signal H : boolean; -- true or false

2.8 Nebenläufige Signalzuweisungen "y<=x"

2.8.1 Definizione dei segnali

```
1 singal <signal_name> :{,<signal_name>} : <type>
2 [:= inizial_val]; -- inicial_value è opzionale
```

2.8.2 Unbedingte Signalzuweisung

L'assegnazione dei segnali è incondizionata, quindi indipendente.

```
1 y <= '0';
2 y <= a and b;
```

2.8.3 Bedingte Signalzuweisung

L'assegnazione dei segnali è eseguita in modo sequenziale, si controlla una condizione e se corretta si assegna il valore, sennò si procede con la prossima condizione.

```
1 y <= '0' when a = '1' else
2   '1' when a = '0';
```

2.8.4 Selektive Signalzuweisung

L'assegnazione dei segnali è eseguita in modo selettivo, viene selezionata il valore in base alla condizione.

```
1 with s select y <=
2   '0' when "00", -- quando s = "00" y <= '0'
3   '1' when "01", -- quando s = "01" y <= '1'
4   'Z' when "10", -- quando s = "10" y <= 'Z'
5   'X' when others; -- quando s = altro y <= 'X'
```

2.8.5 aggregate

L'aggregazione dei segnali permette di aggregare segnali individuali in un unico segnale.

```
1 y <= (a, b, '1', '0'); -- Assegnazione implicita
2 y <= (0 => '0', 1 => '1', 2 => b, others => a); -- Assegnazione esplicita (↳
   ↳ posizione_vettoriale> => <valore>)
```

2.8.6 concatenate

La concatenazione dei segnali permette di concatenare segnali in un unico segnale.

```
1 y <= v_1 & v_2;
```

2.9 Nebenläufige Prozesse

I processi sono "Nebenläufige" di conseguenza iniziano ad essere eseguiti in concorrenza. Ma all'interno il codice viene eseguito normalmente (istruzioni sequenziali, sequenzialmente. istruzioni parallele in modo parallelo).

I processi sono sezioni di codice che vengono eseguite ogni volta che un **Segnale sensibile** nella lista sensibile (Sensitivitätsliste) cambia di stato.

```
1 process (clk, reset)
2   begin
3     if reset = '1' then
4       -- inserisci il codice da eseguire in caso di reset
5     elsif rising_edge(clk) then
6       -- inserisci il codice da eseguire ad ogni fronte di salita del clock
7     end if;
8   end process;
```

2.9.1 sequenzielle Anweisungen im Prozesse

Le istruzioni che vengono eseguite strettamente sequenzialmente all'interno di un processo sono:

```
1 -- struttura if else:
2 if condition_a then
3   {sequential statements}
4 elsif condition_b then
5   {sequential statements}
6 else
7   {sequential statements}
8 end if;
9
10
11 -- struttura case when:
12 case expression is
13   when choice_a => {sequential statements}
14   when choice_b => {sequential statements}
15   when others => {sequential statements}
16 end case;
```

2.9.2 Eigenschaften nebenläufiger Prozesse

Le proprietà più importanti, ovvero le estensioni rispetto alle assegnazioni di segnale, possono essere così riassunte:

- I processi possono assegnare due o più segnali contemporaneamente.
- L'elaborazione delle informazioni per l'assegnazione dei segnali avviene in una sequenza di comandi che vengono eseguiti uno dopo l'altro (procedurale).

- I processi permettono l'uso di variabili per la memorizzazione temporanea dei valori dei segnali.
- Grazie all'uso delle liste di sensibilità è garantito un miglior controllo sulle condizioni di esecuzione della parte di codice.

### 2.9.3 Variablen in nebenläufigen Prozessen

Le variabili offrono due opzioni utili nei processi:

- Accesso a un valore aggiornato all'interno del processo stesso.
- Preparazione di un'espressione di controllo, ad esempio per un "case when".

Le variabili sono dichiarate all'interno dei processi e sono visibili esclusivamente all'interno degli stessi. Il valore assegnato può essere letto immediatamente. L'assegnazione di valore a una variabile avviene con l'operatore `:=`, a differenza dell'assegnazione ai segnali che utilizza `<=`.

```
variable <var_name> {,var_name}: <type> [:= expression];
```

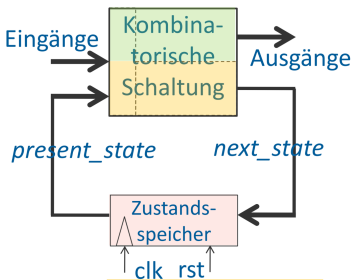
## 3 State machine

Le **Finite State Machine** (FSM) sono macchine a base di circuiti logici sequenziali.

Sono in grado quindi di eseguire operazioni logiche e di poterle memorizzare in modo da consegnare in uscita una funzione che è dipendente dallo stato attuale (memorizzato con gli input precedenti) e opzionalmente anche dagli input attuali (Mealy).

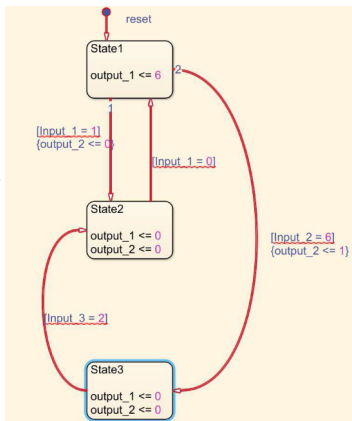
Le tre alternative proposte di seguito sono delle possibilità di incapsulamento standardizzato della funzione desiderata, qualunque essa sia.

- Cit. Alessio Ciceri



### 3.1 Bubble diagram

- **Bolle:** Ogni bolla rappresenta uno stato
- **Freccie:** Condizione per passare da uno stato all'altro dev'essere scritta accanto alla freccia.
- **Moore:** Gli output sono associati agli stati, quindi scritti dentro a quest'ultimi.
- **Melay:** Gli output sono associati alle transizioni, quindi scritti accanto alle frecce.



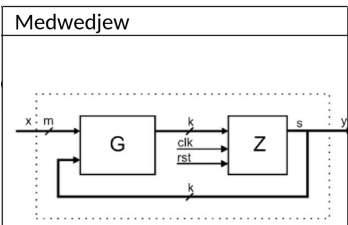
### 3.2 Medwedjew (sincrona)

Composta da un blocco di logica combinatoria(G) che risolve la funzione desiderata e da un blocco di memoria(Z) che memorizza gli stati.

=> Gli Input e lo stato attuale della FSM vengono processati da una logica combinatoria(G)

=> Il risultato della logica viene memorizzato nella Zustandspeicher(Z)

=> L'uscita è esattamente la copia di tutti gli stati memorizzati(s).



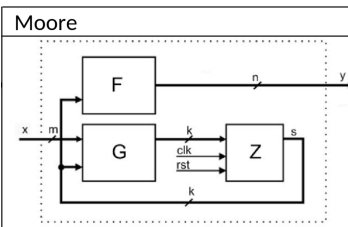
### 3.3 Moore (sincrona)

Come Medwedjew ma con una logica dedicata sul ramo di output(s).

Tipicamente utile per output più complessi/numerosi rispetto agli stati memorizzati(k ≠ n).

=> logica combinatoria aggiuntiva sul ramo d'uscita (F)

=> Più efficiente di Medwedjew per la memorizzazione degli stati

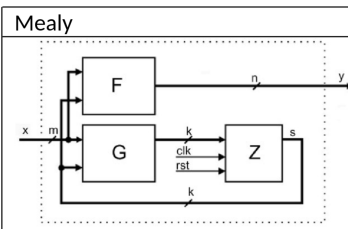


### 3.4 Mealy (sincrona e asincrona in F)

Si tratta della Versione Moore dove la logica sul ramo d'uscita è dipendente anche a segnali provenienti direttamente dagli input della FSM

=> Necessaria se y dipende asincronamente da delle entrate

=> Più complessa.



### 3.5 Codice scheletro FSM (G,Z,F)

```
1
2 G: process(present_state, inputs)
3 begin
4   next_state <= default_state;
5   case present_state is
6     when X_state =>
7       next_state <= Y_state;
8     when others =>
9       next_state <= R_state;
10  end case;
11 end process;
12
13 Z: process(clk)
14 begin
15   if clk'event and clk = '1' then
16     if reset = '1' then
17       present_state <= reset_state;
18     else
19       present_state <= next_state;
20     end if;
21   end if;
22 end process;
23
24 F: process(present_state, )
```

```
25 begin
26   oup <= default_value;
27   case present_state is
28     when X_state =>
29       oup <= "1001";
30     when others =>
31       oup <= "1111";
32   end case;
33 end process;
```

### 3.6 Codifica degli stati (Z-Register)

La dimensione del registro =  $2^n - \text{bit}$  = Flip-Flop necessari Gli stati di una FSM possono essere codificati in diversi modi, tra cui:

- **Codifica binaria:** ogni stato è rappresentato da un codice binario unico.
- **Codifica Gray:** simile alla codifica binaria, ma le transizioni tra stati adiacenti cambiano solo un bit alla volta. La dimensione del registro = n bit
- **Codifica one-hot:** ogni stato è rappresentato da un bit attivo, con tutti gli altri bit a zero.
- **Codifica one-cold:** L'inverso di one-hot.

## 4 Sostituzione discreta per segnali elettrici

### 4.1 Std\_ulogic

| Logik-wert | Bedeutung      | Verwendung   | Für Logik-Synthese |
|------------|----------------|--|--------------------|
| 'U'        | Uninitialized  | Signal ist im Simulator noch nicht initialisiert. In der Hardware ist das Signal unbekannt. Typisch für nicht initialisierte Speicherstelle nach Start-up.                                       | Nein               |
| 'X'        | Undefined      | Signal ist dem Simulator unbekannt. Typisch für Treiberkonflikte, d.h. ein Signal wird von mehr als einer Quelle getrieben. In der Welt der Hardware kommt dies vor allem bei Buskonflikten vor. | Nein               |
| '0'        | Strong zero    | Low Pegel eines Standardausganges. Regelfall für Hardware 0 Pegel.   | Ja                 |
| '1'        | Strong one     | High Pegel eines Standardausganges. Regelfall für Hardware 1 Pegel.  | Ja                 |
| 'Z'        | High Impedance | Signalausgang von Leitung entkoppelt. Typisch für Hochimpedanz Ausgang eines Three-State Treibers bei bidirektionalen Bussen.  | Ja                 |
| 'L'        | Weak zero      | Low Pegel eines schwachen Treiberausganges. Typisch für ein Signal, das mit einem pull-down Widerstand auf Low Potenzial gezogen wird.   | Nein               |
| 'H'        | Weak one       | High Pegel eines schwachen Treiberausganges. Typisch für ein Signal, das mit einem pull-up Widerstand auf High Potenzial gezogen wird.   | Nein               |
| 'W'        | Weak unknown   | Simulator erkennt Treiberkonflikt. Dieser Fall ist analog zu 'X' mit dem Unterschied, dass der Treiberkonflikt hier durch schwache Treiber verursacht wird.                                      | Nein               |
| '-'        | Dont Care      | Logikzustand des Ausgangssignals ist bedeutungslos. Existiert nur in der Schaltungssynthese und wird für Logikminimierung gebraucht.   | Nein               |

## 5 Simulazione e rappresentazione temporale in VHDL

### 5.1 Modelling Temporal Behaviour

- **Event Queue:** Il simulatore utilizza una coda di eventi per elaborare gli eventi in ordine, in modo da non avere conflitti di temporizzazione.
- **Delta-Time Model:** Un ciclo delta compone in tre fasi:

1. Richiesta di aggiornamento: attesa fino a quando il processo o l'assegnazione del segnale venga attivata da un evento.
2. Esecuzione del processo: tutti i processi attivi vengono eseguiti fino alla fine o fine alla prossima istruzione attesa.
3. Assegnazione dei segnali: dopo l'esecuzione dei processi attivi, vengono eseguite le assegnazioni dei segnali corrispondenti.

L'assegnazione del segnale può a sua volta attivare nuovi Processi, che vengono eseguiti nel ciclo delta successivo. Una volta completate queste tre fasi, la simulazione passa al ciclo delta successivo.

**Transport Delay:**

- Vero e proprio ritardo fisico.
- **Tutti gli impulsi si propagano.**
- Syntax: B <= transport A after tp;

**Inertial Delay (default):**

- Impulsi brevi vengono filtrati (ignorati).
- Syntax: B <= A after tp;

**5.2 Infrastruttura di simulazione e Test (Test-Bench)**

L'architettura posta sotto test si chiama DUT (Device Under Test) e viene istanziata all'interno del test-bench. Il Test-Bench stimola le entrate del DUT e verifica le uscite, idealmente dovrebbero essere stimulate tutte le possibilità delle entrate. In linea di principio il Test-Bench non deve essere sintetizzabile, in quanto il suo scopo è quello di verificare il corretto funzionamento del DUT. Blocchi del Test-Bench:

- **Stimulus Generation:** Genera segnali di ingresso per
- **DUT Instantiation:** Istanzia il DUT da testare.
- **Response Monitor:** Verifica le uscite del DUT confrontandole con i risultati attesi.

**5.2.1 Controllo automatico della risposta del DUT**

Anweisung ASSERT:

- Utilizzata per verificare le condizioni attese.
- Sintassi: assert condition report "message";

- Se la condizione non è soddisfatta, viene generato un errore con il messaggio specificato.

**6 Examples**

**6.1 Flip flop**

```
1 entity d_ff_rst is
2   port(
3     clk : in bit;
4     rst : in bit;
5     d : in bit;
6     q : out bit
7   );
8 end d_ff_rst;
9
10 architecture behavioral of d_ff_rst is
11 begin
12   register : process(clk, rst)
13     -- d fehlt in Sens.list
14     begin -- Nur clk und Reset
15       -- aktivieren Prozess
16       if (rst = '1') then
17         -- Asynchroner Reset
18         q <= '0';
19         -- wird zuerst abgearbeitet
20       elsif (clk'event and clk = '1') then
21         -- Sintassi per la detezione del fianco salita del clock
22         q <= d; -- Synchroner Teil
23       end if; -- Kein abschliessendes
24       -- else: in allen anderen
25     end process; -- Füllen wird gespeichert.
26 end behavioral;
```

**6.2 Test bench (dt)**

```
1 -- Library declarations
2 library ieee;
3 use ieee.std_logic_1164.all;
4 use ieee.numeric_std.all;
5
6 -- entity declarations
7 entity exor_tb is
8 end exor_tb;
9
10 -- architecture declarations
```

```
11 architecture tb of exor_tb is
12   -- constant declarations
13   constant sim_cyc : time := 10 ns;
14   -- signal declarations
15   signal tb_a, tb_b : std_ulogic;
16   signal tb_y: std_ulogic;
17   -- Component declarations
18   component xor2
19     port(
20       xor2_a, xor2_b : in std_ulogic;
21       xor2_y
22         : out std_ulogic);
23   end component;
24   -- configuration
25   for all : xor2 use entity work.xor2(behavioral);
26
27 begin
28   -- instance assignments
29   dut : xor2
30     port map(
31       xor2_a => tb_a,
32       xor2_b => tb_b,
33       xor2_y => tb_y
34     );
35
36   -- Signal assignments for stimuli
37   Stimuli : process
38     variable tb_in_vec: std_ulogic_vector(2 - 1 downto 0);
39   begin
40     for i in 0 to (2 ** 2) - 1 loop
41       tb_in_vec := std_ulogic_vector(to_unsigned(i, 2));
42       tb_a <= tb_in_vec(1);
43       tb_b <= tb_in_vec(0);
44       wait for sim_cyc;
45     end loop;
46     wait;
47   end process;
48
49   -- Evaluation of responses
50   Response : process
51   begin
52     wait for (sim_cyc - 1 ns);
53     assert (tb_y = '0') report "error at vector 00" severity error;
54     wait for sim_cyc;
55     assert (tb_y = '1') report "error at vector 01" severity error;
56     wait for sim_cyc;
57     assert (tb_y = '1') report "error at vector 10" severity error;
58     wait for sim_cyc;
59     assert (tb_y = '0') report "error at vector 11" severity error;
60   end process;
61 end tb;
```