

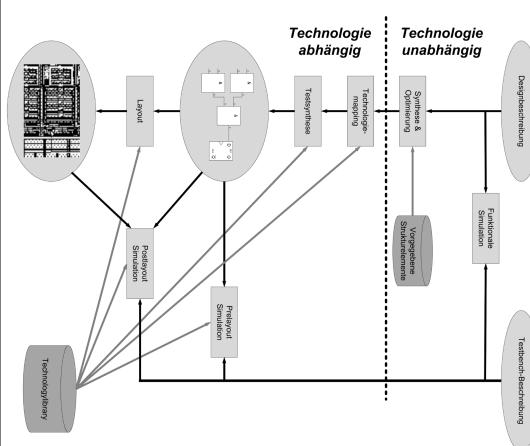
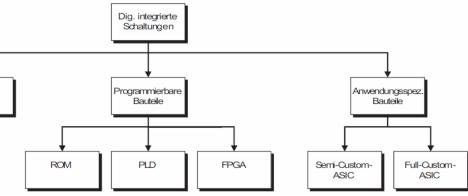
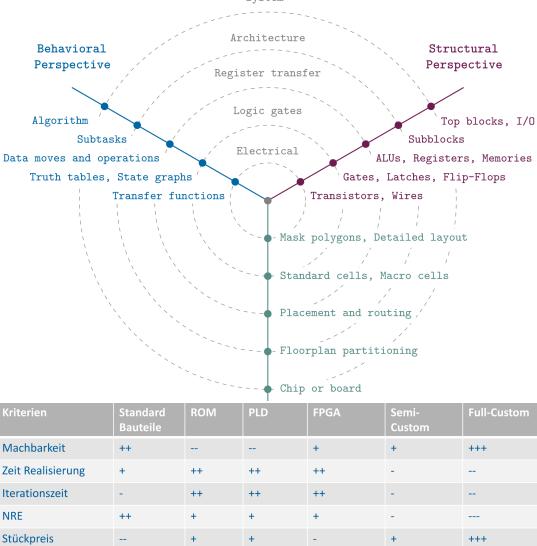
CheatSheet Digital Design

V 1.0 June 22, Fs 2025 – Prof. Dr. Paul Zbinden
Autoren: Ricca Aaron

<https://github.com/Rin-Ha-n/DigDes>

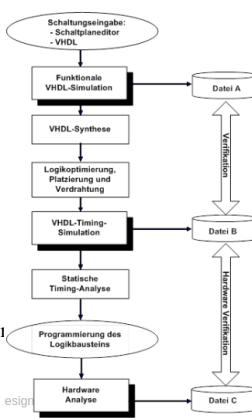
1 Introduzione

1.1 Scelta/caratteristiche dei componenti



1.2 Guida al design

1. Design / Entry
2. Funktionale Simulation
3. Synthese
4. Implementierung
 - Logikoptimierung
 - Platzierung
 - Verdrahtung
5. Timing Simulation
6. Statische Timing Analyse
7. Herstellungsdatenerzeuger



2 Programmazione VHDL

2.1 Introduzione

In VHDL sono vietati gli if e le variabili al di fuori dei processi. È anche vietato l'uso di "Feedback" in quanto creano cose brutte.

2.2 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.

```
library ieee;
use ieee.std_logic_1164.all; -- CPP: using namespace std;
use ieee.numeric_std.all; -- X operazioni aritmetiche x vettori
use ieee.math_real.all; -- X operazioni aritmetiche x scalari
```

2.3 entity dichiarazione

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

```
entity <entity_name> is
    port (
        <port_name> : <mode> <type>; -- <mode> = in | out |
                                         --> inout
    );
end <entity name>;
```

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

2.4 architecture

```
architecture <architecture_type> of <entity_name> is
begin
    -- codice di architettura
end <architecture_type>;
```

2.5 component dichiarazione

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

```
component <component_name>
    port (
        <port_name> : <mode> <type>;
    );
end component <component_name>;
```

2.6 Port mapping

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

```
U1: entity_name
port map (
    <port_name> => <signal_name>,
    <port_name> => <signal_name>
);
```

2.6.1 Esempio

```
architecture structural of half_adder is
    -- dichiarazione del componente xor2
    component xor2
        port (
            in1, in2 : in bit;
            out      : out bit
        );
    end component;

    -- dichiarazione del componente and2
    component and2
        port (
            in1, in2 : in bit;
            out      : out bit
        );
    end component;

begin
    -- instantiation of components XOR2 and AND2
    begin
        -- instantiation of components XOR2 and AND2
    end;
```

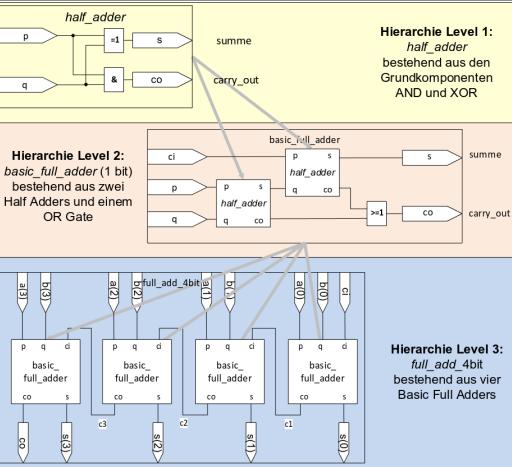
Mappatura esplicita

```
U1 : xor2
port map (
    in1 => q,
    in2 => p,
    out => s
);
```

Mappatura implicita

```
U2 : and2
port map (p, q, s) -- L'ordine delle porte segue quello
                     -- della dichiarazione del componente!
```

2.7 Hierarchie Level



2.8 Tipi

- <architecture_type> = Behavioral | Structural | RTL | Dataflow | Tbl | (...)
- <mode> = in | out | inout
- <type> = bit | bit_vector | std_logic | std_logic_vector | integer | boolean

2.8.1 <architecture_type>

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

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

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

```
U1: and_gate port map (A => A, B => B, Y => Y1);
U2: or_gate port map (A =1, 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 Boolesche Ausdrücke.

```
if rising_edge(clk) then
    reg1 <= A and B;
    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 $Y \leq (A \text{ and } B) \text{ or } (\neg C)$;

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;

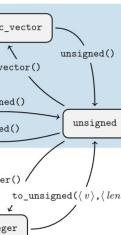
2 A <= '1'; wait for 10 ns;

3 assert (Y = expected_value) report "Test failed" severity error;

2.8.2 <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, bit_vector, std_logic, std_logic_vector, std_ulogic, std_ulogic_vector, integer, boolean



- std_logic: rappresenta un singolo bit con valori '0', '1'.
- signal C : std_logic;
- std_ulogic: rappresenta un singolo bit con valori '0', '1', 'Z' (alta impedenza) e 'X' (indeterminato).
- 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.

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 im 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 Standardausgangs. Regelfall für Hardware 0 Pegel.	Ja
'1'	Strong one	High Pegel eines Standardausgangs. 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 Treiberausgangs. Typisch für ein Signal, das mit einem pull-down Widerstand auf Low Potenzial gezogen wird.	Nein
'H'	Weak one	High Pegel eines schwachen Treiberausgangs. 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

2.9 Generics

- VHDL può passare parametri a un modello

- Dichiarato nella descrizione dell'interfaccia.
 - generic dev'essere dichiarato in entity e in component
- ```

1 entity counter_generic is
2 generic(CNT_MAX : natural := 127); -- generic dichiarato
3 port(
4 clk, rst, ena : in std_ulogic;
5 count : out natural range 0 to CNT_MAX -1);
6 end counter_generic;

```

## 2.10 Nebenläufige Signalzuweisungen "y<=x"

### 2.10.1 Definizione dei segnali

```

1 signal <signal_name> :{,<signal_name>} :<type>
2 [:= inizial_value]; -- inizial_value è opzionale

```

### 2.10.2 Unbedingte Signalzuweisung

L'assegnazione dei segnali è incondizionata, quindi indipendente.

```

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

```

### 2.10.3 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';

```

### 2.10.4 Selektive Signalzuweisung

L'assegnazione dei segnali è eseguita in modo selettivo, viene selezionata la 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'

```

### 2.10.5 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>)

```

### 2.10.6 concatenate

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

```

1 y <= v_1 & v_2;

```

## 2.11 Nebenläufige Prozesse

Solo all'interno dei processi è possibile utilizzare le variabili e le istruzioni sequenziali.

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 (Sensitivitàlistre) cambia di stato.

Nota: se un segnale usato in un processo non è nella lista di sensibilità, il suo valore corrisponde al valore durante la chiamata del processo/indefinito nel caso cambia in concomitanza.

```

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
7 -- salita del clock
8 end if;
9 end process;

```

### 2.11.1 sequenzielle Anweisungen im Prozesse

Le istruzioni che vengono eseguite strettamente sequenziali 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 -- struttura case when:
11 case expression is
12 when choice_a => {sequential statements}
13 when choice_b => {sequential statements}
14 when others => {sequential statements}
15 end case;

```

### 2.11.2 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.

### 2.11.3 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 <=.

```

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

```

## 2.12 Modellazione del Comportamento Temporale

• **Event Queue:** I simulatori utilizzano una coda degli eventi per elaborare gli eventi in ordine cronologico.

• **Modello Delta-Time:** Modello funzionale (nessun ritardo reale). Introduce un ritardo infinitesimale (cicli delta). Separa causa ed effetto nelle forme d'onda, anche se appaiono simultanei.

• **Ritardo di Trasporto:** Modello il ritardo fisico reale.

Sintassi: B <= transport A after tp;

Tutti i cambiamenti di segnale (inclusi i glitch) vengono propagati.

• **Ritardo Inerziale (default):** Modello il filtraggio dei glitch nei circuiti reali.

Sintassi: B <= A after tp;

Gli impulsi brevi (inferiori al tempo di ritardo) vengono ignorati.

## 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 optionalmente anche dagli

input attuali(Mealy).

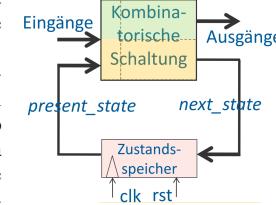
Le tre alternative proposte di

seguito sono delle possibilità

di incapsulamento standardizzato della funzione desiderata,

qualunque essa sia.

- Cit. Alessio Ciceri



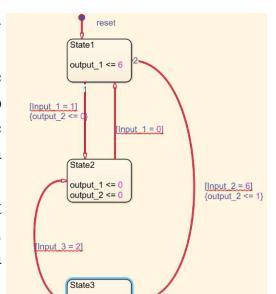
### 3.1 Bubble diagram

• **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.

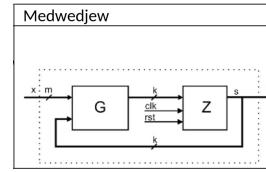
• **Melay:** Gli output sono associati alle transizioni, quindi scritti accanto alle frecce.



## 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).



```

17 if clk'event and clk = '1' then
18 if reset = '1' then
19 present_state <= reset_state;
20 else
21 present_state <= next_state;
22 end if;
23 end if;
24 end process;

25
26 F: process(present_state,)
27 begin
28 oup <= default_value;
29 case present_state is
30 when X_state =>
31 oup <= "1001";
32 when others =>
33 oup <= "1111";
34 end case;
35 end process;

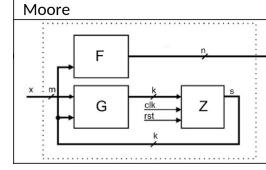
```

## 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( $k \neq n$ ).

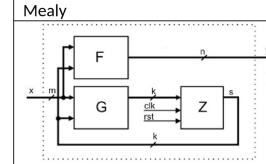
=> logica combinatoria aggiuntiva sul ramo d'uscita (F)  
=> Più efficiente di Medwedjew per la memorizzazione degli stati



## 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 type state_type is (st_normal,st_hset,st_mset,st_ssync);
2 signal <present_state>, <next_state>: <state_type>;
3
4 G: process(present_state, inputs)
5 begin
6 next_state <= default_state;
7 case present_state is
8 when X_state =>
9 next_state <= Y_state;
10 when others =>
11 next_state <= R_state;
12 end case;
13 end process;
14
15 Z: process(clk)
16 begin

```

**Inertial Delay (default):**

## 3.6 Codifica degli stati (Z-Register)

La dimensione del registro =  $2^n -> bit$  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 Simulazione e rappresentazione temporale in VHDL

### 4.1 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:
  - Richiesta di aggiornamento: attesa fino a quando il processo o l'assegnazione del segnale venga attivata da un evento.
  - Esecuzione del processo: tutti i processi attivi vengono eseguiti fino alla fine o fine alla prossima istruzione attesa.
  - 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 completeate queste tre fasi, la simulazione passa al ciclo delta successivo.

**Transport Delay:**

- Vero e proprio ritardo fisico.
- Tutti gli impulsi si propagano.**
- Syntax:  $B \leqslant \text{transport } A \text{ after } tp;$

**Inertial Delay (default):**

- Impulsi brevi vengono filtrati (ignorati).
- Syntax:  $B \leqslant A \text{ after } tp;$

## 4.2 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 stimolate 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.

### 4.2.1 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.

## 5 Examples

### 5.1 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; -- Fällen wird gespeichert.
26 end behavioral;

```

### 5.2 Latch

```

1 process_3 : process(s,a,b)
2 begin if s = '0' then
3 oup <= a;
4 elsif s = '1' then
5 oup <= b;
6 end if;

```

7 end process;

## 5.3 Generic Counter

```

-- Dev'esserci la libreria ieee.math_real.all;
1 entity counter_generic is
2 generic(CNT_MAX : natural := 127);
3 port(
4 clk, rst, ena : in std_logic;
5 count : out std_ulogic_vector
6 (natural.ceil(log2(real(CNT_MAX + 1))) - 1 downto 0));
7 end counter_generic;

```

## 5.4 Clock Divider

```

1 entity clockdivider is
2 generic(DIV_FACTOR : natural := 128);
3 port(
4 clk, rst, ena : in std_ulogic;
5 clk_out : out std_ulogic);
6 end entity clockdivider;
7 architecture behavioral of clockdivider is
8 -- pro Register ein present- und next-signal
9 signal cnt_present, cnt_next : natural range 0 to (DIV_FACTOR
10 -> 1);
11 signal tick_present, tick_next : std_ulogic;
12 begin

```

## 5.5 Test bench (tb)

```

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 : out std_ulogic);
22 end component;
23 -- configuration
24 for all : xor2 use entity work.xor2(behavioral);
25
26 begin
27 -- instance assignments
28 dut : xor2
29 port map(
30 xor2_a => tb_a,
31 xor2_b => tb_b,
32 xor2_y => tb_y
33);
34
35 -- Signal assignments for stimuli
36 Stimuli : process
37 variable tb_in_vec: std_ulogic_vector(2 - 1 downto 0);
38 begin
39 for i in 0 to (2 ** 2) - 1 loop
40 tb_in_vec := std_ulogic_vector(to_unsigned(i, 2));
41 tb_a <= tb_in_vec(1);
42 tb_b <= tb_in_vec(0);
43 wait for sim_cyc;
44 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"
54 ↪ severity error;
55 wait for sim_cyc;
```

```
55 assert (tb_y = '1') report "error at vector 01"
56 ↪ severity error;
57 wait for sim_cyc;
58 assert (tb_y = '0') report "error at vector 10"
59 ↪ severity error;
```

```
60 end process;
61 end tb;
```



# VHDL QUICK REFERENCE CARD

## REVISION 1.1

|               |           |      |                 |
|---------------|-----------|------|-----------------|
| ( )           | Grouping  | [ ]  | Optional        |
| { }           | Repeated  |      | Alternative     |
| <b>bold</b>   | As is     | CAPS | User Identifier |
| <i>italic</i> | VHDL-1993 |      |                 |

### 1. LIBRARY UNITS

```

{[use_clause]}
entity ID is
 [generic ({ID : TYPEID [= expr];});]
 [port ({ID : in | out | inout TYPEID [= expr];});]
 [{declaration}]
begin
 {[parallel_statement]}
end [entity] ENTITYID;
{[use_clause]}
architecture ID of ENTITYID is
 [{declaration}]
begin
 {[parallel_statement]}
end [architecture] ARCHID;
{[use_clause]}
package ID is
 [{declaration}]
end [package] PACKID;
{[use_clause]}
package body ID is
 [{declaration}]
end [package body] PACKID;
{[use_clause]}
configuration ID of ENTITYID is
 for ARCHID
 {[block_config | comp_config]}
 end for;
end [configuration] CONFID;
use_clause::=
 library ID;
 {[use LIBID.PKGID.all;]}
block_config::=
 for LABELID
 {[block_config | comp_config]}
 end for;

```

```

comp_config::=
 for all | LABELID : COMPID
 (use entity [LIBID.]ENTITYID [(ARCHID)]
 {[generic map ({GENID => expr ,})]
 port map ({PORTID => SIGID | expr,});})
 [for ARCHID
 {[block_config | comp_config]}]
 end for;
 end for;) |
 (use configuration [LIBID.]CONFID
 {[generic map ({GENID => expr ,})]
 port map ({PORTID => SIGID | expr,});})
 end for;

```

## 2. DECLARATIONS

### 2.1. TYPE DECLARATIONS

```

type ID is ({ID,});
type ID is range number downto | to number;
type ID is array ({range | TYPEID ,})
 of TYPEID | SUBTYPID;
type ID is record
 {ID : TYPEID,};
end record;
type ID is access TYPEID;
type ID is file of TYPEID;
subtype ID is SCALARTYPID range range;
subtype ID is ARRAYTYPID({range,});
subtype ID is RESOLVFCTID TYPEID;
range ::=
 (integer | ENUMID to | downto
 integer | ENUMID) | (OBJID'reverse_range) |
 (TYPEID range <>)

```

### 2.2. OTHER DECLARATIONS

```

constant ID : TYPEID := expr;
[shared] variable ID : TYPEID [= expr];
signal ID : TYPEID [= expr];
file ID : TYPEID (is in | out string);
 (open read_mode / write_mode
 | append_mode is string,);
alias ID : TYPEID is OBJID;
attribute ID : TYPEID;
attribute ATTRID of OBJID | others | all : class
 is expr;
class ::=
 entity | architecture | configuration |
 procedure | function | package | type |
 subtype | constant | signal | variable |
 component | label

```

```

component ID [is]
 [generic ({ID : TYPEID [= expr];});]
 [port ({ID : in | out | inout TYPEID [= expr];});]
end component [COMPID];
[impure] function ID
 ({constant | variable | signal} ID :
 in | out | inout TYPEID [= expr,]);
 return TYPEID [is]
begin
 {sequential_statement}
end [function] ID;
procedure ID ([{constant | variable | signal} ID :
 in | out | inout TYPEID [= expr,]]);
 [is begin
 {[sequential_statement]}
 end [procedure] ID];
for LABELID | others | all : COMPID use
 (entity [LIBID.]ENTITYID [(ARCHID)])
 (configuration [LIBID.]CONFID)
 {[generic map ({GENID => expr,})]
 port map ({PORTID => SIGID | expr,});}

```

## 3. EXPRESSIONS

```

expression ::=
 (relation and relation) |
 (relation or relation) |
 (relation xor relation)
relation ::= shexpr [relop shexpr]
shexpr ::= sexpr [shop sexpr]
sexpr ::= [+|-] term {addop term}
term ::= factor {mulop factor}
factor ::=
 (prim [* prim]) | (abs prim) | (not prim)
prim ::=
 literal | OBJID | OBJID'ATTRID | OBJID({expr,})
 | OBJID(range) | {[choice {[| choice] =>} expr,]}
 | FCTID({[PARID =>] expr,}) | TYPEID'(expr) |
 TYPEID(expr) | new TYPEID['(expr)] | (expr)
choice ::= sexpr | range | RECFID | others

```

### 3.1. OPERATORS, INCREASING PRECEDENCE

|        |                                   |
|--------|-----------------------------------|
| logop  | and   or   xor                    |
| relop  | =   /=   <   <=   >   =>          |
| shop   | sll   srl   sfa   sra   rol   ror |
| addop  | +   -   &                         |
| mulop  | *   /   mod   rem                 |
| miscop | **   abs   not                    |

© 1995 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

See reverse side for additional information.

#### 4. SEQUENTIAL STATEMENTS

```

wait [on {SIGID,}] [until expr] [for time];
assert expr
[report string] [severity note | warning |
error | failure];
report string
[severity note | warning | error |
failure];
SIGID <= [transport] | [reject TIME inertial]
{expr [after time]};
VARID := expr;
PROCEDUREID({{{PARID =>} expr,}});

[LABEL:] if expr then
{sequential_statement}
[elsif expr then
{sequential_statement}]
[else
{sequential_statement}]
end if [LABEL];

[LABEL:] case expr is
{when choice [{choice}] =>
{sequential_statement}}
end case [LABEL];

[LABEL:] while expr loop
{sequential_statement}
end loop [LABEL];
[LABEL:] for ID in range loop
{sequential_statement}
end loop [LABEL];
next [LOOPLBL] [when expr];
exit [LOOPLBL] [when expr];
return [expression];
null;

```

#### 5. PARALLEL STATEMENTS

```

[LABEL:] block [is]
[generic ({ID : TYPEID;});
[generic map ({GENID => expr,});]
[port ({ID : in | out | inout TYPEID});
[port map ({PORTID => SIGID | expr,});]
[declaration]
begin
{parallel_statement}
end block [LABEL];
[LABEL:] [postponed] process [({SIGID,})
[declaration]
begin
{sequential_statement}
end [postponed] process [LABEL];
[LBL:] [postponed] PROCID({{{PARID =>} expr,}});

```

```

[LABEL:] [postponed] assert expr
[report string] [severity note | warning |
error | failure];
[LABEL:] [postponed] SIGID <=
[transport] | [reject TIME inertial]
{expr [after time]} / unaffected when expr
else} {expr [after time]} | unaffected;
[LABEL:] [postponed] with expr select
SIGID <= [transport] | [reject TIME inertial]
{expr [after time]} |
unaffected when choice [{choice}];
LABEL: COMPID
[[generic map ({GENID => expr,})]
port map ({PORTID => SIGID,});]
LABEL: entity [LIBID.]ENTITYID [(ARCHID)]
[[generic map ({GENID => expr,})]
port map ({PORTID => SIGID,});]
LABEL: configuration [LIBID.]CONFID
[[generic map ({GENID => expr,})]
port map ({PORTID => SIGID,});]
LABEL: if expr generate
[{parallel_statement}]
end generate [LABEL];
LABEL: for ID in range generate
[{parallel_statement}]
end generate [LABEL];

```

#### 6. PREDEFINED ATTRIBUTES

|                             |                             |
|-----------------------------|-----------------------------|
| TYPID'base                  | Base type                   |
| TYPID'left                  | Left bound value            |
| TYPID'right                 | Right-bound value           |
| TYPID'high                  | Upper-bound value           |
| TYPID'low                   | Lower-bound value           |
| TYPID'pos(expr)             | Position within type        |
| TYPID'val(expr)             | Value at position           |
| TYPID'succ(expr)            | Next value in order         |
| TYPID'prec(expr)            | Previous value in order     |
| TYPID'leftof(expr)          | Value to the left in order  |
| TYPID'rightof(expr)         | Value to the right in order |
| TYPID'ascending             | Ascending type predicate    |
| TYPID'image(expr)           | String image of value       |
| TYPID'value(string)         | Value of string image       |
| ARYID'left([expr])          | Left-bound of [nth] index   |
| ARYID'right([expr])         | Right-bound of [nth] index  |
| ARYID'high([expr])          | Upper-bound of [nth] index  |
| ARYID'low([expr])           | Lower-bound of [nth] index  |
| ARYID'range([expr])         | 'left down/to 'right        |
| ARYID'reverse_range([expr]) | 'right down/to 'left        |
| ARYID'length([expr])        | Length of [nth] dimension   |
| ARYID'ascending([expr])     | 'right >= 'left ?           |
| SIGID'delayed([expr])       | Delayed copy of signal      |
| SIGID'stable([expr])        | Signals event on signal     |
| SIGID'quiet([expr])         | Signals activity on signal  |

|                           |                          |
|---------------------------|--------------------------|
| SIGID'transaction([expr]) | Toggles if signal active |
| SIGID'event               | Event on signal ?        |
| SIGID'active              | Activity on signal ?     |
| SIGID'last_event          | Time since last event    |
| SIGID'last_active         | Time since last active   |
| SIGID'last_value          | Value before last event  |
| SIGID'driving             | Active driver predicate  |
| SIGID'driving_value       | Value of driver          |
| OBJID'simple_name         | Name of object           |
| OBJID'instance_name       | Pathname of object       |
| OBJID'path_name           | Pathname to object       |

#### 7. PREDEFINED TYPES

|                     |                                     |
|---------------------|-------------------------------------|
| BOOLEAN             | True or false                       |
| INTEGER             | 32 or 64 bits                       |
| NATURAL             | Integers >= 0                       |
| POSITIVE            | Integers > 0                        |
| REAL                | Floating-point                      |
| BIT                 | '0', '1'                            |
| BIT_VECTOR(NATURAL) | Array of bits                       |
| CHARACTER           | 7-bit ASCII                         |
| STRING(POSITIVE)    | Array of characters                 |
| TIME                | hr, min, sec, ms,<br>us, ns, ps, fs |
| DELAY_LENGTH        | Time => 0                           |

#### 8. PREDEFINED FUNCTIONS

|                                           |                                 |
|-------------------------------------------|---------------------------------|
| NOW                                       | Returns current simulation time |
| DEALLOCATE(ACCESTYPOBJ)                   | Deallocate dynamic object       |
| FILE_OPEN([status], FILEID, string, mode) | Open file                       |
| FILE_CLOSE(FILEID)                        | Close file                      |

#### 9. LEXICAL ELEMENTS

Identifier ::= letter { [underline] alphanumeric }

decimal literal ::= integer [. integer] [E[+|-] integer]

based literal ::= integer # hexint [. hexint] # [E[+|-] integer]

bit string literal ::= B|O|X " hexint "

comment ::= -- comment text

© 1995 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

**Qualis Design Corporation**

Beaverton, OR USA

Phone: +1-503-531-0377 FAX: +1-503-629-5525  
E-mail: info@qualis.com

Also available: 1164 Packages Quick Reference Card  
Verilog HDL Quick Reference Card



# 1164 PACKAGES QUICK REFERENCE CARD

## REVISION 1.0

|             |                                             |      |                 |
|-------------|---------------------------------------------|------|-----------------|
| ()          | Grouping                                    | [ ]  | Optional        |
| {}          | Repeated                                    |      | Alternative     |
| <b>bold</b> | As is                                       | CAPS | User Identifier |
| b ::=       | BIT                                         |      |                 |
| u/l ::=     | STD_ULOGIC/STD_LOGIC                        |      |                 |
| bv ::=      | BIT_VECTOR                                  |      |                 |
| uv ::=      | STD_ULOGIC_VECTOR                           |      |                 |
| lv ::=      | STD_LOGIC_VECTOR                            |      |                 |
| un ::=      | UNSIGNED                                    |      |                 |
| sg ::=      | SIGNED                                      |      |                 |
| na ::=      | NATURAL                                     |      |                 |
| in ::=      | INTEGER                                     |      |                 |
| sm ::=      | SMALL_INT<br>(subtype INTEGER range 0 to 1) |      |                 |
| c ::=       | commutative                                 |      |                 |

## 1. IEEE's STD\_LOGIC\_1164

### 1.1. Logic Values

|         |                     |
|---------|---------------------|
| 'U'     | Uninitialized       |
| 'X'/'W' | Strong/Weak unknown |
| '0'/'L' | Strong/Weak 0       |
| '1'/'H' | Strong/Weak 1       |
| 'Z'     | High Impedance      |
| '-'     | Don't care          |

### 1.2. PREDEFINED TYPES

|                                             |                         |
|---------------------------------------------|-------------------------|
| <b>STD_ULOGIC</b>                           | Base type               |
| Subtypes:                                   |                         |
| <b>STD_LOGIC</b>                            | Resolved STD_ULOGIC     |
| X01                                         | Resolved X, 0 & 1       |
| X01Z                                        | Resolved X, 0, 1 & Z    |
| UX01                                        | Resolved U, X, 0 & 1    |
| UX01Z                                       | Resolved U, X, 0, 1 & Z |
| <b>STD_ULOGIC_VECTOR(na to   downto na)</b> | Array of STD_ULOGIC     |
| <b>STD_LOGIC_VECTOR(na to   downto na)</b>  | Array of STD_LOGIC      |

## 1.3. OVERLOADED OPERATORS

| Description | Left      | Operator   | Right     |
|-------------|-----------|------------|-----------|
| bitwise-and | u/l,uv,lv | <b>and</b> | u/l,uv,lv |
| bitwise-or  | u/l,uv,lv | <b>or</b>  | u/l,uv,lv |
| bitwise-xor | u/l,uv,lv | <b>xor</b> | u/l,uv,lv |
| bitwise-not |           | <b>not</b> | u/l,uv,lv |

## 1.4. CONVERSION FUNCTIONS

| From  | To  | Function                           |
|-------|-----|------------------------------------|
| u/l   | b   | <b>TO_BIT</b> (from, [xmap])       |
| uv,lv | bv  | <b>TO_BITVECTOR</b> (from, [xmap]) |
| b     | u/l | <b>TO_STDULOGIC</b> (from)         |
| bv,ul | lv  | <b>TO_STDLOGICVECTOR</b> (from)    |
| bv,lv | uv  | <b>TO_STDLOGICVECTOR</b> (from)    |

## 1.5. PREDICATES

|                            |                       |
|----------------------------|-----------------------|
| <b>RISING_EDGE(SIGID)</b>  | Rise edge on signal ? |
| <b>FALLING_EDGE(SIGID)</b> | Fall edge on signal ? |
| <b>IS_X(OBJID)</b>         | Object contains 'X' ? |

## 2. IEEE's NUMERIC\_STD

### 2.1. PREDEFINED TYPES

**UNSIGNED(na to | downto na)**

**SIGNED(na to | downto na)**

Arrays of STD\_LOGIC

### 2.2. OVERLOADED OPERATORS

| Left | Op               | Right | Return |
|------|------------------|-------|--------|
|      | <b>abs</b>       | sg    | sg     |
| -    |                  | sg    | sg     |
| un   | +,-,*/,rem,mod   | un    | un     |
| sg   | +,-,*/,rem,mod   | sg    | sg     |
| un   | +,-,*/,rem,mod_c | na    | un     |
| sg   | +,-,*/,rem,mod_c | in    | sg     |
| un   | <,>,<=,>=,=/     | un    | bool   |
| sg   | <,>,<=,>=,=/     | sg    | bool   |
| un   | <,>,<=,>=,=/c    | na    | bool   |
| sg   | <,>,<=,>=,=/c    | in    | bool   |

### 2.3. PREDEFINED FUNCTIONS

|                             |    |
|-----------------------------|----|
| <b>SHIFT_LEFT(un, na)</b>   | un |
| <b>SHIFT_RIGHT(un, na)</b>  | un |
| <b>SHIFT_LEFT(sg, na)</b>   | sg |
| <b>SHIFT_RIGHT(sg, na)</b>  | sg |
| <b>ROTATE_LEFT(un, na)</b>  | un |
| <b>ROTATE_RIGHT(un, na)</b> | un |
| <b>ROTATE_LEFT(sg, na)</b>  | sg |
| <b>ROTATE_RIGHT(sg, na)</b> | sg |
| <b>RESIZE(sg, na)</b>       | sg |
| <b>RESIZE(un, na)</b>       | un |

## 2.4. CONVERSION FUNCTIONS

| From  | To | Function                       |
|-------|----|--------------------------------|
| un,lv | sg | <b>SIGNED</b> (from)           |
| sg,lv | un | <b>UNSIGNED</b> (from)         |
| un,sg | lv | <b>STD_LOGIC_VECTOR</b> (from) |
| un,sg | in | <b>TO_INTEGER</b> (from)       |
| na    | un | <b>TO_UNSIGNED</b> (from)      |
| in    | sg | <b>TO_SIGNED</b> (from)        |

## 3. IEEE's NUMERIC\_BIT

### 3.1. PREDEFINED TYPES

**UNSIGNED(na to | downto na)** Array of BIT  
**SIGNED(na to | downto na)** Array of BIT

### 3.2. OVERLOADED OPERATORS

| Left | Op               | Right | Return |
|------|------------------|-------|--------|
|      | <b>abs</b>       | sg    | sg     |
| -    |                  | sg    | sg     |
| un   | +,-,*/,rem,mod   | un    | un     |
| sg   | +,-,*/,rem,mod   | sg    | sg     |
| un   | +,-,*/,rem,mod_c | na    | un     |
| sg   | +,-,*/,rem,mod_c | in    | sg     |
| un   | <,>,<=,>=,=/     | un    | bool   |
| sg   | <,>,<=,>=,=/     | sg    | bool   |
| un   | <,>,<=,>=,=/c    | na    | bool   |
| sg   | <,>,<=,>=,=/c    | in    | bool   |

### 3.3. PREDEFINED FUNCTIONS

|                             |    |
|-----------------------------|----|
| <b>SHIFT_LEFT(un, na)</b>   | un |
| <b>SHIFT_RIGHT(un, na)</b>  | un |
| <b>SHIFT_LEFT(sg, na)</b>   | sg |
| <b>SHIFT_RIGHT(sg, na)</b>  | sg |
| <b>ROTATE_LEFT(un, na)</b>  | un |
| <b>ROTATE_RIGHT(un, na)</b> | un |
| <b>ROTATE_LEFT(sg, na)</b>  | sg |
| <b>ROTATE_RIGHT(sg, na)</b> | sg |
| <b>RESIZE(sg, na)</b>       | sg |
| <b>RESIZE(un, na)</b>       | un |

### 3.4. CONVERSION FUNCTIONS

| From  | To | Function                  |
|-------|----|---------------------------|
| un,bv | sg | <b>SIGNED</b> (from)      |
| sg,bv | un | <b>UNSIGNED</b> (from)    |
| un,sg | bv | <b>BIT_VECTOR</b> (from)  |
| un,sg | in | <b>TO_INTEGER</b> (from)  |
| na    | un | <b>TO_UNSIGNED</b> (from) |
| in    | sg | <b>TO_SIGNED</b> (from)   |

© 1995 Qualis Design Corporation. Permission to reproduce and distribute strictly verbatim copies of this document in whole is hereby granted.

See reverse side for additional information.