CheatSheet Digital Design

Fs 2025 – Prof. Dr. Paul Zbinden

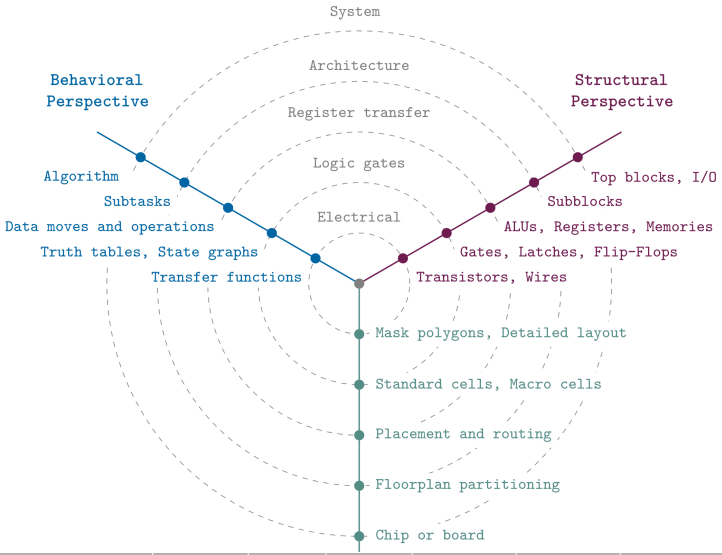
Autoren: Ricca Aaron

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

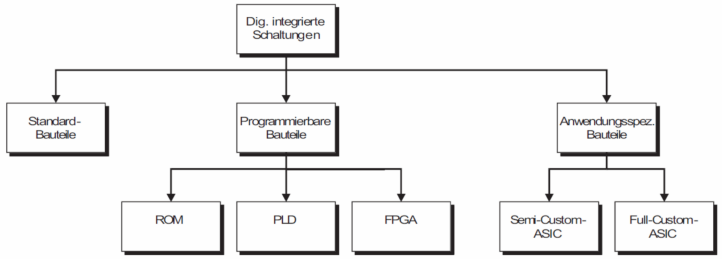
V 1.0June 21, 2025

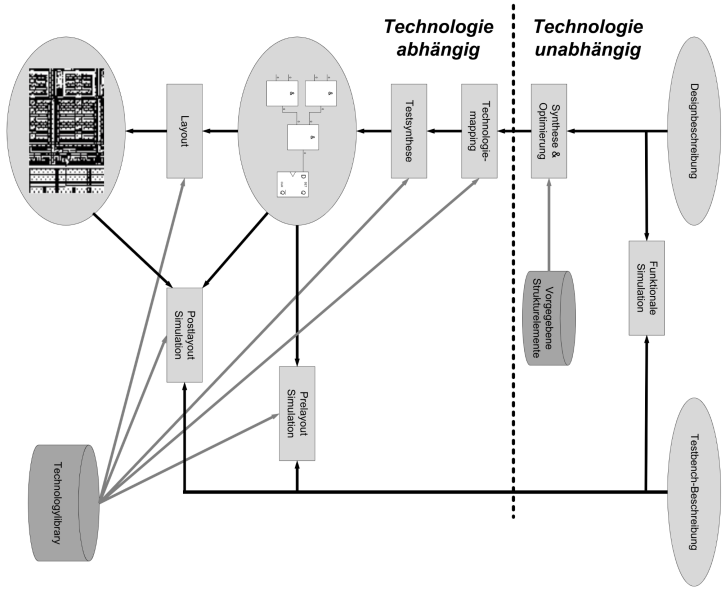
1 Introduzione

1.1 Scelta/caratteristiche dei componenti



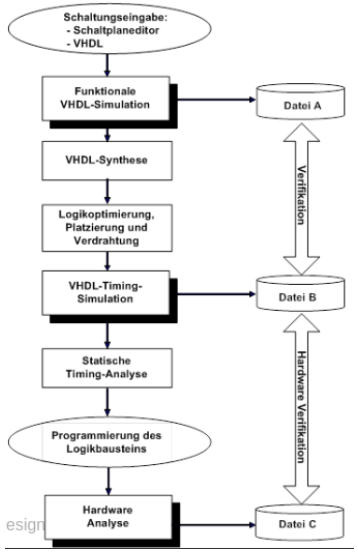
Kriterien	Standard Bauteile	ROM	PLD	FPGA	Semi-Custom	Full-Custom
Machbarkeit	++	--	--	+	+	+++
Zeit Realisierung	+	++	++	++	-	--
Iterationszeit	-	++	++	++	-	--
NRE	++	+	+	+	-	---
Stückpreis	--	+	+	-	+	+++





1.2 Guida al design

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



2 Programmazione VHDL

2.1 Library

Una libreria puo contenere componenti e o pachetti. I componenti sono descrizione di circuiti e realizzazione specifiche, vengono memorizzati nella libreria in modo da poter essere riutilizati piu volte e da piu 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.

```vhdl
library ieee;
use ieee.std_logic_1164.all; -- CPP: using namespace std;
use ieee.numeric_std.all; -- Solo per operazioni aritmetiche per vettori
```

2.2 entity dichiarazione

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

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

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

2.3 architecture

```vhdl
architecture <architecture_type> of <entity_name> is
    [type_declaration]
        [component_declaration]
        [subtype_declaration]
        [constant_declaration]
        [signal_declaration]
begin
    -- codice di architettura
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.

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

2.5 Port mapping

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

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

2.5.1 Esempio

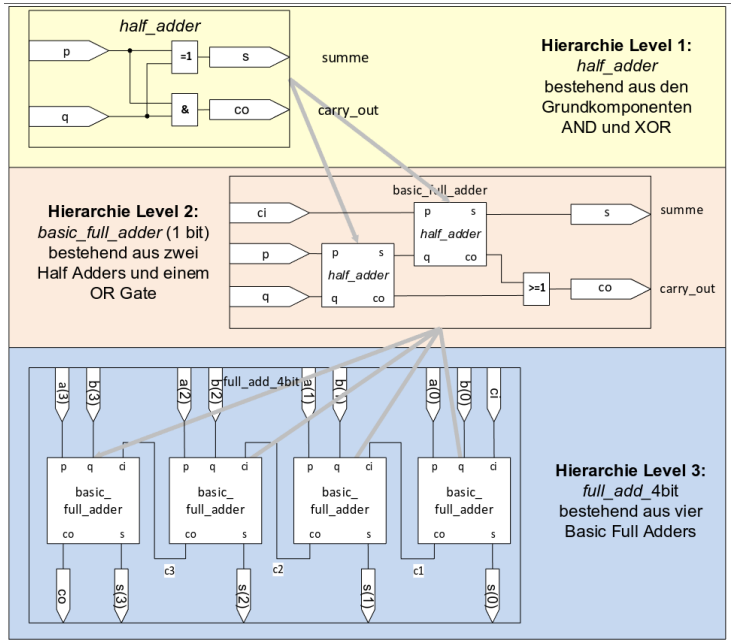
```vhdl
architecture structural of half_adder is
    -- dichiarazione del componente xor2
    component xor2
        port (
            in1, in2 : in bit;
            oup      : out bit
        );
    end component;

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

```
    end component;

    begin
    -- instantiation of ocmponents XOR2 and AND2
    - Mappatura esplicita
    U1 : xor2
        port map (
            in1 => q,
            in2 => p,
            oup  => s
        );
    - Mappatura implicita
    U2 : and2
        port map (p, q, s) -- L'ordine delle porte segue quello della dichiarazione del
            ↪ 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.

```
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 Boolsche Ausdrucke.

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

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

```
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.

```
A <= '0'; wait for 10 ns;
A <= '1'; wait for 10 ns;
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, é vietato inizzializzare i segnali nella dichiarazione!

- `bit`: rappresenta un singolo bit, con valori `'0'` e `'1'`.
  ```
  signal A : bit;
  ```
- `bit_vector`: rappresenta un vettore di bit, con valori `'0'` e `'1'`.
  ```
  signal B : bit_vector(7 downto 0); -- vettore di 8 bit
  ```
- `std_logic`: rappresenta un singolo bit con valori `'0'`, `'1'`.
  ```
  signal C : std_logic;
  ```
- `std_logic_vector`: rappresenta un vettore di std_logic, con valori `'0'`, `'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).
  ```
  signal E : std_ulogic;
  ```
- `std_ulogic_vector`: rappresenta un vettore di std_ulogic, con valori `'0'`, `'1'`, `'Z'` e `'X'`.
  ```
  signal F : std_ulogic_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).
  ```
  signal G : integer range 0 to 255; -- intervallo di utilizzo
  ```
- `boolean`: rappresenta un valore booleano, con valori `true` e `false`.
  ```
  signal H : boolean; -- true or false
  ```

2.8 Nebenläufige Signalzuweisungen "y<=x"

2.8.1 Definizione dei segnali

```
singal <signal_name> :{,<singal_name>} : <type>
[:= inizianol_value]; -- inizial_value é opzionale
```

2.8.2 Unbedingte Signalzuweisung

L'assegnazione dei segnali é incondizionata, quindi indipendente.

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

2.8.3 Bedingte Signalzuweisung

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

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

2.8.4 Selektive Signalzuweisung

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

```
with s select y <=
    '0' when "00", -- quando s = "00" y <= '0'
    '1' when "01", -- quando s = "01" y <= '1'
    'Z' when "10", -- quando s = "10" y <= 'Z'
    '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.

```
y <= (a, b, '1', '0'); -- Assegnazione implicita
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.

```
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 (Sensitivitätliste) cambia di stato.

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

2.9.1 sequenzielle Anweisungen im Prozesse

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

```
-- struttura if else:
if condition_a then
    {sequential statements}
elsif condition_b then
    {sequential statements}
else
    {sequential statements}
end if;


-- struttura case when:
case expression is
    when choice_a => {sequential statements}
    when choice_b => {sequential statements}
    when others => {sequential statements}
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:

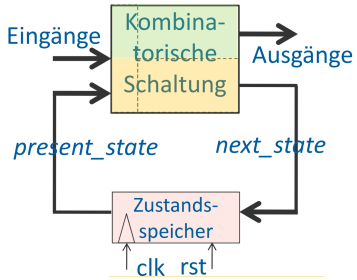
- Accesso a un valore aggiornato all'interno del processo stesso.
- Preparazione di un'espressione di controllo, ad esempio per un "case when".

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];
```

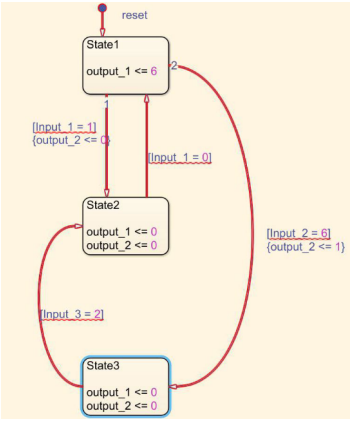
3 State machine

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



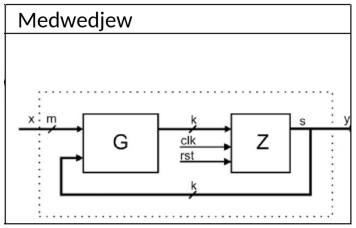
3.1 Bubble diagram

- **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.



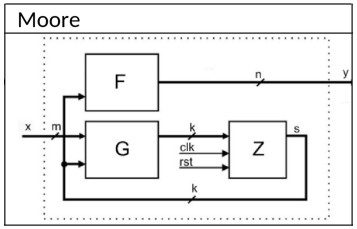
3.2 Medwedjew (sincrona)

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).



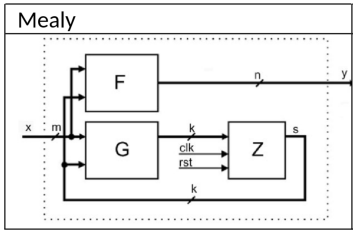
3.3 Moore (sincrona)

Come Medwedjew ma con una logica dedicata sul ramo di output(s).
Tipicamente utile per output più complessi/numerosi rispetto agli stati memorizzati($k \neq n$).
=> 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
2  G: process(present_state, inputs)
3  begin
4      next_state <= de fault_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 $= 2^{n->bit}$ 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.
- **Codifica one-hot**: ogni stato è rappresentato da un bit attivo, con tutti gli altri bit a zero.
- **Codifica one-cold**: simile alla codifica one-hot, ma solo un bit è a zero e tutti gli altri sono attivi.

4 Sostituzione discreta per segnali elettrici

4.1 Segnali elettrici

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 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

5 Examples

5.1 Flip flop

```
1 entity d_ff_rst is
2     port(
3         clk : in bit;
4         rst : in bit;
```

```
        d : in bit;
        q : out bit
        );
end d_ff_rst;

architecture behavioral of d_ff_rst is
begin
    register : process(clk, rst)
           -- d fehlt in Sens.list
    begin  -- Nur clk und Reset
           -- aktivieren Prozess
        if (rst = '1') then
            -- Asynchroner Reset
            q <= '0';
            -- wird zuerst abgearbeitet
        elsif (clk'event and clk = '1') then
              ↪ - Sintassi per la detezione del fianco salita del clock
            q <= d; -- Synchroner Teil
        end if; -- Kein abschliessendes
        -- else: in allen anderen
    end process; -- Fällen wird gespeichert.
end behavioral;
```

6 notebooklm.tex

`std_ulogic` (unresolved) [8]:

- Replaces `bit` type [8].
- **Allows only one driver per signal**; compiler reports error on multiple drivers [8].
- Suitable for three-state outputs using 'Z' [9].
- Recognises uninitialized ('U'), weak ('H', 'L'), and don't care ('-') states [9, 10].
- Recommended for educational projects for its strictness [11].

`std_logic` (resolved) [10]:

- Removes the 'unresolved' restriction, allowing multiple drivers [10].
- **Simulator automatically resolves driver conflicts** based on a resolution function (e.g., '0' and '1' resolving to 'X', '0' and 'Z' to '0') [10, 12].
- More prone to hidden errors (compiler won't catch multiple assignments) [13].
- Widely used in practice for its flexibility, especially for physical implementation concerns [14].
- In VHDL 2008, `std_logic` is a subtype of `std_ulogic` [15].

6.0.1 Modelling Bidirectional Buses in VHDL

- Uses the **`inout`** port mode.
- Writing to the bus: `bus_i_o <= out_i when enable = '1' else (others => 'Z');`
- Reading from the bus: `in_i <= bus_i_o;`

6.1 VHDL Key Concept IV: Event-Based Concept for Temporal Representation in Simulations

6.1.1 Simulation as a Central Step

- Crucial for design verification.
- **Reusable Test-Benches** help with regression testing.
- **Later error detection is more costly.**

6.1.2 Modelling Temporal Behaviour

- **Event Queue:** Simulators use an event queue to process events in order.
- **Delta-Time Model:**
  - Functional simulation model.
  - Infinitesimal time delay (*delta cycle*).
  - Ensures cause-effect separation in waveform views.
- **Transport Delay:**
  - Models real delay.
  - Syntax: `B <= transport A after tp;`
  - **All pulses propagate**.
- **Inertial Delay (default):**
  - Models physical filtering of glitches.
  - Syntax: `B <= A after tp;`
  - **Short pulses filtered out**.

6.1.3 Simulation Setup (Test-Bench)

- Not synthesizable.
- **Device Under Test (DUT)**: Component under test.
- **Stimulus Generator**: Drives input signals.
- **Response Monitor**: Verifies DUT output.
- **`ASSERT`** Statement:
  - Syntax: `assert (cond) report "msg" severity level;`
  - Levels: `note`, `warning`, `error`, `failure`.

6.2 VHDL Key Concept VIII: Arithmetic and Data Types

6.2.1 Arithmetic Operations

- Problem: `bit_vector`, `std_logic_vector` don't support arithmetic.
- Solution: Use `ieee.numeric_std`.
- Defines:
  - **`signed`**: For signed numbers.
  - **`unsigned`**: For unsigned numbers.
- Functions: `+`, `-`, , `abs`, `/`, `mod`, `rem`, `**` (limited).
- Prefer `numeric_std` over older `std_logic_signed` or `unsigned`.

6.2.2 Data Types

- Strong typing enforces correctness.
- **Scalar Types:**
  - Discrete: `integer`, `boolean`, `bit`, `std_ulogic`, etc.
  - Physical: `time`, etc.
- **Composite Types:**
  - Arrays: `std_logic_vector`, etc.
  - Records: Heterogeneous collections.

6.2.3 Type Conversion

- **`resize`**: For `signed`/`unsigned` vectors.
- **Type Casting:**
  - Syntax: `target_type(signal)`
  - Example: `o_int <= unsigned(a) + unsigned(b);`
- **Type Conversion Functions:**
  - Between different base types.
  - Examples: `to_unsigned(int, len)`, `to_integer(vec)`

6.3 VHDL Key Concept IX: Parametrisability of Models

6.3.1 Reusability

- **Essential for efficient design**.
- General models can be adapted via parameters.

6.3.2 Using `generic`

- Allows time-invariant parameters in `entity`.
- Example: `generic(max_count: integer := 127);`
- Used for widths, delays, strengths.
- Assigned via `generic map`:
  - `instance_name : component_name generic map (...) port map (...);`

6.3.3 Timing of Parameter Fixation

- **Design-time**: via `constant`.
- **Compile-time**: via `generic`.
- **Runtime**: via `signal` assignments.